Architectural Introspection and Applications

by

Lionel Litty

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Architectural Introspection and Applications

Lionel Litty
A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2010

Widespread adoption of virtualization has resulted in an increased interest in Virtual Machine (VM) introspection. To perform useful analysis of the introspected VMs, hypervisors must deal with the *semantic gap* between the low-level information available to them and the high-level OS abstractions they need. To bridge this gap, systems have proposed making assumptions derived from the operating system source code or symbol information. As a consequence, the resulting systems create a tight coupling between the hypervisor and the operating systems run by the introspected VMs. This coupling is undesirable because any change to the internals of the operating system can render the output of the introspection system meaningless. In particular, malicious software can evade detection by making modifications to the introspected OS that break these assumptions.

Instead, in this thesis, we introduce *Architectural Introspection*, a new introspection approach that does not require information about the internals of the introspected VMs. Our approach restricts itself to leveraging constraints placed on the VM by the hardware and the external environment. To interact with both of these, the VM must use externally specified interfaces that are both stable and not linked with a specific version of an operating system. Therefore, systems that rely on architectural introspection are more versatile and more robust than previous approaches to VM introspection.

To illustrate the increased versatility and robustness of architectural introspection, we describe two systems, *Patagonix* and *P2*, that can be used to detect rootkits and unpatched software, respectively. We also detail *Attestation Contracts*, a new approach to attestation that relies on architectural introspection to improve on existing attestation approaches. We show that because these systems do not make assumptions about the operating systems used by the introspected VMs, they can be used to monitor both Windows and Linux based VMs. We emphasize that this ability to decouple the hypervisor from the introspected VMs is particularly useful in the emerging cloud computing paradigm, where the virtualization infrastructure and the VMs are managed by different entities. Finally, we show that these approaches can be implemented with low overhead, making them practical for real world deployment.

Acknowledgements

First, I would like to acknowledge the invaluable contribution of professor David Lie to this work. Without his supervision, this work would never have seen the light of day. He was always available to discuss my progress, provide constructive criticism and suggest fruitful avenues of research. One stilted paragraph cannot possibly do justice to the part he played in my intellectual life over the past six years.

I want to especially thank the members of my supervisory committee, professor Ashvin Goel and professor G. Scott Graham. They both provided insightful feedback throughout my time as a PhD student.

Other graduate students, both in the Computer Science department and the Electrical and Computer Engineering department, were instrumental in the process of writing this work in many ways. In particular, I want to highlight the contribution of Andrés Lagar-Cavilla. I also want to thank Ekin Akkus, Kurniadi Asrigo, Lee Chew, Vladan Djeric, Tom Hart, Stan Kvasov, Jesse Pool, Ian Sin and Richard Ta-Min.

Last but not least, I want to thank my family and Jennifer Wang for their support. In particular, I still cannot believe that Jen actually spent time, sometimes late at night, proof-reading my work. Even the snarky comments were appreciated.

This work would not have been possible without the funding provided by the University of Toronto, Bell University Labs, MITACS, ISSNet and the Canadian Foundation for Innovation. In addition, VMware provided me with free licenses to use their software.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer science and engineering's short history has clearly shown that writing secure, bug-free software is a hard problem that does not appear likely to vanish in the near future. The consequences of software bugs can be severe. One of these consequences is the malicious compromise of the integrity of computer systems. Even in the absence of program bugs, users fall prey to social engineering attacks, where they are mislead into executing software created by a malicious entity. For decades, computer systems have been the target of attackers whose goal, be it for fun or profit, is to take control of systems they do not own or administer. Once in control of a system, attackers proceed to install malicious software on that system, unbeknownst to the user of the system.

This malicious software, or malware, can be used for a multitude of nefarious purposes: from stealing personal information to using the processing power and network bandwidth of the compromised machine to conduct attacks on other computer systems. These compromised computers are also increasingly used as *bots* to form large *botnets* that are used by criminal organizations for large scale illegal activities such as Distributed Denial of Service (DDoS) attacks. Unfortunately, attackers are too often successful and the scale of the problem is staggering. For example, in December 2008 and January 2009, researchers at SRI's Computer Science Laboratory [69] conducted an analysis of a large honeynet they maintain. Based on this analysis, they estimate that 4.7 Million IP addresses were infected with the Conficker.A worm, and that an even larger 6.7 Million IP addresses were infected by the next version of the worm, Conficker.B.

The last decade has seen another trend alongside this worsening computer security landscape: computer systems are becoming increasingly virtualized. On a virtualized system, the operating system (OS) runs on top of a Virtual Machine Monitor (VMM), also referred to as a *hypervisor*, rather than directly on top of the hardware. A hypervisor exports an interface to Virtual Machines (VMs) that is identical to the hardware on top of which it is run. Consequently, one can run existing, unmodified or almost unmodified operating systems inside VMs. The hypervisor isolates VMs from one another, offering few or no mechanisms for inter-VM communication. In addition, the hypervisor is isolated from the VMs it executes: the hypervisor is a thin layer of software whose role is to transparently multiplex hardware resources between VMs. As a result, there has been increasing interest in implementing security features in the virtualization layer to address the problems outlined in the first two paragraphs.

In addition, by decoupling OS images from the hardware they run on, a hypervisor enables useful capabilities such as moving virtual machines between hosts, consolidating several underutilized VMs onto a single host, and the ability to checkpoint/rollback VMs. The availability of such capabilities has lowered data center costs immensely. Recently, it has also made possible one of the many forms of *cloud computing*, sometimes referred to as *Infrastructure as a Service*, in which customers execute their OS images on hardware rented from *cloud providers* such as Amazon's EC2 service, GoGrid or Mosso. By shifting the burden of infrastructure ownership and maintenance to the provider, cloud computing allows subscribers to scale their applications and leverage large pools of resources while only covering costs proportional to their actual resource usage.

While enticing, there remains significant obstacles to the cloud computing vision. In a recent survey of 244 IT executives and CIOs, security was ranked as the number one challenge facing cloud computing [36]. Cloud computing introduces unique security challenges for cloud providers. Most importantly, the provider needs to guarantee isolation between customers. This means that a malicious customer should not only be unable to access information about other customers, she should also be unable to affect the performance of other customers. This is the guarantee hypervisors aim to provide and cloud providers can rely on a properly designed hypervisor to isolate VMs from one another.

In addition, cloud providers have to adopt some form of cloud monitoring in order to prevent their infrastructure from becoming a haven for malicious activities. Malicious activities originating from its cloud hurt the provider's reputation – and consequently its business. Moreover, malicious activities occurring on its cloud have technical consequences. For example, spam e-mails were discovered to have originated from IP addresses belonging to Amazon's EC2 service, which resulted in the blacklisting of a large swath of Amazon's IP addresses [46]. The IP addresses of the cloud provider are a shared resource that can permit one misbehaving customer to adversely affect other customers. The terms of service of the aforementioned cloud providers have broad language prohibiting illegal activities [3, 29], but the technical means that can be used to enforce these terms of services are currently ill-defined.

A cloud provider could implement network-level monitoring and control in a way similar to that of an Internet Service Provider (ISP). While this may detect some malicious activities, it is not a panacea. To wit, deep inspection of encrypted traffic is not possible. Moreover, even unencrypted malicious traffic can be made challenging to detect via network monitoring. For example, to hide a network scan an attacker may use a botnet to perform a stealthy, distributed scan. For these reasons, ISPs have had, for the most part, a great deal of difficulty protecting their networks from abuse using only network monitoring.

However, unlike ISPs, cloud providers have the advantage of being able to use *VM introspection* to assist them in analyzing VMs. VM introspection consists in examining a VM to gather information about the state of that VM. The code performing the VM introspection runs either in the hypervisor itself or in another VM, possibly with assistance from the hypervisor to allow the introspecting VM to interpose on events taking place in the introspected VM and to inspect the state of the introspected VM. Because the hypervisor is isolated from VMs and isolates VMs from one another, security monitors that use VM introspection cannot be tampered with by malware that has compromised a monitored VM.

While security monitors that use VM introspection are isolated from the potentially malicious VMs they monitor, they face a major challenge in bridging the *semantic gap*: interpreting low level events that take place in a monitored VM into higher level concepts that can be leveraged to make security decisions. Previous approaches [39, 66, 67] have proposed using detailed, expert knowledge about the internals of monitored VMs. These approaches have two major limitations. First, they require experts who have detailed knowledge of the software run by monitored VMs to specify how internal data structures should be interpreted. Second, they tightly couple the security monitoring system to the specifics of the operating system run by the monitored VM. The implicit assumption is that the guest VMs and the hypervisor are owned by the same principal, making it easier for this principal – usually the system administrator – to use knowledge they have about the guest VMs to tune VM introspection.

This assumption often does not hold in a non-cloud setting and, by design, it does not hold in a cloud setting. Ideally, cloud providers would like to place as few restrictions on their customer's VMs as possible. As a result, a cloud provider faces a significantly different introspection problem because it can make very few assumptions about the operating system version and configuration of its customer's VMs. Instead, a cloud provider must be very conservative about how it interprets its monitoring state, for fear of incorrectly labeling a benign customer as malicious.

1.1 Architectural Introspection

In this thesis, we propose that introspection should use architectural constraints placed on the monitored OS by the hardware and the external environment as a basis to bridge the semantic gap rather than dissect internal OS data structures. The architecture of the Memory Management Unit (MMU) is an example of a hardware constraint. The binary file format used by applications is an example of an environmental constraint: the compilers that create these binary files use a specification that the operating system must also follow to then load and execute them. The specifications of these interfaces are stable: they change rarely because any change to these externally specified specifications results in many different software systems needing to be updated. This is in sharp contrast to internals of OSes that can be changed with no impact on third parties. As a result, using *architectural introspection* is a more robust solution than relying on internal representations of data structures used by operating systems.

Architectural introspection [49] is a new approach introduced in this thesis. The principle behind architectural introspection is to restrict monitoring to only well-defined interfaces that are difficult or unlikely to change. In this way, architectural monitoring achieves increased robustness, because the monitored interfaces are stable. Moreover, architectural introspection is unintrusive, since it only monitors stable, low-level interfaces through the hypervisor. These interfaces are either OS-independent or OS version-independent, a property we call *OS-agnostic.*

Because monitoring is achieved by making only minimal assumptions about the OS, architectural introspection can be made tamper-evident as well. Attackers can fool previous approaches to introspection in two ways: remove the traps inserted by the monitor and violate the assumptions made by the monitor about the functioning of the OS of the monitored VM. Architectural introspection does not insert traps into the OS image, but instead relies on monitoring hardware events. This means that unless an attacker is able to prevent the monitored hardware event from taking place, architectural introspection is resilient to the first type of attack. The resilience of architectural introspection to the second type of attack will depend on the type of assumption made about the interaction between the monitored VM and the virtualized hardware. We will show that this is the case when architectural introspection is used for execution monitoring.

The increased robustness of architectural introspection comes at the price of reduced flexibility. While other approaches to VM introspection can usually be used to monitor arbitrary events, architectural introspection requires that a monitored event triggers a hardware event without having to insert a trap to cause the event. We explore the extent to which this added constraint reduces the power of architectural introspection and what kind of information can be obtained about the monitored VM while adhering to the principles laid out above.

In particular, we demonstrate that two mechanisms, *execution monitoring* and *file monitoring*, can be achieved without using information about the internals of the OS run by a monitored VM. We then discuss how these mechanisms can be applied to enable new security solutions. Because architectural introspection does not rely on the assumption that the VMs and the hypervisor are operated by the same entity, these solutions are uniquely suited to tackle security challenges that arise in a cloud environment.

1.2 Overview

The next chapter presents background material and a historical perspective on virtualization. Virtualization was first proposed for mainframes in the 1960s and was an active area of research in the 1970s. The advent of less powerful personal computers resulted in diminished interest, but the increase in computing power meant that virtualization was back in the spotlights by the second half of the 1990s. We then examine existing security systems that use introspection and their shortcomings.

Chapters 3 and 4 lay out the principles behind two architectural introspection techniques: architectural introspection for execution monitoring and architectural introspection for file monitoring.

Chapter 5 describes attestation contracts. Attestation contracts enable two parties to negotiate a contract specifying what software can be executed on their respective computers. This contract is then enforced using architectural introspection techniques.

We evaluate both execution and file monitoring in Chapter 6. Related work is discussed in Chapter 7 and we conclude in Chapter 8.

Chapter 2

Background

2.1 Virtualization

Virtualization is by no mean a new research idea. It was introduced in the 1960s and was an active area of research in the 1970s. In his seminal 1974 paper, "Survey of Virtual Machine Research" [30], Goldberg referenced more than 50 research publications that explore virtualization. In addition, interest in virtual machine technology was not confined to the research community: IBM built the System/360 Model 40 VM as early as 1965 and continued developing the VM/370 for the System/370. Besides, Goldberg lists an additional half dozen virtual machine systems that had been developed by 1974.

In a virtual machine system, a *virtual machine monitor (VMM)* runs one or several *Virtual Machines (VMs)*. The VMM is a software layer that exports an interface that is identical or at least very similar to the underlying hardware Instruction Set Architecture (ISA). The VMM can run either on the bare hardware, on top of a conventional operating system or on another VMM. VMMs running on the bare hardware are referred to as Type I VMMs [45] or native VMs [87]. In this work, we use the term *hypervisor* [8] to designate a type I VMM. VMMs running on top of an operating system are called Type II VMMs or hosted VMMs. In that case, the underlying operating system is called the *host OS* while an operating system running inside one of the VMs managed by the VMM is called a *guest OS*. Finally, the concept of running a hypervisor on top of another hypervisor is referred to as Recursive Virtualization.

Applications of virtualization technologies are manifold. Goldberg lists a number of such applications. Virtual machines can facilitate the transition to a new operating system by allowing an older operating system to be run alongside a newer one until all applications have been ported to the newer one. By allowing several OSes to be run on the same piece of hardware, VMs can help assuage tension on the operating systems to be as versatile as possible and allow special purpose OSes to be run alongside commodity ones. VMs can help in the retrofitting of functionalities to older OSes by adding these functionalities in the hypervisor. VMs facilitate debugging of OSes. VMs are also very useful pedagogical tools. Finally, VMs can help increase the reliability and security of a system because they can isolate components of the system that should not interact with each other, or limit the interaction components can have with each other. Most of these uses are particularly relevant when the computing infrastructure consists of large, time-shared main frames. But hypervisor technology has since been found to be useful in environments where the computing infrastructure is made up of a large number of smaller computers. Notable examples are server consolidation and system migration. Server consolidation helps improve the utilization of hardware resources by co-locating several servers on the same piece of hardware. System migration allows the transparent relocation of a running VM to another hypervisor on a different piece of hardware, which facilitates hardware upgrades.

Goldberg highlights that a hypervisor is different from a simulator in that most instructions are executed natively on the hardware, as opposed to being simulated in software. The term virtual machine has since then been used in numerous other contexts, such as High Level Language VMs (e.g., the Java Virtual Machine) or Co-designed Virtual Machines (e.g., the Transmeta Crusoe). As a result, the virtual machine systems as defined by Goldberg are now sometimes referred to as System VMs [87]. The rest of this work focuses on system VMs and other types of VMs will not be further mentioned.

The late 1990s saw a resurgence of interest in system virtual machines, with the introduction of several new research virtual machines: Disco [15] and Cellular Disco [31], Denali [98], UMLinux [45] and finally Xen [8]. Several commercial virtual machines, such as VMware's various products and Microsoft Virtual PC, as well as open source projects (User-mode Linux, Plex86...) also emerged at the time. These projects, with the exception of Disco and Cellular Disco, target the x86 ISA. The viability of virtualization for the PC platform resulted in widespread adoption in data centers.

2.2 Hypervisor Operation

In this section, we describe how Virtual Machine Monitors operate. To present the VM with the illusion that it is running on the bare hardware, the hypervisor needs to virtualize the CPU, memory, as well as device I/O, including storage and network. We focus on the aspects of virtualization that are relevant to VM introspection in general and architectural introspection in particular.

2.2.1 Virtualizing the CPU

Since virtualization was often not a goal of the CPU designer, it is not always possible to efficiently virtualize an architecture. Popek and Goldberg [68] formalized the requirement for an architecture to be efficiently virtualizable. This requirement is that on a dual mode machine (the two modes are referred to as supervisor mode and user mode), all sensitive instructions must be privileged instructions. Privileged instructions are instructions that can only be run when the processor is running in supervisor mode, the more privileged CPU mode. These instructions will result in an exception being raised when run in user mode, the less privileged CPU mode. In a virtual machine system, VMs execute in user mode. This causes privileged instructions executed by the VM to trap. The hypervisor can then emulate these instructions and effect the possible changes they would make to the system's state to the virtual machine's state instead.

Sensitive instructions can be either control-sensitive or behavior-sensitive [87]. Control-sensitive instructions modify system-wide states, while behavior-sensitive instructions are affected by system-wide states. An example of a control-sensitive instruction is an instruction that changes the CPU mode. An example of a behavior-sensitive instruction is an instruction that returns 0 if the CPU is currently in user mode and 1 if it is currently in supervisor mode.

The hypervisor has to emulate sensitive instructions, because system-wide states are virtualized. But if sensitive instructions are not privileged, these instructions will not trap when executed in user mode. Popek and Goldberg's requirement stems from these observations. An architecture that does not satisfy this requirement is not efficiently virtualizable, at least by Popek and Goldberg's definition of efficient, that is, most instructions are executed natively on the hardware, not emulated.

The main architecture in use today, which is the focus of this thesis, the Pentium ISA or IA-32, is unfortunately not efficiently virtualizable [72]. For example, the POPF instruction modifies the content of the EFLAGS register, a register that contains a number of flags that control the operations of the CPU. One of these flags, the Interrupt Flag (IF), disables interrupt and can only be changed while in supervisor mode. Nevertheless, if POPF is run in user mode, the modification of the IF will simply fail silently (other flags in the EFLAGS register may be changed successfully). As a result, the OS of the VM, which is now running in user mode, will silently fail to disable interrupts, which may result in incorrect operations. Robin and Irvine found that 17 out of approximately 250 instructions[1] were sensitive but were not privileged [72]. Some examples can be found in Table 2.1.

As a result of these limitations of the IA-32, hypervisors for this architecture have to modify

[1]Robin and Irvine analyzed the Pentium processors up to the Pentium III.

Instruction	**Functionality**
POPF	Pops 16 bits from the stack and stores them in the EFLAGS register.
SMSW	Store the content of the machine status word to memory.
VERR	Test if a segment is readable in the current mode.
VERW	Test if a segment is writable in the current mode.
PUSH	Can be used to read the CS and SS registers (to access the current mode).

Table 2.1: Example non-privileged sensitive instructions of the Pentium ISA.

the binary code (*dynamic binary rewriting*) or the source code (*paravirtualization*) of the OS to be able to run this OS. In both cases, the goal is to remove sensitive instructions that are not privileged from OS code. These modifications allow running system code at a different privilege level without affecting its correctness. Privileged instructions are replaced by calls to the hypervisor. The hypervisor then emulates the outcome of the privileged instruction on behalf of the virtualized operating system. In addition, the hypervisor ensures that the emulated instruction cannot affect either the hypervisor itself or other VMs executing on the machine.

Dynamic binary rewriting is a complex task that can introduce significant overhead and paravirtualization is a labor-intensive task that requires modifying the target OS. Moreover, paravirtualization is not possible when the source code of the target OS is not available. As a result, both AMD and Intel have recently added hardware support for virtualization to their architectures to make them more easily and efficiently virtualizable. When hardware support for virtualization is present, neither dynamic binary rewriting nor paravirtualization are necessary. Instead, a simpler hypervisor can run unmodified OSes. The Xen hypervisor can run both *Paravirtualized (PV)* domains and *Hardware Virtual Machine (HVM)* domains. In this thesis, we assume that hardware support for virtualization is present and focus on HVM domains, which takes advantage of the technology.

VT-x is the extension Intel designed for the IA-32 architecture [93]. When the virtualization mode of operations is turned on, a CPU can be in either *VMX root mode* or *VMX non-root mode*. The hypervisor will typically run in root mode while guest operating systems will run in non-root mode. The hypervisor explicitly switches from root to non-root mode and has the ability to specify which events will cause an exit from non-root mode. For example, the hypervisor can indicate which hardware interrupts result in an exit from non-root mode. On exit, the hardware indicates to the hypervisor exactly why the VM exited. The hypervisor would act upon this information and then resume execution of the VM. Both the hypervisor and guest VMs can now use the four rings provided by the Pentium ISA and the hypervisor no longer needs to claim part of the VM's linear address space. AMD-v, the extension designed by AMD, operates in a similar manner. These extensions significantly facilitate the development of hypervisors, but their implementation is currently slower than software binary translation [2].

2.2.2 Virtualizing memory

On systems using paged virtual memory, also referred to as *paging*, all memory accesses occur using virtual addresses. These virtual addresses are translated to physical addresses using mapping kept in page tables. Most commonly, the OS keeps one set of page tables per process and the set of addresses that can be accessed through these page tables is known as the address space of the process. Paging is used by the OS to give each process the illusion that it owns the entire physical memory and to isolate processes from one another: a process can only access physical memory that is mapped in its address space.

On machines using the IA-32 architecture, the translation is performed in hardware by the *Memory Management Unit (MMU)*. The page tables are organized in a multi-level hierarchy that is maintained by the OS. The root of this hierarchy is kept in a configuration register, the *CR3*. The content of this register can only be changed by privileged code. This happens when a context switch occurs. Virtual addresses are divided into an index within the page table hierarchy and an offset within the physical page that corresponds to the page table entry for that virtual address. Pages are 4 KB on IA-32 machines. Each page table entry contains permission bits: a *present* bit that indicates whether the entry is valid and can be used to translate physical addresses, a *supervisor* bit that indicates that a page can only be accessed in supervisor mode and a *writable* bit that indicates that a page can be written to. In addition, an *NX* bit was introduced in 2004 by both Intel and AMD to indicate whether the content of a page can be executed. When code attempts to perform a memory access that is not permitted by the set of permission bits associated with the accessed page, the CPU generates a software exception referred to as a *page fault*.

In a virtualized environment, an additional level of indirection is introduced so that each virtual machine is given the illusion that it owns all physical memory. The virtualized OS is unaware of this additional level of indirection and still maintains *guest page tables* for each of its processes. These page tables now translate virtual addresses to guest physical addresses. The guest physical addresses then need to be translated to machine addresses. To this end, the hypervisor maintains a set of *shadow page tables*. The shadow page tables mirror the page tables maintained by the virtual OS, but contain virtual to machine address translations. To keep the shadow page tables in sync with the page tables used by the OS, the hypervisor makes sure that the entries in the shadow page tables that correspond to these page tables are not writable by the virtualized OS. All writes by the OS to these page tables cause page faults.

When these page faults occur, the hypervisor emulates the attempted writes and updates the shadow page tables to reflect the modifications. In this example, the permission bits set by the hypervisor in the shadow page tables are more restrictive than those set by the virtualized OS. More generally, this technique allows the hypervisor to regain control when the OS attempts actions that are not allowed by the more restrictive permission bits set by the hypervisor. Since page faults can be caused by the hypervisor, the hypervisor determines the cause of each page fault by comparing the permission bits in the guest page tables with the permission bits in the shadow page tables. The hypervisor handles page faults it created and forwards page faults created by the virtualized OS by injecting virtual page faults.

We note that some processors provide an optional *nested page table* mechanism. When this mechanism is present, the hardware is made aware of both levels of indirections. A memory access by a VM now results in a two stage address translation. First, the virtual address is translated to a guest physical address using page tables maintained by the guest OS. Second, the guest physical address is translated to a machine address using page tables maintained by the hypervisor. As a result, the hypervisor only needs to maintain one set of page tables translating guest physical addresses to machine addresses for each VM, rather than mirroring all of the guest page tables in shadow page tables, which are no longer needed. However, in this work, we focus on hypervisors that use shadow page tables.

2.2.3 Virtualizing persistent storage

In a virtualized environment, persistent storage is provided by virtual block devices. In Xen, virtual block devices are accessed differently depending on whether a PV domain or an HVM domain is used. PV domains use a paravirtualized disk driver while HVM domains interact with emulated devices. The devices are emulated by a user space QEMU [11] process that runs in Domain 0. Domain 0 is a special VM used to support the Xen hypervisor. It is the only VM that has access to the physical devices on the machine and the interactions with the devices of all other VMs must happen through Domain 0. The QEMU process faithfully emulates a physical disk in software, exporting an interface that is identical to a common device that is supported by the virtualized OS. As a result, it is not necessary to install a special driver. However, HVM domains can also use PV drivers when front end drivers are available for the virtualized OS and have been installed on the HVM domain. These three possible scenarios are illustrated in Figure 2.1.

In all three scenarios, the virtual block device driver can be thought of as having two halves: a front-end executing in the PV or the HVM domain, and a back-end executing in Domain 0. The front-end signals the back-end when a data transfer is requested by the virtualized OS. The back-end then performs the data transfer to or from the disk using Direct Memory Access

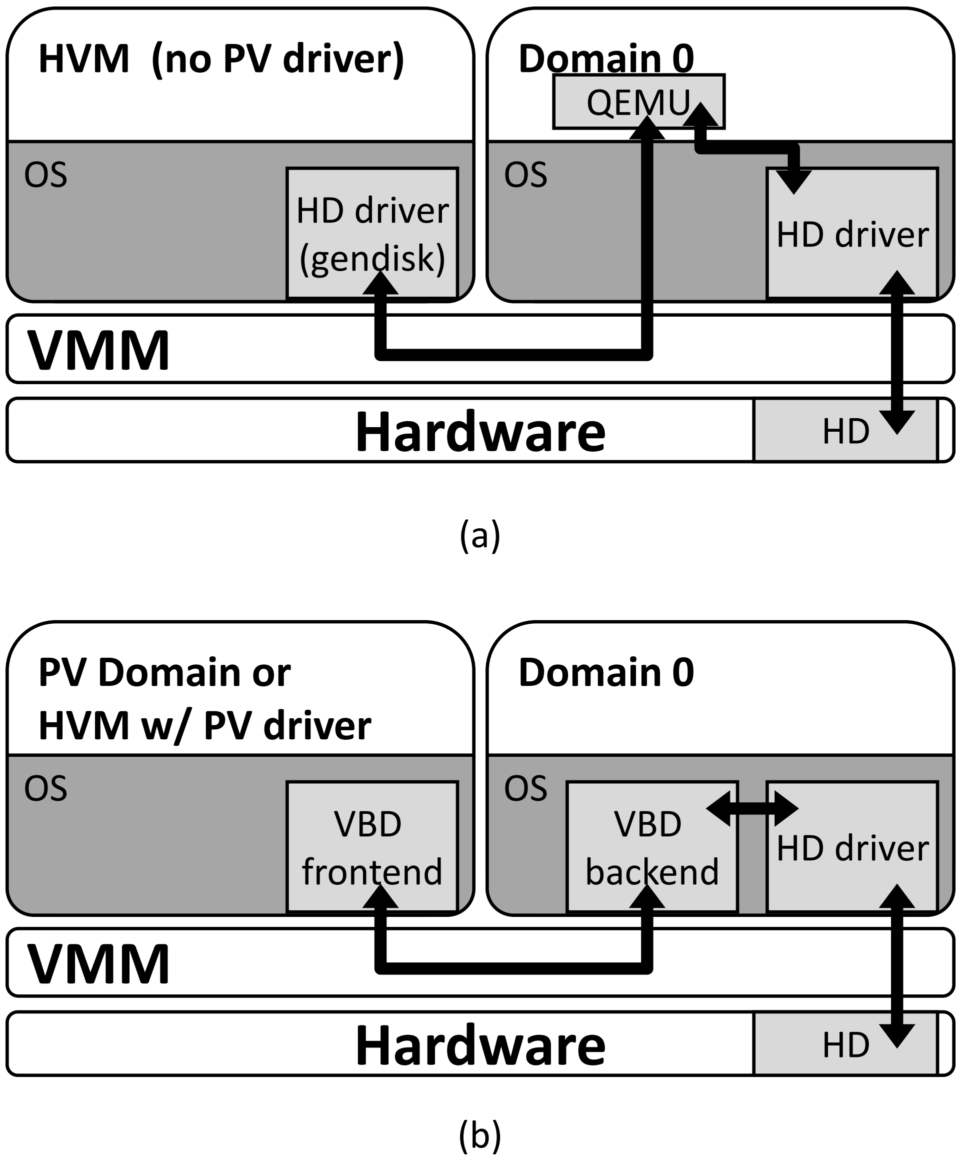


Figure 2.1: Virtual disk architecture for (a) an HVM domain without PV drivers and (b) a PV domain or an HVM domain with PV drivers. In the HVM domain case, a QEMU process is used to emulate a generic hard drive device. In the PV case, the virtual block device (VBD) driver is split between a back-end and a front-end.

(DMA). When QEMU is used, the HVM domain is not explicitly signaling the QEMU backend. Instead, when the guest OS device driver sends DMA requests to the virtual disk, they are intercepted by Xen and forwarded to the QEMU disk emulator in Domain 0. QEMU then services the requests by accessing the physical disk and responding to the guest OS via Xen.

We note that disk drives can be accessed in either of two ways: via Programmed I/O (PIO) or via Direct Memory Access (DMA). DMA mode minimizes the involvement of the CPU in data transfers and therefore reduces the CPU load when performing IO. It also offers faster transfer speeds. As a result, DMA is the preferred access method and is used by all modern OSes.

2.3 VM Introspection

2.3.1 The Semantic Gap problem

The resurgence of virtual machine research and the availability of free, open source virtual machine monitors triggered a renewed interest in using virtual machine monitors for security purposes. In a seminal 2001 position paper [17], Chen and Noble argue that widespread use of virtualization technology will help improve the security of computer systems. Adding functionalities to the hypervisor layer provides opportunities for security applications that will be isolated from the operating system and that can be made *OS agnostic* by only relying on aspects of the hardware.

Chen and Noble propose two such hypervisor-based security applications: secure logging and intrusion prevention and detection. They note that logs produced by the OS are not safe, since an attacker could tamper with them after having compromised the system. Even if logs are output to append-only media, the attacker could turn off logging right after taking control of the system, depriving the system administrator of precious forensic information. In addition, these logs may be incomplete, in that they do not include information that would have been of interest. Secure logging from the hypervisor can address these shortcomings. Logs produced by the hypervisor are isolated from the logged VM. In addition, the hypervisor witnesses all external input to the VM. By logging this input, the hypervisor has all the information necessary to reproduce what happened on the VM, and therefore to analyze security incidents.

Chen and Noble also argue that hypervisor-based intrusion prevention and detection can address failings of security monitoring applications that run within the monitored system or that analyze the network input to the monitored system. Hypervisor-based security monitors are isolated from the monitored system, a clear advantage over in-system monitors. Moreover, hypervisor-based security monitors have more visibility than network-based ones, in that they can observe all aspects of the system, including keyboard and disk input. In addition, hypervisors offer a powerful tool when trying to prevent intrusions: one can clone the monitored VM and try potentially malicious input on the cloned VM before forwarding it to the primary VM. Such a technique gives the monitoring system a glimpse into the effect some input will have on a VM and may enable it to prevent the delivery of malicious input. Chen and Noble concede that such an approach may significantly impact the overall responsiveness of the system, but highlight that for traffic whose latency can be high, such as email traffic, the cost may be acceptable.

These security applications are similar to security applications that monitor the network interface, such as firewalls or network intrusion detection systems (NIDSs) [62, 88]: Firewalls and NIDSs can be isolated from the system they are monitoring by running on another physical machine and they are OS agnostic, since they focus on aspects of network protocols. Network-based security applications and hypervisor-based security applications are similar in a third respect: they have to address the *semantic gap* between the data they are observing and the effect this data will have on the system they are monitoring. For example, NIDSs need to parse packet headers to reconstruct data in a form identical to that in which it will be delivered to the application, notably by re-assembling fragmented IP packets.

2.3.2 Bridging the semantic gap via VM introspection

Chen and Noble highlight that a key challenge for hypervisor-based security systems is to bridge the semantic gap that exists between low-level events observed by the hypervisor and high level OS concepts. On the one hand, low-level events observable by the hypervisor include device input/output, such as keyboard input, network accesses and disk accesses; the instruction stream executed on the CPU; and memory accesses. On the other hand, security decisions are made in terms of high level OS concepts. For example, a security decision may be whether a given process, such as the `Apache` web server process, can open a given file, such as the file `/etc/passwd`.

One way to bridge the semantic gap is to use *virtual machine introspection*: peer into the VM and interpret its state. VM introspection is challenging, because a VM typically contains an enormous amount of code and data. To make introspection tractable, it makes sense to adopt a reductionist approach that seeks to divide a VM into smaller, manageable components and analyze them individually. Since a running VM can naturally be conceived of as a set of processes collaborating to perform tasks, the process boundary is a natural way of compartmentalizing events in a VM. This approach also follows naturally from the OS's attempt to isolate untrusting principals into separate processes.

The capabilities of introspection are determined by the nature of the VMs being monitored. In this work, we distinguish between two types of VMs: on the one hand, *Non-malicious VMs*, which may be vulnerable to attacks but have not yet been compromised. On the other hand, *Malicious VMs*, which are under the control of an attacker. In the case of malicious VMs, the attacker may have complete control of the VM: they have acquired super-user privileges and are able to load code into the VM's kernel to modify its behavior.

While the idea of a VM as a set of distinct processes that can be analyzed separately is attractive, a malicious VM can easily blur the boundaries between processes, defeating a reductionist approach. Attackers who gain administrative privileges in a commodity OS are usually able to circumvent the process-level protections implemented by the OS. For example, an attacker can use the `ptrace` debugging facility on Linux to manipulate the memory and alter the behavior of another process. Nearly every other operating system (including Windows) provides such powers to an attacker who gains administrative control.

As a result, a reductionist approach is only applicable to non-malicious VMs and malicious VMs where the attacker is able to exert only limited control over a single process. However, before an attacker fully compromises a VM, he initially controls only a single process. This represents a window of opportunity to detect that the VM has transitioned from non-malicious to malicious. This window will close once the attacker has gained sufficient control of the VM to defeat monitoring.

Intrusion prevention, that is, detecting the transition from a benign to a malicious state is one goal of a hypervisor-based security system. Another goal is *intrusion detection*: Trying to detect that a VM is malicious. Since the monitored VM is now malicious, the attacker may actively try to defeat the monitoring. To be most effective in such a scenario, the security system should be *tamper-evident*, meaning that the security system will either report complete and accurate information, or it will report that the information is incomplete. This means that an attacker controlling the VM cannot fool a tamper-evident system into reporting misleading information. Systems that are not tamper-evident are still useful: they report accurate information for non-malicious VMs. In addition, they may also report accurate information for compromised VMs when the attacker is unaware of the existence of the monitors, or when the attacker has only gained partial control of the VM being monitored.

Below, we examine two introspection approaches that have been proposed in the literature and their limitations: trap and inspect, and checkpoint and rollback. We examine these approaches along three axes: power, unintrusiveness and robustness.

We define the *power* of an approach as the scope of VM events it can monitor as well as its ability to interpose on specific events. The *unintrusiveness* of an approach characterizes how much disturbance it introduces in the monitored VM. Lastly, the *robustness* of an approach depends on the nature of the assumptions made about the monitored VMs and how likely these assumptions are to hold. This includes robustness to different versions of an OS or even different

OSes, as well as robustness to OS modifications as a result of an update, the introduction of new code in the form of a module or even malicious changes aimed at evading the monitoring.

Trap and Inspect

In a Trap and Inspect approach [7, 47], a monitor inserts traps in the monitored VM code to be alerted on specific events, much like a debugger. The monitor code executes either in the hypervisor or in another VM. If the monitor resides in another VM, it enlists the help of the hypervisor to place traps in the monitored VM's code and to gain access to the monitored VM's memory. This has the advantage of isolating the introspection code from the introspected VM, preventing tampering with the code and avoiding interference with the execution context of the VM. With adequate support from the hypervisor, this is a powerful approach that is afforded full visibility over the introspected VM.

However, the monitor needs to insert traps in the VM code, much like a debugger. And like a debugger, this requires additional information to locate which instructions in code memory should be replaced by traps, as well as a thorough understanding of the code. Because trap and inspect depends on knowing minute details about the code being monitored, it is not a robust introspection techniques. Trap placement for access control has been shown to be a complex problem that is hard to get right [90, 100]. Moreover, bridging the semantic gap requires inspecting data structures in memory, another task that requires expert knowledge of the OS used by the VM. The complexity of trap and inspect makes it brittle in the face of skilled attackers. As a result, trap and inspect is an engineering heavy task that is closely tied to a specific version of an operating system, resulting in an approach with low robustness.

If one is willing to lower accuracy, it is possible to do without traps and instead scan the content of memory periodically [65–67]. However, the asynchrony of this approach means that it is possible for malware to avoid detection by periodically removing traces of its presence from memory.

Checkpoint and Roll Back

To alleviate the need to understand the memory layout of the data structures of the OS of the monitored VM, checkpoint and roll back takes advantage of the ability of the hypervisor to perform these operations on VMs. The introspection monitor can checkpoint a VM, inject code that can call any support function provided by the OS of the monitored VM and then roll back the VM to the checkpoint [43]. As a result, this approach subsumes trap and inspect: it is similarly powerful while being more robust, since it can leverage OS APIs to perform queries as opposed to reverse-engineer in-memory data structures. While these APIs are usually more stable than the minute details of data structure layout in memory, they remain internal to the OS and more likely to change than external specifications. In addition, checkpoint and roll back still relies on traps to be invoked.

Chapter 3

Architectural Introspection: Execution Monitoring

Execution monitoring consists in identifying all running binary code in a VM. This capability is natively provided by many operating systems. For example, the Linux kernel reports the code loaded in the address space of each process in `/proc` and provides execution-reporting utilities such as `ps` and `lsmod`. Likewise, there exists numerous execution-reporting utilities, such as Process Explorer [74], that provide the same information for Windows. Trap and inspect approaches also exist for execution monitoring [67].

In the case of execution monitoring, we show that architectural introspection is as powerful as other introspection techniques. In this chapter, we present Patagonix [48, 50], a system that ensures that no binary code can be covertly executed on the monitored system. Patagonix uses the processor MMU to receive notifications whenever binary code is executed, and identifies the code using the binary format specification. The Patagonix monitor is tamper-evident, because it never misidentifies code. If there is code that it does not recognize, either because it is malicious or because it is in a form that it cannot understand, it will report it as unidentified. The approach is OS-agnostic: any OS can be monitored, and if this OS uses an executable file format understood by the monitor, the executed code will be identified.

Patagonix addresses two shortcomings of the execution-reporting utilities mentioned above. First, these utilities all depend on the integrity of the kernel, both as a source of information and for protection against tampering. However, since attackers can use rootkits to subvert the kernel, the trust that these utilities and the administrator invest in the kernel is misplaced. Second, these utilities do not verify the integrity of the binaries they report as executing. This shortcoming allows a rootkit to covertly execute code by injecting malicious code into a running binary or by tampering with the binary image on disk. Utilities that monitor binaries on disk, such as Tripwire [44], may detect tampering of on disk binaries, but will miss tampering of binaries once they are loaded in memory.

As discussed above, these approaches require expert knowledge of the monitored OS. Similar to host-based monitoring, an attacker can circumvent trap and inspect monitors. Trap and inspect relies on the execution of instrumented instructions to invoke the monitor. If the attacker can insert code in the kernel (by loading a module for example), she can create new code paths that will not be monitored.

In this chapter, we make three main contributions:

- **Patagonix Prototype.** We have implemented a Patagonix prototype that leverages the capabilities of a hypervisor and the non-executable (NX) bit of the Memory Management Unit (MMU) to detect and identify all executing binaries regardless of the state of the OS kernel. Our prototype, built on the Xen 3.0.3 hypervisor [8], makes no assumptions about the OS kernel. As a result, with the exception of the binary format information, which differs from OS to OS, it can be used to neutralize rootkits on Windows XP, Linux 2.4 and Linux 2.6 OSes without modification.

- **Identity Oracles.** The semantic gap between the hypervisor and the OS requires special support to differentiate legitimate modifications made to running code by the OS from malicious ones made by a rootkit. To differentiate legitimate modifications from malicious tampering, we introduce the concept of an *identity oracle*, which when given a page of code in memory and a database of binaries, will either identify the binary from which the code page originated, or indicate that the code page is not from any of the binaries in the database. We have designed an oracle construction framework and implemented identity oracles for ELF binaries, PE binaries, the Linux kernel, the Windows XP kernel, and Windows driver interrupt handlers.

- **System Usage** We present two complementary usage modes for Patagonix. In *reporting mode*, Patagonix serves as a trusted replacement for the standard execution-reporting utilities of an OS, allowing the administrator to see all executing processes even if hidden by a rootkit. This augments the administrator's ability to audit the state of the system during regular inspections and after an attempted rootkit removal. In *lie detection mode*, Patagonix compares the executing binaries reported by the OS with the executing binaries it identifies and reports any discrepancies to the administrator [27].

3.1 Assumptions and Guarantees

To provide security guarantees, Patagonix relies on two properties of the hypervisor. First, Patagonix assumes that the hypervisor will protect both itself and Patagonix from tampering by a rootkit that has subverted the OS kernel. This assumption is consistent with the guarantees that hypervisors aim to provide. Second, Patagonix relies on the hypervisor to provide a secure communication channel between it and the user. Patagonix uses this channel to inform the user of what binaries it detects are running. Because the hypervisor is the only principal with direct access to the hardware, this channel can be provided in a straightforward way by providing separate consoles for the OS and Patagonix.

Patagonix identifies executing binaries by the cryptographic hash of the executing code. To convey this information to the administrator in a useful way, these hashes must be mapped to the name of a file or application. Extracting this mapping from the disk image is not trustworthy since a rootkit can tamper with the disk. Instead, Patagonix relies on a trusted database to provide such a mapping. This database is assumed to contain the names of all legitimate software binaries that the administrator has installed on the machine and can also optionally contain mappings of known malicious binaries. Any executing binary that does not match one in the database is identified as "not present" and should be scrutinized by the administrator. Publicly available databases currently exist – for example, our prototype could use the NSRL [58]. We note that the labeling of binaries as legitimate or malicious is made available purely for the convenience of the administrator and is not used by Patagonix. History has shown that such labeling may be flawed – there have been many documented cases of trojaned, vulnerable, or patently malicious binaries being distributed by reputable entities [34]. Patagonix correctly handles situations where malware is executing on the OS because it was incorrectly labeled as legitimate in the database. For example, Patagonix can be used to confirm that the incorrectly labeled application is no longer executing after an attempted removal.

Even with malware in control of the OS, Patagonix guarantees that it is able to identify and report all executing binaries. Rootkits may try to hide malware binaries from the administrator by either appropriating the name of a legitimate application, or by trying to make it invisible. Patagonix prevents the former by using mappings from the trusted database. This also defeats any attempts to inject malicious code into legitimate binaries on disk or in memory since this will alter the contents of the code when it executes. If the rootkit tries to hide the execution of a binary by subverting the OS kernel or execution-reporting utilities, Patagonix will still identify and report the executing binary to the administrator since Patagonix monitors the processor hardware for executing code, not the OS kernel. With these guarantees, Patagonix can report the identities of all executing binaries to the user in reporting mode. Correspondingly, in lie detection mode, it can notify the administrator of any discrepancies between the code it detects and that reported by the OS. Table 3.1 summarizes the information reported by Patagonix.

	Patagonix will identify as		
	Legitimate	**Malicious**	**Not Present**
Legitimate binary	Always	Never	Never
Malicious binary	Never	If present in database	If not present in database

Table 3.1: Guarantees Patagonix provides for the detection and identification of binaries executing on a system.

3.2 System Architecture

3.2.1 Overview

The architecture of Patagonix is illustrated in Figure 3.1. The majority of Patagonix is implemented in the *Patagonix VM*, while a small amount of functionality that requires kernel mode privileges is implemented in the hypervisor. The *Monitored VM* contains the *Monitored OS* for which the administrator wants trustworthy binary execution information and the hypervisor protects Patagonix from tampering by the monitored VM. While implementing Patagonix entirely within the hypervisor may reduce performance overhead, splitting the functionality of Patagonix into hypervisor and VM components has the benefits of increased modularity, ease of portability to a different hypervisor, and a reduction on the size of the code being added to the security critical hypervisor. As we shall see in Section 6.1, the boundary crossings between the hypervisor and VM components of Patagonix have a minimal impact on overall performance.

The Patagonix VM contains three components. First, several identity oracles, one for each type of binary in the monitored VM, enable Patagonix to identify pages of code that are executed in the monitored VM. The identity oracles use cryptographic hashes of binaries from the trusted database to identify binaries executing in the monitored VM. Second, a *management console* implements the interface between the user and Patagonix. Finally, the *control logic* coordinates events between the management console, the oracles and the hypervisor component of Patagonix.

Only the identity oracles are binary-specific as one must be written for every binary format used by the OS in the monitored VM. All other components, which we collectively refer to as the *Patagonix Framework*, are independent of both the OS run by the monitored VM and of the binary formats used by this OS. An identity oracle for a given binary format can be used across OSes that use this binary format.

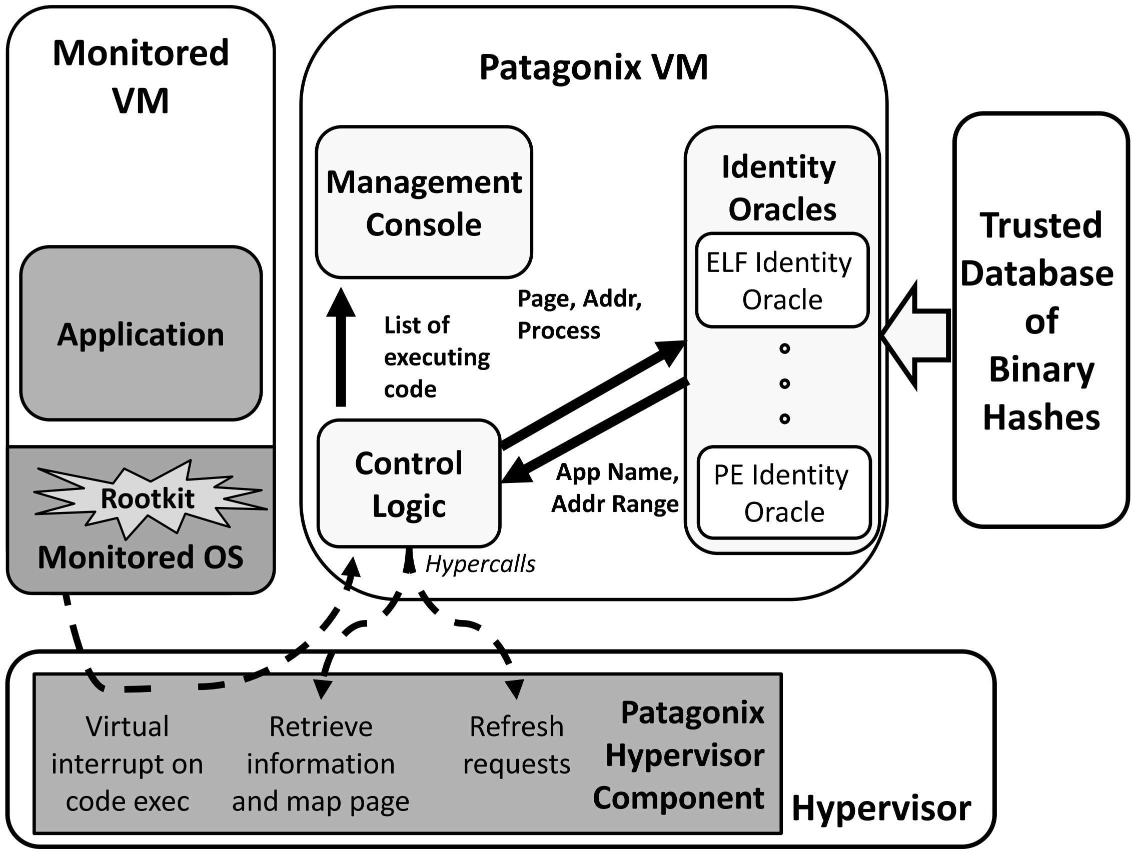


Figure 3.1: The Patagonix architecture.

3.2.2 Patagonix Framework

The Patagonix framework has three main responsibilities. First, the framework must detect when code is being executed in the monitored VM. Second, when code execution is detected, it invokes the identity oracles to identify the code and maintain a list of executing code. The identity oracles will either match the executing code to an entry in the trusted database, or will indicate that the identity of the code is not present in the database. Finally, the framework is responsible for conveying these results to the user in a way that is free of tampering by malware in the monitored VM.

Detecting code execution is performed by the Patagonix hypervisor component using the non-executable (NX) page table bit, which is available on all recent AMD and Intel x86 processors. When set on a virtual page, this bit causes the processor to trap into the hypervisor component whenever code is executed on that page. The hypervisor component then informs the control logic in the Patagonix VM by sending it a virtual interrupt.

Frequent traps into the hypervisor will hurt performance so Patagonix uses the processor to only inform it when either code is executed for the first time, or code it has already identified changes and is executed. To identify code when it executes for the first time, the hypervisor component initially sets the NX-bit on all pages in the monitored VM so that it will receive a trap from the processor when a code page is executed. When it receives such a trap, the hypervisor component invokes the Patagonix VM to identify the code and then clears the NX-bit on the page, making it executable. At the same time, to detect if the identified code is subsequently modified, the hypervisor component makes the page read-only by clearing the writable bit in the page table. As long as the page remains unchanged, subsequent executions of code on that page do not cause a trap. If the identified code is modified, the processor will trap into the hypervisor, at which time the hypervisor component will make the page writable but non-executable again. If the modified code is executed, the hypervisor component will again receive a trap, at which point it will use the Patagonix VM to re-identify the code. To eliminate the possibility of a race where the rootkit alters the code page after it is identified, but before it is made executable, the monitored VM is paused while the Patagonix VM identifies the executing code. Setting executable or writable privileges on entire pages at a time is fairly straightforward. However, pages that contain mutable data and code require the ability to prevent writes to the code portions of the page and execution for the data portions of the page. While this can be implemented with additional hardware, we have been able to emulate such support in software. We defer the details of the solution to Section 3.4.2.

To identify code in memory, the identity oracles require the contents of the code page being executed, the virtual address at which the page is located, and the process the code comes from.

The control logic retrieves this information via new *hypercalls*, which are hypervisor analogs of OS system calls we have added to Xen. The control logic then passes this information to each of the identity oracles, which either return the identity of the binary from which the code originated, or indicate that the identity of the originating binary is not in the trusted database. We note that Patagonix does not use OS process IDs to identify processes as these are controlled by the OS and can be subverted by a rootkit. Instead, Patagonix identifies a process by its virtual address space, which is an equivalent hardware proxy since by definition there is a one-to-one relationship between OS processes and address spaces. A process' address space is denoted by the base address of its page table hierarchy, which is maintained in a dedicated register on x86 processors.

Because the hardware only reports when code is executing, rather than when it is not going to be executed any more, the control logic records the most recent time it observed each binary execution and periodically instructs the hypervisor to perform a *refresh*, i.e., set all pages as non-executable. Code that is no longer executing will not trigger any more traps. Patagonix does not infer process termination by observing when a page table does not contain any valid mappings like Antfarm [40] because malware that controls the OS can toggle the page table bits between valid and invalid without actually removing the process from memory, thus circumventing this process termination heuristic.

The control logic uses the management console to securely report the list of observed executing binaries and times they were last observed executing. Because the hypervisor has control over the hardware, it is able to provide the management console in the Patagonix VM with an interface separate from that of the monitored VM, thus ensuring that the monitored VM cannot tamper with the output of the Patagonix VM.

3.2.3 Identity Oracles

Executable binaries are mapped from disk into memory by a *binary loader*, whose behavior is governed by the binary format that it loads. The task of the identity oracles is to use the information provided to them to reverse the transformations that the loader applies to binaries, and identify which binary in the trusted database (if any) the page of code being executed originates from.

Aside from the information provided to the oracles by the hypervisor component, the oracles also require information about the binaries in the database they are trying to match against. For example, information such as hashes of each individual code page in the file and information about relocations are required depending on the particular format of the binary. While current binary databases generally only contain hashes of binary files, additional information can be extracted from files on disk after they have been authenticated using the trusted database.

Each oracle initially collects such information by searching the disk of the monitored VM for all executable binaries. The authenticity of an executable file is verified when its hash is found in the database, and the oracle can then proceed to extract additional information from the file. Patagonix needs to rescan the disk each time binaries are added, or alternatively, a program in the OS can be used to gather information about new binaries as they are introduced into the system. If an executable file is hidden from Patagonix by a rootkit, Patagonix will not have the necessary information to identify executing code from this binary and thus will not be able to match code originating from these binaries against entries in the database. As a result, such code will be identified as "not present", thereby indicating to the administrator that a rootkit is likely on the system. In either case, access to the trusted database itself must be free of tampering by the rootkit. We implement our prototype database by combining hashes from the NSRL database, hashes from signed RPM packages and hashes computed from pristine binaries directly into the Patagonix VM image. Had the database been maintained remotely, it would need to be accessed over a secure, authenticated channel such as one offered by SSL.

Once the information about the binaries is acquired, the main challenge for the oracles is to reverse the transformations done by the loader without trusting information from the OS. Formally, each binary loader can be modeled as a function $L(B, S) = (\mathbf{M}, \mathbf{A})$, which maps a particular binary B, and the state of the OS at the binary load-time S, to a set of memory pages $\mathbf{M}$ and a set of addresses $\mathbf{A}$. $\mathbf{M}$ denotes the set of possible executable pages that the loader may transform the binary into and $\mathbf{A}$ denotes the possible virtual addresses at which the loader may place the transformed binary. The oracle for a particular binary format is a function $O_L(M, A, P) = \mathbf{B}$, which given a page M detected as executing by the hypervisor, the virtual address of the executing code A, and the process it was executing in P, produces a set of binaries $\mathbf{B}$, from which the page could have originated. Since M and A are produced by the loader, they are elements of sets $\mathbf{M}$ and $\mathbf{A}$ respectively. One cannot implement O_L by only relying on S, since a rootkit can subvert S. This inability to safely infer S represents the semantic gap that the identity oracles bridge. Since we do not know S, O_L's task can be generalized to searching the set $\mathbf{MA'}$ for the observed code page and address (M, A), where $\mathbf{MA'}$ contains all code page/address combinations that the loader could have generated for all binaries and all legitimate OS states.

$\mathbf{MA'}$ can be very large, making the performance cost of a naïve search impractical. For example, in Windows, a code page can be mapped at 2^{20} possible locations (for a 32-bit address space when using 4KB pages) and its contents will be different for each of those possible locations. If applied to code pages in all binaries in an average Windows installation, this would result in an $\mathbf{MA'}$ several terabytes in size, which would be overly expensive to search. To reduce these costs, we exploit two characteristics that every binary format we have examined exhibits.

The first is that these formats specify that code sections should be mapped to contiguous regions of memory. As a result, once the binary that occupies a memory region in a process is known, the oracle only needs to check that other code executing in the same region is the appropriate page in the same binary, eliminating the need to search $\mathbf{MA}'$ in these instances (in this case, binary can refer to a program binary or a dynamically linked library). Knowing the address where a binary is mapped also enables the oracle to reverse run-time modifications and derive the original code page, eliminating the need to store all versions of the page. To establish what binary occupies a region, the oracle exploits the second characteristic: binary executables have only a few entry-points (usually only one), which are executed before any other code in the binary. As a result, if code executes in a memory region where the oracle has not identified a binary before, the oracle only has to check for code at pages containing entry-points in $\mathbf{MA}'$. This reduces the search space, and also adds a desirable security check since the oracle will identify code as "not present" if the malware tries to jump into a binary at any point other than a legitimate entry-point. We use these assumptions about binaries as hints to improve the performance of Patagonix. However, Patagonix does not trust these hints, so its security guarantees are not affected – tampering with the binaries that violates these assumptions will result in the tampered binary being identified as "not present".

Figure 3.2 illustrates our oracle construction framework. Four components in the framework are binary loader specific. The first is an *entry-point database*, which contains information on the entry-points of known binaries. This database is searched using an entry-point *search function*. The other two components are the *code database*, which contains information on the rest of the code sorted by binary, and the code *check function* which checks code against the code database. An oracle invocation begins with the control logic forwarding the page contents, faulting virtual address and process to the oracle. The oracle first checks whether the virtual address and process of the code are from a region where the binary is known. If not, then the binary has just started executing because no code has been observed executing at this location before. The oracle searches the entry-point database for a match to identify the binary. If a match is found, it records the binary's name and memory range it should occupy and returns the name of the binary. Otherwise, the oracle identifies the code as "not present" in the database.

If the address is from a memory region whose binary has been previously identified, then the oracle checks that the executing page is from the associated binary. If it is, the oracle returns the name of the binary. If it is not, then the binary no longer occupies that memory range in that process. The memory region record is removed and the oracle searches for the page in the entry-point database.

We have observed cases of related binaries containing identical code pages. If there have not been enough pages executed to uniquely identify the binary, the identity oracles return a

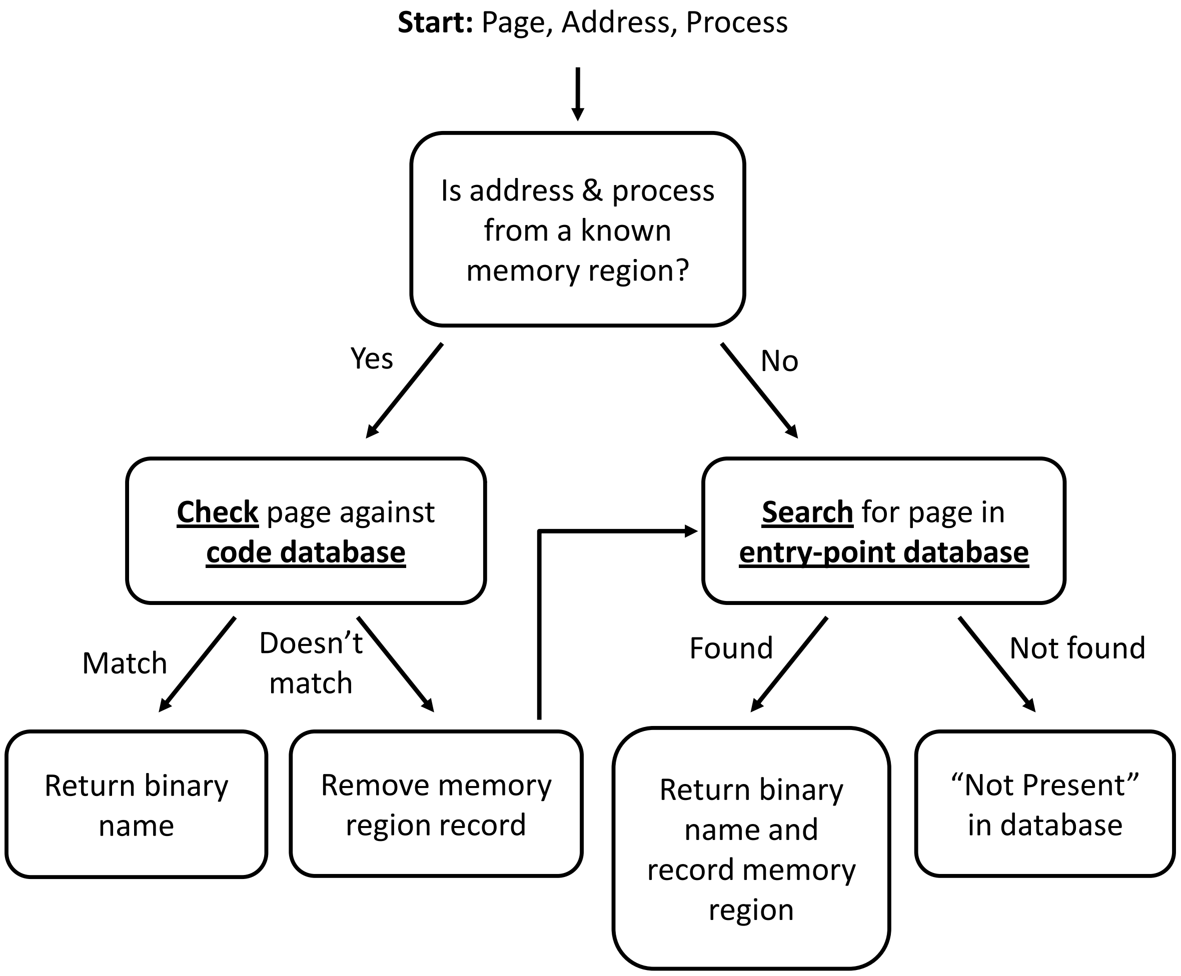

Start: Page, Address, Process
Is address & process from a known memory region?
Yes
No
Check page against code database
Search for page in entry-point database
Match
Doesn't match
Found
Not found
Return binary name
Remove memory region record
Return binary name and record memory region
"Not Present" in database

Figure 3.2: Identity Oracle framework. The functions and databases that are binary loader specific have been underlined.

list of candidate binaries until a unique page of code is executed. Should a page contain a mix of data and code, the oracles also return the sub-page range of the code.

3.3 Oracle Implementation

In this section, we describe the oracles we have constructed for various binary formats and their loaders. We find that while binary formats may differ, the operations performed by the loaders of these formats have similarities, allowing common techniques to be used across the oracles for different formats. We classify our oracles into two categories based on the type of binaries they identify. The first category consists of oracles for application code in Linux and Windows. We discuss support for the two main methods for dynamic code loading: position independent code and run-time code relocation, both of which are represented in the ELF and PE formats used by Linux and Windows respectively. The other category consists of kernel code in Linux and Windows. This code poses some extra challenges because both kernels contain self-modifying code. However, our oracles are able to verify that they are applied correctly. Finally, we finish this section with a discussion on the generality of our identity oracles.

3.3.1 Application Binary Oracles

ELF Oracle

The Executable and Linkable Format (ELF) [92] is used by Linux, as well as other OSes such as Solaris, IRIX and OpenBSD. An ELF file is divided into segments and contains a program header table that specifies the address at which each segment should be mapped into memory. ELF segments in the binary are identical to the segments that will be loaded in memory and no run-time modifications are required from the loader. Code in executable segments can either be relocatable, meaning it can be loaded at any address in memory, or non-relocatable, meaning that it must be loaded at a particular address. All references to absolute addresses in relocatable code go through indirection tables, which are filled in by the run-time linker. ELF shared libraries are typically relocatable, while executable binaries are typically non-relocatable.

Since ELF shared libraries use position independent code, both ELF libraries and ELF applications are mapped from disk into memory without any modifications, making this our simplest oracle. To populate the entry-point database for the ELF oracle, pages containing entry-points are placed in the database – all shared objects have an `init` subroutine that is run when the shared object is loaded and executables always begin execution in `_start`. To save space, the ELF oracle does not store the entire page contents in the database, but instead stores a cryptographic hash (SHA-256) of the page instead. The hashes are stored in a sorted list and the entry-point search function computes the hash of the page where code execution was detected and searches the entry-point database for a match.

The code database stores hashes of all pages for each binary in a two dimensional array that is indexed first by binary and second by page offset from the beginning of the binary. The check function uses the binary name attached to the memory region to compute the first index in a look up and the offset of the executing page from the start of the memory region to compute the second index. A hash of the executing page is then compared to the hash from the code database. Because we use a collision-resistant cryptographic hash function, any tampering of the binary will result in the binary being identified as not present.

PE Oracle

The Portable Executable (PE) format [53] is used in all versions of Windows after Windows NT 3.1. Similar to ELF files, PE files have a header table that describes how sections in the file should be mapped in memory. However, code in PE files contains absolute addresses, and thus is not position independent. All PE files have an image base, which indicates the *preferred address* for loading the file. If an application needs to load two or more Dynamically Linked Libraries (DLL) that occupy overlapping preferred address regions, the OS must *relocate* one or more of the binaries. To do this, the absolute addresses in the executable must be adjusted by adding the offset between the preferred address and the actual address where the binary is loaded. This relocation operation is performed by the OS using the information stored in the binary header. Not all PE binaries contain relocation information – application EXE files (as opposed to DLLs) usually lack this information and must be loaded at their preferred addresses.

PE binaries pose two challenges. First, because the OS may adjust the absolute addresses in a binary, one cannot directly use page contents to identify code pages in the entry-point database. Instead, the PE oracle exploits the fact that the PE loader only relocates binaries by 4KB page offsets, meaning that the offset of the entry-point from the top of the page (i.e. the page-offset) is always the same. Thus, the entry-point database is indexed by the page-offset of the entry point and contains the locations of the absolute addresses in each candidate page, as well as a hash of its contents. The search function then searches the entry-point database for the page-offset of the faulting address to determine the binary.

In some cases, several binaries may have the same entry-point offset, so the search function must find the matching page within a set of more than one candidate pages. For each candidate, the search function undoes the absolute address adjustments made by the OS during relocation. This is accomplished by making a copy of the executed page and subtracting the relocation offset from each absolute address. This offset is the difference between the entry-point address of the executed page and the entry-point address of the candidate if it were mapped at its preferred address. A hash of the copy can then be compared against the hash of the candidate.

The second challenge is that some PE binaries have memory pages that contain both code and mutable data. For example, the Import Address Table (IAT), which is used to dynamically link DLLs against an application, is typically put in the code section by the Microsoft compiler. As a result, the search function only uses the portions of these pages that contain code to identify them, and will notify the control logic, which in turn will instruct the hypervisor to make only the identified portions of the pages executable. Naturally, the entry-point database entries for these pages must also contain information listing what portions of the page contain code.

The rest of the PE oracle is straightforward. The code database and check function are also similar to the ELF oracle except that they must undo any relocations before comparing the page contents and they must account for pages that only partially contain executable code. Thus, the code database also stores the preferred address with each binary, and the locations of all absolute addresses and sub-page code ranges (if necessary) with each page entry. To undo the relocations, the check function uses the actual address the binary was mapped in at, which is given by the start address of the memory region record, and then uses the same technique as the entry-point search function. In this way, the PE oracle provides the same guarantees as the ELF oracle.

Address inference algorithm

While the PE oracle described in the previous paragraphs works well, it may fail to identify code when a VM is not monitored from boot time, even if this code is present in the code database. This is because it relies on the fact that the first instruction that will be executed for any binary or library is the instruction at the entry point address. When a checkpointed VM is resumed, the first instruction executed by an application that was executing when the VM was suspended will not be the entry point for that application. Therefore, the code will only be identified if the page does not contain relocated code or a mix of code and data. If we want to monitor checkpointed VMs that may have been checkpointed by hypervisors that do not perform execution monitoring, we need to remove this limitation. To this end, we extended the PE oracle with an algorithm that can identify applications that are in mid-execution. This algorithm is based on *address inference.*

Note that this algorithm is more complicated and more costly than the one previously described and will only be used in scenarios that require handling checkpointed VMs. If the VM was previously monitored by a hypervisor that implements execution monitoring, state that pertains to execution monitoring can be checkpointed alongside the VM. However, this is not possible in scenarios involving migration of checkpointed VMs between distinct hypervisors where the originating hypervisor does not implement Patagonix.

The challenge is to identify the portion of a memory page that contains code and to undo the relocations performed by the loader without relying on the assumption that the first executed instruction in a binary will correspond to the entry point address in the PE header. Our solution relies on the observation that absolute addresses used in a binary will be addresses that fall within the binary itself. Let S be the size of the binary, and a, the address at which the code execution is detected. Since we know that a points to valid code and a is somewhere within a binary of size S, then all absolute addresses in the code segment of the binary must fall within the range $[a - S, a + S]$.

While we do not know S, we can use the size of the largest binary in the database, s, as an upper-bound for S (i.e., $S \leq s$). The algorithm then identifies any sequence of 4 bytes that fall within the range $[a-s, a+s]$ as a *candidate address*. When a RISC design is used, instructions are aligned and have fixed sized op-codes. However, there is no alignment requirement for x86 instructions, and instruction op-codes have a variable length. As a result, it is not possible to identify which part of the page consists of operands and which part of the page consists of instructions. Instead, the algorithm considers all overlapping 4-byte sequences on the page.

As a result, some of the identified sequences will not be absolute addresses, but instead simply sequences of instructions or instruction parts that happen to fall within the range $[a - s, a + s]$. To get an idea of the likelihood of such a *false inference* in general, assume that the byte values in the page are randomly distributed. The probability that a 4-byte candidate address will happen to fall within the range $[a - s, a + s]$ is

$$p = \frac{2 * s}{2^{32}}$$

. We remark that this probability increases linearly with s and that this is only an estimate, as code bytes are not randomly distributed. For example, if we use 4 MB for s, we get a 0.002 probability that a candidate address is not actually an absolute address.

To identify the binary from which the code originates, the algorithm makes a copy of the byte sequence and sets all candidate addresses to zero. A hash of this sequence is then computed and checked against the database of binaries, which contains hashes of sequences where the absolute addresses have also been set to zero. To address false inferences, all subsets of the candidate addresses are considered (i.e., the power set $\mathcal{P}$ of candidate addresses). This results in 2^k possibilities, where k is the number of candidate addresses. The algorithm starts by zeroing out all candidate addresses and searching for a match in the binary database. If no match occurs, it then tries all combinations where one candidate address is not zeroed out, then two candidate addresses are not zeroed out and so on... The search stops as soon as the sequence is matched with a binary in the database. However, in the worst case, 2^k combinations need to be tested.

Since a page consists of 4096 bytes, k could be as large as 4096, causing 2^k to be extremely large. This would make the algorithm intractable. Rather than consider the entire page, the algorithm considers only substrings of length l within the page. Only considering substrings also enables the algorithm to handle pages that contain both code and data. l should be chosen so that strings will have a small number of candidates but be long enough to occur only in at most a few binaries. In our implementation, we used l=64 bytes.

The l-length substrings should be taken at several offsets to provide a spread of offsets throughout the page so that at least one will fall in a code region should a page contain a mix of code and data. At the same time, none of the substrings should straddle a page boundary. In our implementation, we use offsets o in the following order: 0x3; 0x1000 - 0x3 - l; 0x103 + i*0x100, $i \in [0, 14]$. The code database is augmented with an index that contains hashes of the first l-length string at offset o in each binary. The database construction procedure searches for this string by trying different o's in the order given above.

During identification, all combinations of candidate addresses are searched. If a match occurs, then all relocations in the page are undone and a hash over the entire page is computed and checked to verify that the match is indeed correct. Because the hashes on the substrings are taken frequently, our implementation uses the non-cryptographic, fast murmur2 hash [4] for the hash on the substring and a sha256 hash on the whole page to verify the match.

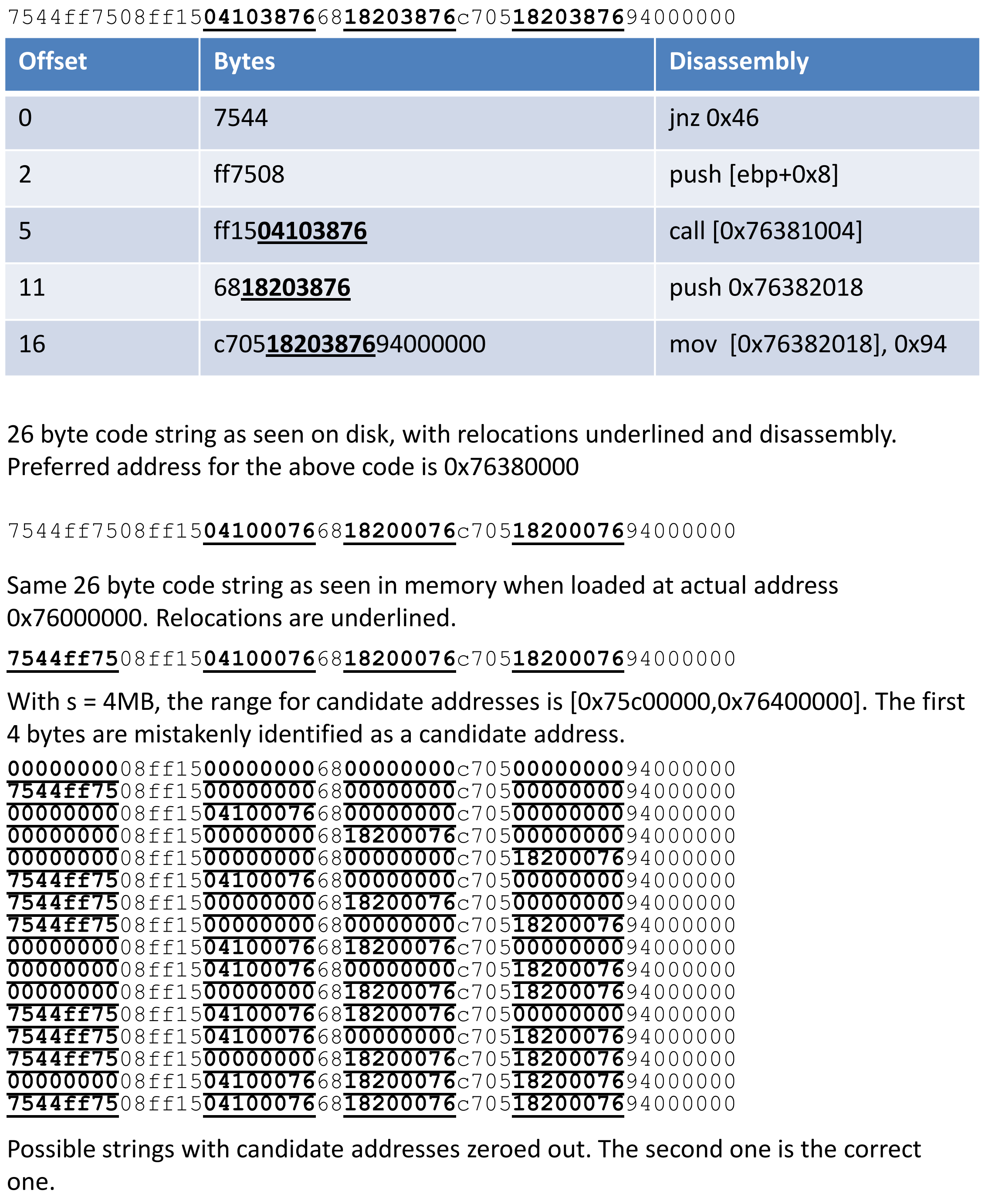
The process of obtaining candidate strings is illustrated in Figure 3.3. The address inference algorithm is summed up in pseudo-code in Algorithm 1.

3.3.2 Kernel Binary Oracles

Linux Kernel Oracle

The Linux kernel's code pages in memory are not always identical to their on-disk representation. Recent versions of the Linux kernel customize their binaries at run-time depending on the availability of more efficient instructions for the CPU the kernel is executing on. For example, the kernel will implement memory barriers with `LFENCE` and `MFENCE` instructions if running on newer x86 processors with SSE2 extensions. Altering these instructions at run-time allows a single kernel binary to be used on different CPUs. In addition, the Linux kernel can dynamically load and unload kernel modules at run-time.

The aspects of the Linux kernel that differentiate it from application code are self-modifying code and the ability to dynamically load modules. However, both of these can be handled with the techniques used in the PE oracle. In the Linux kernel, the locations of customizable instructions, the instructions they can be replaced with, and the conditions to permit replacement are



7544ff7508ff15**04103876**68**18203876**c705**18203876**94000000

Offset	Bytes	Disassembly
0	7544	jnz 0x46
2	ff7508	push [ebp+0x8]
5	ff15**04103876**	call [0x76381004]
11	68**18203876**	push 0x76382018
16	c705**18203876**94000000	mov [0x76382018], 0x94

26 byte code string as seen on disk, with relocations underlined and disassembly. Preferred address for the above code is 0x76380000

7544ff7508ff15**04100076**68**18200076**c705**18200076**94000000

Same 26 byte code string as seen in memory when loaded at actual address 0x76000000. Relocations are underlined.

7544ff7508ff15**04100076**68**18200076**c705**18200076**94000000

With s = 4MB, the range for candidate addresses is [0x75c00000,0x76400000]. The first 4 bytes are mistakenly identified as a candidate address.

0000000008ff15**00000000**68**00000000**c705**00000000**94000000
7544ff7508ff15**00000000**68**00000000**c705**00000000**94000000
0000000008ff15**04100076**68**00000000**c705**00000000**94000000
0000000008ff15**00000000**68**18200076**c705**00000000**94000000
0000000008ff15**00000000**68**00000000**c705**18200076**94000000
7544ff7508ff15**04100076**68**00000000**c705**00000000**94000000
7544ff7508ff15**00000000**68**18200076**c705**00000000**94000000
7544ff7508ff15**00000000**68**00000000**c705**18200076**94000000
0000000008ff15**04100076**68**18200076**c705**00000000**94000000
0000000008ff15**04100076**68**00000000**c705**18200076**94000000
0000000008ff15**00000000**68**18200076**c705**18200076**94000000
7544ff7508ff15**04100076**68**18200076**c705**00000000**94000000
7544ff7508ff15**04100076**68**00000000**c705**18200076**94000000
7544ff7508ff15**00000000**68**18200076**c705**18200076**94000000
0000000008ff15**04100076**68**18200076**c705**18200076**94000000
7544ff7508ff15**04100076**68**18200076**c705**18200076**94000000

Possible strings with candidate addresses zeroed out. The second one is the correct one.

Figure 3.3: Example of deriving candidate strings from a string observed in memory.

Algorithm 1 The address inference algorithm.

```
function identify_code(string $code$, int $a$, int $l$, int $s$)
$offset\_list$ = [0x3, 0x1000 - 0x3 -l, 0x103, 0x203, ..., 0xf03]
for $offset \in offset\_list$ do
  $candidates \leftarrow \emptyset$
  for $i \in [0, l+3]$ do
    $value$ = integer value of 4-byte sequence of $code$ from $offset{+}i - 3$ to $offset{+}i$
    if $value \in [a - s, a + s]$ then
      $candidates \leftarrow i$
    end if
  end for
  for $candidate\_set \in \mathcal{P}(candidates)$ do
    $candidate\_string \leftarrow code[offset..offset{+}l[$ with sequences in $candidate\_set$ zeroed out
    $hash \leftarrow$ hash of $candidate\_string$
    if $hash$ is in database then
      $candidate\_pages \leftarrow$ pages in index that match $hash$
      for $candidate\_page \in candidate\_pages$ do
        $relocated\_code \leftarrow code$ with relocations undone according to $candidate\_page$ relocation information.
        if hash of $candidate\_page$ stored in database == hash of $relocated\_code$ then
          return binary
        end if
      end for
    end if
  end for
end for
```

stored in special sections of the kernel binary. Using this information, the search and check functions make a copy of the page, verify that the substitutions are legitimate, and then undo them by replacing them with the default on-disk instructions. The pages are then hashed and compared against the entries in the databases.

Linux kernel modules can be loaded at any location in memory and have both relocations and customizations that are adjusted at load-time. They also contain an initialization function that can serve as an entry-point for the module, making their loader very similar to that of a PE DLL. As a result, much like in the PE oracle, the Linux kernel oracle uses an entry-point database consisting of entry-point offsets. Once a kernel module is identified, the memory range it occupies is recorded.

Windows Kernel Oracle

The Windows kernel exhibits behavior similar to the Linux kernel, where some of its code pages are customized at run-time by patching the kernel code. In addition, Windows also permits run-time loading of kernel modules and drivers.

Unlike the Linux kernel, the Windows kernel's replacements are not specified in the kernel binary, but are applied in an ad hoc fashion by various functions throughout the kernel. However, since these customizations are deterministic for a given hardware platform and occur early during boot, it is possible to record the customizations from a pristine kernel and use these to verify the customizations in the monitored VM. While this approach cannot guarantee completeness (for example, we do not know what replacements will take place for other hardware), we believe that a developer with more information about the Windows kernel customizations would be able to exhaustively enumerate the transformations the kernel performs at run-time. The Windows kernel oracle handles the run-time loading of drivers in exactly the same way as the Linux kernel oracle.

Both the Linux kernel oracle and the Windows kernel oracle provide the same guarantees as the ELF and PE oracles. While the PE oracle validates relocations by using the difference between the actual address and the preferred address, the kernel oracles perform an equivalent validation for run-time customizations by ensuring that modified instructions are replaced with legitimate substitutes.

Windows Interrupt Handler Oracle

To allow drivers to register interrupt service routines, the Windows kernel provides an *interrupt object* abstraction. To allow for driver portability, when such an interrupt object is initialized by the driver, 106 bytes of kernel-specific code is copied from an interrupt handling template into the object, and will be executed whenever an interrupt associated with the object occurs [75].

While this appears to be a form of dynamic code generation, it is actually very easy to write an oracle that identifies the Windows Interrupt Handler. The code is shorter than a page, so it can be efficiently identified and validated in its entirety with one oracle invocation. As a result, the Interrupt Handler oracle does not need a code database or check function. Furthermore, the code is exactly the same every time it is copied except for an 8 byte field that contains run-time parameters and absolute addresses, which is customized for each driver. As a result, no entry-point database exists for this oracle, and the search function simply performs a byte-by-byte comparison of the code starting at the faulting address with the 106 byte template. If there is a match, the code is identified as a Windows Interrupt Handler and only the 106 byte region is made executable and non-writable.

Our prototype oracle currently does not perform further checks on the 8 bytes that are modified dynamically by the kernel. This means that an attacker can arbitrarily modify these bytes. However, this is a small amount of memory, and these bytes are not contiguous. A more sophisticated oracle could also validate the contents of these bytes.

3.3.3 Discussion

To better understand the generality of the approaches we have employed for our prototype oracles, we examined descriptions of other common binary formats and loaders. We found that for application code, the main reason for run-time code modifications is to support the need to be able to dynamically load libraries at any base address. Nearly every binary format we examined, which included common formats such as the Mac OS X Mach-O format, the COFF format used by SysV, and a.out, uses either position independent code or rebasing – both of which we are able to handle.

Another interesting class of loaders are executable packers. They incorporate code into a compressed binary to decompress the code just before execution. As a result, the compressed binary needs to be unpacked first before the oracle gathers information from it. This extra step is conducted when Patagonix adds a packed binary to the code database. Our prototype currently only handles binaries that have been packed using the popular UPX [60]. To support additional packers, Patagonix only needs to be provided with an unpacker. For example, Patagonix could use PolyUnpack [73] to automatically support a large number of executable packers.

Finally, we observed two binaries that are not JIT compilers but that dynamically generate code: `winlogon.exe`, which authenticates users, and the Windows Genuine Advantage application, which checks the Windows OS for evidence of piracy. No formal specification exists for the code generated by these applications and there is evidence that the code is generated to obfuscate self-integrity-checking operations. Without more information (like we had for the

Windows interrupt handlers) or reverse engineering (which would violate the EULA), we cannot build an oracle that validates the legitimacy of the generated code. Thus, these binaries are treated as JIT compilers – we can identify that they are executing, but do not examine other code pages in their address space.

3.4 Framework Implementation

We used the Xen 3.0.3 hypervisor as a basis for building our Patagonix prototype. When used in Hardware Virtual Machine (HVM) mode, Xen utilizes virtualization support in x86 processors to run unmodified operating systems, including both Linux and Windows. With the exception of our emulated sub-page privileges support, our implementation of Patagonix can run on both AMD and Intel processors. In implementing Patagonix, we found that while the MMU provides a way to efficiently detect code execution, care needs to be taken to ensure that all code execution in the monitored VM is detected. Another shortcoming of the processor support was the inability to allow or deny execution or write pages at a sub-page granularity. Finally, we discuss a performance optimization that reduces the number of Patagonix VM invocations the hypervisor must make.

3.4.1 Detecting Code Execution

The non-executable permission bit was primarily implemented to allow an OS to prevent unauthorized code execution. When this mechanism is virtualized, there are two issues that must be taken into account to ensure that all instances of new code execution are detected by the hypervisor.

The first issue arises from the fact that page permission bits apply to a virtual page mapping and not to a physical page. Since there can be more than one virtual mapping for a physical page, our hypervisor modifications must ensure that there cannot be writable and executable mappings of a physical page simultaneously. Otherwise, the rootkit could use one mapping to modify the page and the other to execute it. We accomplish this by leveraging Xen's frame map, which maintains a count of the number of mappings of each physical page. Whenever a page changes from writable to executable or vice versa, Xen consults the count in the frame map to see if any other virtual mappings need to be updated appropriately. Xen's frame map only maintains a count of the number of mappings, and is not a reverse frame-map; as a result, we must walk the page tables to find and change all other mappings.

This issue could also be fixed by upcoming nested-page table (NPT) support, which provides full hardware virtualization support for page tables. NPTs add a shadow page table, which allows the hypervisor to specify a second translation between the guest physical frame numbers and the actual machine frame numbers. With this, the hypervisor could simply control the permissions for the machine frames, removing the need to track the number of guest virtual mappings for each physical page. To be notified when new code is executed, Patagonix marks pages as non-executable in the shadow page table, and then makes them executable after they have been identified. We do note that in doing this, Patagonix will negate one of the possible advantages of NPTs, which is to allow superpage mapping of a contiguous set of guest physical frames with a single NPT entry.

The second issue stems from the fact that the virtual Direct Memory Access (DMA) unit in Xen runs in a separate protection domain (the privileged `Domain0`) and thus is not constrained by the page access restrictions placed on the rest of the monitored VM. Malware that is aware of this could abuse the virtualized DMA to modify memory pages that have been marked as executable and read-only. To make sure that memory content was always checked before being executed, we modified the emulated DMA devices to inform the hypervisor when they write to any pages. If any of these pages are marked as executable, Xen makes these pages non-executable again.

3.4.2 Sub-page support

Sub-page permissions are necessary when a memory page contains a mix of identified code and mutable data: the code must be made non-writable, and the data must be made non-executable. Ideally, sub-page support would be provided in hardware using a scheme such as Mondrian memory [99] or Transmeta's Crusoe processor [23]. However, because such support is not available on x86 processors, we devised a method to emulate this support based loosely on a technique that Van Oorschot et al. used to circumvent code tampering detection [94]. The technique takes advantage of the separate Translation Lookaside Buffers (TLB) for instructions (ITLB) and data (DTLB) present in x86 processors.

Our solution maps an execute-safe version of the page to a virtual address for instructions, and the original to the same virtual address for data. The execute-safe version is a copy of the mixed page where the data sections have been made non-executable by replacing them with trap instructions. A mapping to this version is loaded into the ITLB by temporarily setting the shadow page table entry to be executable, pointing it to the execute-safe version and executing a single instruction from that page. After that, the shadow page table entry is switched back to the original page and made writable and non-executable. This emulates the sub-page permission control we require since any attempt to execute at an address from the data regions will go through the ITLB and result in a trap, and any modifications to the code region will go through the DTLB and will not be applied to the page that instructions are being fetched from. To ensure that the execute-safe page is not accidentally loaded into the DTLB by an unintended load or store while setting up the TLBs, Patagonix disables interrupts for the monitored VM during this operation.

The emulation has some drawbacks over native hardware support. First, the emulation does not trap into the hypervisor when a write is attempted to a code region. Such functionality would be needed to deal with run-time modifications to a mixed page, but we have not found this necessary in practice. Second, this TLB manipulation needs to be undertaken every time to correctly load the ITLB mapping for this page, ITLB misses for such pages are transformed into page faults that require two traps into the hypervisor. Finally, while this functionality can be emulated on AMD processors, it cannot be emulated on Intel processors because, at the time of writing, Intel processors flush both TLBs on every crossing between the hypervisor and the VM.

3.4.3 Performance Optimizations

The dominant source of overhead in Patagonix is the page faults that occur when the monitored VM executes pages marked non-executable by Patagonix and the subsequent Patagonix VM invocation to identify the newly executing code. Some of these page faults are unnecessary because the executing code is on a physical page that has already been identified when it was executed in another process. Thus, we added an optimization that avoids the extra page fault and Patagonix VM invocation for pages whose identities are already known. This is accomplished by maintaining a list of physical pages that have been identified and whose virtual mappings are all executable and non-writable. When the monitored VM attempts to map such a page as executable in a new process, Patagonix preemptively makes the new mapping executable and non-writable.

The hypervisor must log each time this optimization is applied for two reasons. One reason is that this information is required to maintain the consistency of the memory region information for the oracles. The second reason is that this information is required by the Patagonix VM to maintain an accurate record of when pages from each binary were observed executing. To avoid extra domain crossings but keep the Patagonix VM's view of the monitored VM current, this log is read by the Patagonix VM whenever it is invoked by the hypervisor to identify a page, whenever it requests the hypervisor to perform a refresh and whenever the user requests a list of executed binaries through the management console. As a result, this optimization has no effect on how current the Patagonix VM's information on executing binaries is, and thus has no impact on the security guarantees of Patagonix.

3.5 Usage

Patagonix has two usage modes. In *reporting mode*, Patagonix provides trustworthy execution-reporting information and is functionally similar to utilities such as `ps`, `lsmod` and the `task manager`. This gives the system administrator a trustworthy alternative information source when evaluating if their system has processes hidden by a rootkit, or whether an attempted rootkit removal has been successful. In *lie detection mode*, Patagonix compares the list of executing binaries reported by the monitored OS with what it detects is executing. Differences mean that the OS is lying and indicate that a rootkit is present on the system.

When in reporting mode, Patagonix displays a list of all executing binaries on the management console. This is semantically similar to the list displayed by utilities such as `top` or the `task manager`. Patagonix also displays the times they were last observed executing. The administrator can also use Patagonix to terminate or suspend the execution of all instances of a binary by issuing commands to the management console, creating a trustworthy version of the UNIX `kill` utility. To terminate a binary, Patagonix sets all pages of that binary to non-executable. When an execution fault occurs on one of the code pages, Patagonix replaces the instruction at the faulting address with an illegal instruction. This makes it appear to the monitored OS that the binary tried to execute an illegal instruction, causing the monitored OS to terminate it. Suspending execution is achieved by replacing the code with an empty loop instead of replacing it with an illegal instruction. Thus, the binary is still executing from the OS' point of view, yet no code from the actual binary is being executed. A more efficient, but OS-specific implementation could inject code that causes the application to sleep.

In lie detection mode, Patagonix compares execution information reported by the monitored OS with its own list of executing binaries. Patagonix obtains execution information from the monitored OS via an agent in the monitored VM. The agent is a program that queries the monitored OS via standard interfaces to obtain a list of executing processes. Previous systems that performed lie detection in this way can suffer from false positives due to asynchrony between the measurement of running processes taken from within the monitored OS and the measurement taken from the hypervisor – a new process may begin executing and be detected by the hypervisor before the OS has had a chance to update the information it exports to the agent [27, 39]. To avoid this, Patagonix's agent registers a function with the OS kernel that synchronously informs Patagonix of process creation via a hypercall. Both Linux and Windows provide facilities for this.

Patagonix's lie detection detects both OS under-reporting (hiding executing binaries) and over-reporting (reporting binaries that are not actually executing). Usually, rootkits under-report to hide the execution of malicious binaries, but over-reporting could also be used maliciously. For example, a rootkit may wish to lead the administrator to believe that a critical program (such as an anti-virus scanner) is still running when it is not. Over-reporting requires the administrator to specify a threshold which dictates how long Patagonix will allow a binary that is reported as executing by the OS to be not observed running any code before declaring it as being over-reported.

3.6 Limitations

The goal of Patagonix is to provide a trustworthy alternative to traditional OS execution-reporting utilities, thus denying rootkits the ability to hide executing binaries from the administrator. However, detecting and preventing the exploitation of vulnerabilities is outside the scope of Patagonix. For example, Patagonix does not detect attacks that do not inject new code, but instead alter the control flow of an application, such as in a return-to-libc attack [84]. More generally, neither Patagonix nor traditional execution-reporting utilities prevent legitimate applications from taking malicious actions as a result of malicious inputs. For example, the attacker can cause a legitimate interpreter or a just-in-time (JIT) compiler to perform malicious actions by using it to run a malicious script. Despite this, Patagonix provides strong and useful guarantees. While Patagonix cannot tell if a script is malicious or not, it guarantees that the administrator will be aware of all executing interpreters and JITs.

Identifying and verifying the integrity of interpreters is the same as other binaries because all the machine level instructions that can be executed by the interpreter are known a priori. However, this is not the case for JIT compilers because they dynamically generate and execute code whose content can be heavily dependent on the workload and run-time state. Thus, once Patagonix identifies a program as a JIT compiler, it will ignore pages it observes executing in the JIT compiler address space that are not present in the trusted database (JIT compilers must always execute code from their binary before any dynamically generated code, so Patagonix is always able to identify the process first). While a rootkit may exploit this to inject arbitrary code into the JIT compiler and escape any sandboxing enforced by the JIT compiler, Patagonix's guarantees still hold because the rootkit will not be able to hide the execution of the JIT compiler, nor can the rootkit cause Patagonix to misidentify the JIT compiler as another application.

Finally, as mentioned earlier, Patagonix used in lie detection mode is not a generic rootkit detector: it focuses on rootkits that hide executing binaries.

Chapter 4

Architectural Introspection: File Monitoring

Once one has identified what code is executed by processes running in a VM, the logical next step is to identify which files are being accessed by these processes. For example, as highlighted earlier, execution monitoring does not provide any information about interpreted or dynamically generated code. For instance, Java may be used to run Eclipse or Tomcat, and the PHP interpreter may be used to run phpBB. Once a process has been identified as an interpreter or JIT, knowledge of which files it has read will allow decision making based on the scripts or byte-code it is executing.

Just as with execution monitoring, a variety of host based introspection solutions already exist and are supported by most OSes. Examples of host-based agents providing these features include `filemon` on Windows and `strace` on Linux. These tools intercept either system calls or library calls to determine which files are being accessed by a given process.

Likewise, an approach based on trap and inspect can be used to obtain this information. For instance, one can trap the system calls used to access files and then inspect their arguments. Since the arguments may be relative path rather than absolute path, one could then ask the OS to resolve relative paths and use checkpoint and rollback to ensure that the query does not interfere with the proper functioning of the monitored VM. Alternatively, the path can be resolved by parsing memory data structures that will contain the necessary information to resolve the path: the current working directory for the process, the file system mount point, etc.

Difficulties with this approach arise, just as with execution monitoring, when the VM is running an unknown OS or even a different version of a known OS with different data structure layouts or code paths. In addition, these approaches are prone to Time Of Check To Time Of Use (TOCTTOU) races [13] because the path resolution is performed twice: once by the security monitor and once by the OS.

In this chapter, we demonstrate that architectural introspection can be used to identify which process is accessing specific files that one wants to monitor. The key idea is to hash data blocks when they are read from disk by the virtual block device driver. Using a precomputed database of files of interest, this will enable identification of blocks that belong to files inserted in the database, such as JAVA class files or PHP scripts.

However, this only provides the monitor with the knowledge that the VM accessed the file. In many cases, it is not sufficient to know that the VM accessed a monitored file. One would also like to know which process accessed that file. For example, should the goal be to detect that the phpBB application is being executed by the PHP interpreter, flagging accesses due to an anti-virus scanner or a file indexer would constitute false positives.

As a result, we need an additional mechanism to map file accesses to specific processes: once blocks of interest are identified, we track them in kernel memory to determine if information flows from the files into the address space of a binary of interest, such as a JAVA virtual machine. To make such information flow tracking OS-agnostic, we again leverage the processor MMU. By manipulating the page table entries that correspond to monitored blocks, we can ensure that a hardware trap will occur when the page is accessed. We also instrument the virtual block device driver to track the destination of blocks read off monitored disks. In this way, we can detect when a block read off disk is accessed by a process.

Our current prototype relies on two assumptions about the OS. First, that data from the disk is accessed via page sized blocks using Direct Memory Access (DMA). Second, that if a process accesses data read from disk, this access occurs while the process accessing the file is scheduled. This second assumption holds unless the operating system kernel asynchronously accesses the data.

Finally, we describe how file monitoring can be used to monitor an Infrastructure as a Service cloud for the execution of unpatched applications. We built a prototype, named P2, that leverages information about patched and unpatched applications to determine what applications are executed by VMs regardless of which operating system they are using. P2 can handle both binary applications and non-binary applications, such as the ones mentioned in the first paragraph (Eclipse, Tomcat, phpBB). After further motivating the need for P2 and outlining the shortcomings of existing approaches, we detail the results of a survey of security updates to Fedora Core. This survey confirms that monitoring non-binary files is crucial. We then describe how execution monitoring, file monitoring and a software database are combined to build P2.

In this chapter, we make the following contributions:

- **File Monitoring Prototype.** We have implemented a prototype that can track which files are read by processes executing in a monitored VM. In keeping with the principle of Architectural Introspection, the system only makes high level assumptions about the guest OS. It can successfully identify files read by Windows XP and Linux OSes and does not require modifying these OSes.

- **Patch study** We performed a study of patches on the Fedora Core 10 and 11 Linux distributions. We found that across both distributions, 102,819 files were updated, of which 23,711 are non-documentation files that may contain executable code or configuration resources. Among these files, only 36% are binary code files. This serves as a strong motivation for the ability to be able to detect the execution of unpatched non-binary code.
- **P2 architecture** We describe P2, which uses architectural introspection to detect the execution of unpatched binary and non-binary code. We demonstrate the OS-agnostic property of P2 by running it on both Windows XP and Linux VMs.

4.1 Assumptions

4.1.1 Assumptions about the Virtual disk architecture

As discussed in Section 2.2.3, devices emulated by QEMU [11] can support both Direct Memory Access (DMA) and Programmed I/O (PIO). For simplicity and because PIO is an obsolete method to access disk data, we focus on DMA. We ensure that PIO is not used by the monitored OS by disabling PIO support in the QEMU back-end.

The file monitor also assumes that the file system block size is the same as the memory page size (4096 bytes), which is true for all major file systems today. Current hard drives use a sector size of 512 bytes, which is the smallest addressable chunk of data that can be read from the disk. The memory page size for the x86 architecture is 4096 bytes. Since the sector size is smaller than the memory page size, it is possible that one memory page could contain multiple files or file fragments. However, no file system we know of uses this capability. Instead, file systems match their minimum addressable block size with that of the CPU memory system for simplicity and efficiency. In response to this trend, hard drive manufacturers have agreed on a new standard [18], which specifies that all disks will eventually use a sector size of 4096 bytes instead of 512 bytes.

4.1.2 Assumptions about the monitored OS

We make several assumptions about the OS that will be installed in the VMs. First, we assume that an efficient, commodity OS is installed. File monitoring relies on the efficiency assumption because it uses information flow tracking to infer whether a file is read by an application.

Specifically, we assume that when a process requests data from a file, data is copied directly from the disk into a buffer cache using DMA, and then copied from there into the address space of the requesting process. If the OS inefficiently makes extra copies of the data before returning it, this will confuse the information flow tracking performed by the file monitor. Similarly, DMA transfers are much faster than PIO transfers. OSes will generally use DMA if it is available and only fall back to PIO if DMA is not available. We have confirmed that this assumption holds for all flavors of Linux and Windows. Memory copying is expensive and we believe that all OS implementations that consider performance important will try to avoid unnecessary copying of disk data.

Second, this approach will fail if the content of the file is accessed by the kernel when a process that is not reading the file is scheduled. If this were to happen, the process that is currently scheduled would be incorrectly flagged as having accessed the file. However, the kernel should only access the content of a file when it is copying it to user space, at which point the process that is accessing the file should be scheduled. Our experimental evaluation discussed in Chapter 6 bears out this claim on both Windows XP and Linux.

Third, we note that the system can only be used to monitor files that are being read from storage through the virtual disk driver. If a file were to be downloaded via the network interface and then opened by a process, our system would not detect that the file is being opened by that process if the file has not been flushed out to disk yet. However, once the file is being flushed out to disk, accesses to the file can be monitored using the technique described above. To address this limitation, the file monitor assumes that a newly downloaded file may have been opened by any process that was seen executing for a period of time T prior to the file being written to disk. T should be greater than the disk cache flushing period used by the kernel.

Fourth, since the file monitor inspects disk content in the back-end, files that are stored encrypted or compressed on disk cannot be identified by the driver. File monitoring can be extended to handle encrypted or compressed files, but additional support is necessary to allow the driver to take hashes of the unencrypted and uncompressed data. Details of such an extension depend on the specifics of the encryption or compression system.

Finally, our file monitoring approach assumes that the system being monitored is not trying to covertly execute unpatched code. If the monitored VM is under the control of an attacker trying to hide their actions, the attacker can mislead the file monitor into believing that a file was opened by another process. For example, they could use `ptrace` to get process A to open a file and then again use `ptrace` to copy the content of the file to process B. As a result, the file monitor is only useful to monitor non-malicious VMs or malicious VMs that do not attempt to mislead the file monitor.

We summarize the assumptions made about the virtual disk architecture as well as the

Assumption	**Consequence if assumption does not hold**
Data transfers are performed using DMA.	Data transfers that attempt to use PIO will fail.
The file system block size is equal to the memory page size.	Both false negatives and false positives: accesses to files sharing a memory page will be conflated, resulting in false positives. A write to the page can result in false negatives.
The VM uses an efficient OS	Both false negatives and false positives, as tracking is no longer accurate.
If the VM maintains a buffer cache, accesses to the buffer cache are synchronous.	Both false negatives and false positives: accesses through the buffer cache will be attributed to the wrong process.
Monitored files are stored on persistent storage.	False negatives if a file is accessed but never flushed to disk.
The VM does not encrypt or compress data on its file system	False negatives: Hashes of blocks read from the disk will not match hashes in the database.
The VM is not malicious	Both false negatives and false positives, as the attacker can fool the monitor.

Table 4.1: Summary of assumptions about the monitored VM. Should an assumption be unwarranted for the monitored OS, the second column sums up the consequence of the mistaken assumption.

assumptions made about the OS executing in the monitored VM in Table 4.1. In Chapter 6, we demonstrate that these assumptions hold for both Windows XP and the Linux kernel.

4.2 System Architecture

The problem of determining what process accesses which files can be thought of as an information flow tracking problem, which is a well-known information security problem. However, the overhead incurred by software systems that perform byte-level information flow tracking is too large for deployment on production machines. For example, LIFT [70], an optimized information flow tracking system that uses Dynamic Binary Instrumentation, incurs an average overhead of 3.6 times the original execution time for SPEC INT2000 applications. Other systems achieve more reasonable overhead, but they require either hardware support [89] or recompilation of all the binaries executing on the system [16].

In contrast, we opt for an approach that can operate without modifications to the existing hardware or to the software executing in the monitored VM and does not result in significant overhead. We can achieve this goal because we do not require the full generality of information flow tracking. First, the granularity of our tracking is coarser than the one used by the systems mentioned above. We track information at the page granularity, as opposed to the memory word or byte granularity. Second, we assume that the information flow of file data in kernel space is simple: file content is either written in a buffer cache managed by the kernel or directly in a buffer in the process address space. If the file content is first written in a buffer cache, the file content is then copied to user space when the process is scheduled. This copy may occur multiple times, either to the same process or to distinct processes, but we assume that it is always synchronous: the copy to user space takes place when the process accessing the data is scheduled. Third, we do not track file content beyond the initial access.

The architecture of our file system monitoring framework is summarized in Figure 4.1, which shows the file monitor, the memory tracking component in the hypervisor and the execution monitor from Chapter 3. While we could have implemented file monitoring to support all three disk access modes discussed in Section 2.2.3, for simplicity we focus on our implementation for HVM domains that do not use paravirtualized drivers.

4.3 File monitoring implementation

The purpose of the file monitor is to detect when a file is read and report which file was read by which process. As a result, the most natural place to implement file monitoring is in the Domain 0 QEMU disk emulator. On each disk access, the file monitor computes a hash of

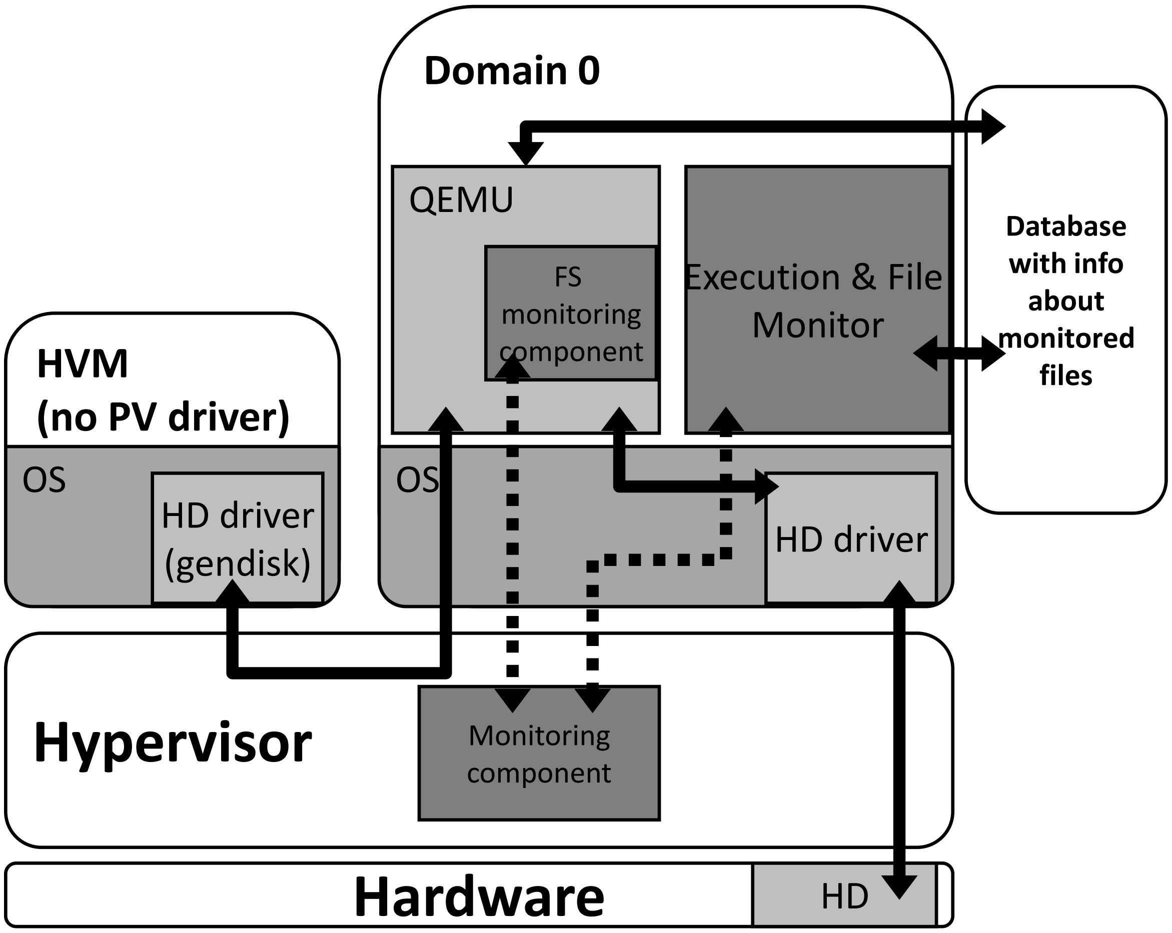


Figure 4.1: Our implementation of file monitoring consists of three components: a file system monitor process, modifications to the QEMU-emulated driver and modifications to the hypervisor.

the block being read. This hash is computed after the transfer to memory is complete. As a result, file monitoring does not introduce additional disk traffic. The file monitor then searches a database of files that should be monitored. This database is indexed by hash values computed over each block-sized chunk of the file. If there is a hit, the file monitor informs the hypervisor of the physical address that the disk data will be read to. The file monitor is able to get the destination address from the DMA request protocol, which includes this information. The file monitor also conveys the identity of the file to the hypervisor, which keeps track of the memory page to file mapping.

File system monitoring then leverages a memory tracking component in the hypervisor. When the memory tracking component is notified that a file was transferred to memory, it clears the "present" bit of the corresponding page table entries in the shadow page tables. As a result, any subsequent access to the page will result in a page fault. The hypervisor component also sets a flag, the *tracked* flag, in the per page-frame data that is kept by the Xen hypervisor. This flag was added so that when a fault occurs on that page, the Xen hypervisor knows to invoke the memory tracking component. In addition, if the guest OS creates new mappings of a tracked page, the Xen hypervisor will also clear the "present" bit on the new mappings in the shadow page table.

If and when a fault to a tracked page occurs, the memory tracking component is invoked by Xen. This component will inspect the instruction that caused the trap. The instruction can be either in the guest kernel or in a user space process in the guest VM. If the instruction is in the guest kernel, then the file data is being accessed by a process via a system call such as `read`. In this case, the memory tracking component inspects the instruction and verifies that the instruction is copying data from the tracked page into the user space process. If the trapping instruction is in a user-space process, then the file data has been mapped into the address space of the user space process. Thus, the file monitor can infer that the file data has flowed into the address space of a particular process. In either case, the memory tracking component has inferred that a file has been read into a user space process. It then signals the execution monitoring process with the `CR3` of the accessing process and pauses the monitored VM until it gets a response.

The content of the `CR3` is also used by Patagonix to identify processes, so the data can now be combined to determine which application accessed the file. The execution monitoring process then logs or reports the access. It can also prevent the file access from taking place. This decision takes place before the accessing process can access the data. If the access is permitted, the hypervisor sets the "present" bit in the page table entry and resumes execution of the monitored VM.

The hypervisor does not clear the "present" bit immediately after the access has taken place. Instead, it waits until the content of the `CR3` is changed by the monitored OS, indicating a context switch, to clear the bit. At this point, the memory tracking component must clear the present bit so that it can detect accesses by other processes. Since multiple traps may take place between two context switches, the tracking component maintains a list of page table entries that need to have their "present" bit cleared upon the next context switch. In this way, the number of traps caused by the monitoring is minimized.

In addition to detecting when a page containing file data is being accessed, the monitor needs to detect when the memory page no longer contains the file data. This may occur as a result of two events: the monitored VM either modified the content of the memory page itself, or it requested that a virtual device performs a DMA transfer to the memory page. To detect the first scenario, the monitor clears the "writable" bit in the page table entry of any monitored page. As a result, when the OS writes to the page, either because the page is being reused by the kernel for a different purpose or because the page content is being changed by the application, a page fault will occur upon access. At this point, the page fault handling code detects that the tracked flag is set. It then clears this flag so that any subsequent page fault caused by accesses to this page simply results in the shadow page tables being updated and new mappings of this physical page can be created without the "present" bit being cleared.

To detect DMA transfers, we modified the virtual devices so that they inform the hypervisor of all DMA transfers, not just DMA transfers that involve files that are tracked. If an input/output memory management unit (IOMMU) is available, it could also be used to detect when devices are writing data to monitored pages. In both cases, the monitor stops tracking accesses to the page after it has been written to.

Two cases deserve special attention in the monitor: files may be smaller than a single 4096 block and some blocks appear in more than one file. To be able to handle small files, the database stores a hash of a sub-block prefix of each block of the file along with the amount of space occupied by the file in this block in a *block prefix index*. The length of this prefix needs to be chosen when the database of hashes is constructed. The length of the prefix is chosen to minimize the chances of a collision while retaining the ability to handle even very small files. Files that are smaller than the chosen prefix length cannot be detected by the system. For each block access, the file monitor first computes a non-cryptographic hash of this prefix and searches the block prefix index for a match. If there is a hit in the block prefix index, either a hash of the remainder of the block is taken if the file occupies the entire block, or a hash of the portion of the block that is occupied by the file is taken if the file does not occupy the entire block. This serves to verify that the file is indeed a match. If multiple files exist for a given prefix, then each candidate needs to be tested individually until a match is found. Use of the prefix index means that for blocks read from the disk that do not contain data from files that should be tracked, only a short prefix of the block needs to be hashed before the monitor can establish that the block does not contain data that needs to be tracked.

To handle cases where a file block appears in multiple files, the file monitor conveys the set of files that could have matched. While rare, we have observed that different files do occasionally have exactly matching 4096 byte chunks.

4.4 Patch auditing in Infrastructure as a Service Clouds

A large number of security vulnerabilities stem directly from software implementation flaws in critical code. To maintain the security of a system against attackers, it is critical that patches, which fix these flaws, be applied in a timely manner. As a result, many software packages and operating systems (OSes) contain support for automatic patch installation. These systems are simple – they periodically check a central server for the existence of patches and apply any patches that have not yet been applied to the system they are running on.

However, automatic patch installation cannot always prevent the exploitation of known vulnerabilities and can reduce the stability of computer systems. Patches are not applied instantly as the patch installation system only checks for updates periodically. This creates an exploitable window of vulnerability between the time the patch is disclosed and the time it is applied [14]. In addition, automatic patch systems are unaware of which patches need to be applied, and proactively apply all available patches, even if the patched component is never used. Unfortunately, patches can have unintended side effects, so such unnecessary patches can needlessly cause system failures or performance degradation [9].

This problem is made more acute in a virtualized cloud environment. Public and private virtualized cloud environments offering Infrastructure as a Service (IaaS) have grown in popularity due to their ability to provide elasticity of resources and reduce user costs. These environments have two characteristics that make patch management even more complicated than in an unvirtualized environment. First, virtualization introduces capabilities that break standard automatic patch installation systems [28]. Patch installation systems rely on machines always being powered on so that they can check for updates and apply them. They also assume that time proceeds in a linear fashion so that patches are naturally applied in sequence. Unfortunately, virtual machines (VMs) can be archived and left powered off for long periods of time. They can be rolled back to a previous checkpoint in time, cloned, migrated, created and destroyed easily and frequently. These capabilities skew time in the VM and can cause automatic patch installation systems to fail.

Second, by separating the administration and maintenance of hardware from that of the software, IaaS clouds will permit a larger number of administrative domains, resulting in a greater diversity of OSes and software environments. Whereas a single organization with a single IT department may have forced all users to conform to a homogeneous computing environment, a private cloud environment removes that restriction, giving users more freedom. Users may have any OS (i.e. Windows, Linux), any flavor of the OS (i.e. Vista, XP, Ubuntu, Red Hat), and any version level (i.e. Linux kernel version, Windows service pack). In a public cloud, such as those implemented by Amazon's EC2, GoGrid and Mosso, there are no restrictions at all on what OSes and software users may install. The de-federalization of administrative control and greater diversity in software make it difficult for a cloud provider to ascertain the patch level of VMs running on their infrastructure. Unfortunately, the insecurity of a single cloud user can negatively impact both their fellow cloud users and the cloud provider itself [71]. Thus, cloud providers are motivated to identify and protect themselves from VMs on their cloud that are vulnerable to attack.

In this section, we demonstrate that virtualization can also solve the patch management problems it creates in cloud environments. We design and implement *P2*, a patch audit solution that leverages the hypervisor to detect and mitigate vulnerabilities in unpatched software in VMs. P2 has several advantages over existing solutions. First, P2 provides *continuous* protection, and does not fail if the machine is powered off or check-pointed and rolled back. Second, P2 is more *accurate* than existing solutions. P2 only reports unpatched software if it is actually executed, reducing the number of alerts and enabling administrators to apply only the minimal number of patches required. Finally, P2 is *OS-agnostic*, allowing it to work on any standard commodity OS. This allows the cloud provider to have a single patch audit solution, instead of having to support an OS-specific one for every OS their customers use.

P2 accomplishes this by relying on architectural introspection. By restricting monitoring to only hardware state, P2 is able to detect unpatched software in VMs without having to rely on any detailed or implementation-specific information about the OS and software in the VM. P2 detects unpatched binary and non-binary software. Binary software denotes software that will execute natively on the underlying processor, such as executables or libraries. Non-binary software generally refers to scripts or byte-code, which will execute with the aid of an interpreter or just-in-time compiler (JIT). However, non-binary software can also include configuration files and other application resources.

To detect the execution of an unpatched binary, P2 monitors memory using execution monitoring. To detect scripts and byte code that are executed by an interpreter or JIT compiler, P2 uses file monitoring. This lets P2 determine if an unpatched file is read from disk into the address space of an interpreter or JIT compiler that is capable of executing the non-binary file. In this way, P2 can detect the execution of unpatched non-binary code with low overhead and few false positives.

When P2 detects unpatched code it can take one of two actions depending on the mode it is used in. In *reporting mode*, it simply reports that a VM has executed unpatched code to the cloud administrator, who can then inform the VM administrator and take actions according to their service agreement (i.e. the VM administrator may have to patch the code or the cloud administrator could adjust firewall rules to prevent exploitation). In *prevention mode*, in addition to reporting the unpatched code, P2 actively prevents the exploitation of the unpatched code by injecting code into the vulnerable process that prevents it from executing any more instructions.

4.4.1 Overview

Motivation

A survey conducted by Bellissimo et al. in 2006 [12] found that 69 out of 71 CERT Technical Cyber Security Alerts recommended applying updates or patches to fix vulnerabilities. This supports the conventional wisdom that many security attacks can be prevented by keeping a system patched. Software developers have come to the same conclusion and recent software systems now commonly incorporate automatic software update components. In this section, we motivate P2 by summarizing the existing state of the art in patch update systems and illustrating their deficiencies compared to P2 in terms of how accurate, continuous and OS-agnostic they are.

Automatic update systems can be provided by the OS, or built specifically into an application. OS-wide update systems, such as the ones built into Windows, Mac OS and Linux periodically check for patches and can be configured to automatically download and apply them. OS-wide automatic update systems may also be built into system administration tool suites, such as IBM's Tivoli system, which also performs various other security and compliance auditing tasks. Application-specific automatic update systems only handle patches for a specific application. Many popular applications, such as the Firefox and Chrome browsers, Acrobat Reader, and many games implement such solutions. Application-specific update systems usually execute when the application is run, when they will check for patches and if necessary, download and install them.

All automatic update systems are host-based: they execute as part of the software system, either as separate software agents or as software mechanisms built into applications. As a result, these systems are tied to a specific OS and are thus not OS-agnostic. In addition, the cloud provider must rely on the cloud user to properly install and configure host-based systems since the cloud provider does not have administrative access to the VMs. Automatic update systems can be accurate or continuous depending on whether they are OS-wide or application-specific.

OS-wide updaters periodically compare a database of installed patches on the host with an online repository of patches to determine if patches need to be applied. They are unaware of whether the components they maintain are executed or not, making them inaccurate. In addition, because the check for patches is periodic, they are not continuous. Application-specific updaters that exist as separate applications from the application they maintain have the same properties as OS-wide updaters. However, many application-specific updaters are part of the application they maintain and only run when the application they are patching is run, making them accurate as a result. However, they are only partially continuous. These systems will check for patches when the application first starts up, and then periodically afterwards. Thus, if a patch exists at start-up, they will apply it before the application actually runs. However, if a patch becomes available while the application is running, it will not be detected and applied until the next time the update system checks.

An alternative, OS-agnostic approach for detecting the presence of unpatched software is a network vulnerability scanner such as Nessus [56]. These scanners attempt to detect unpatched code by scanning all hosts and ports in a range of IP addresses and trying to identify the version of any network facing services installed on those hosts. The identification of the software version can be done simplistically by examining the banner returned by the service, or in a more sophisticated way by sending requests to the hosts and observing the responses. After such scans, the network vulnerability scanner informs system administrators of the presence of software with known vulnerabilities on machines connected to the network. A network vulnerability scanner does not rely on any software agent being present on the host and thus is attractive in IaaS clouds where there is a diversity of OSes running and the cloud administrator has no administrative access to install and manage any host-based solution.

Unfortunately, a network-based approach has limited coverage. It can only be used to detect unpatched server software for which the version can be determined via network queries. As a result, it will not protect a user that uses an out-of-date browser to visit a malicious website. In addition, network scanning has limited precision and may not be able to definitively determine if a service is vulnerable or not. For example, if the network scanner relies on banners, it will fail if the server is configured not to return a banner. More sophisticated scanners that attempt exploits may fail if the exploit's success depends on random factors. Thus, network scanners have limited accuracy. Network scanning is also non-continuous since the scans only happen periodically. In a cloud environment, this problem is compounded by the fact that software executing on VMs that are only powered on occasionally could remain out-of-date for long periods of time.

In this work, we show how P2 takes advantage of a virtualized infrastructure to implement a patch audit system that has the combined advantages of existing systems. Table 4.2 summarizes

Solution	**Continuous**	**Accurate**	**OS-agnostic**
OS-wide update	No	No	No
Application update	On start-up	Yes	No
Network scanning	No	No	Yes
P2	**Yes**	**Yes**	**Yes**

Table 4.2: Comparison of P2 and current solutions.

the advantages of P2 over host-based automatic update systems and network-based vulnerability scanning. In a virtualized environment, a ubiquitous layer of software executes below OSes which now interact with virtualized hardware. Because it facilitates management and allows system administrators to retain some control over desktop machines, such a setup has become popular and is offered by VMware ACE [97] and Citrix XenClient [19], amongst others. This is also increasingly the architecture used in clouds, be they public or private.

Assumptions and Guarantees

Because P2 uses file monitoring, it relies on the same assumptions as the ones listed in Section 4.1.2. The first three assumptions are realistic in a cloud environment: VMs will use an efficient OS and are usually self-contained, meaning that all the code they execute originates from persistent storage. The fourth assumption implies that the cloud provider can only offer patch monitoring to customers that do not use disk encryption. Finally, the information provided by the system will only be reliable as long as the monitored VMs are not compromised. Since the goal of P2 is to prevent compromises in the first place, applying it to malicious VMs would be pointless.

Given that these assumptions hold, P2 provides two guarantees. First, P2 will identify the execution of all unpatched applications, regardless of whether they are composed of native binary code, non-binary code, or some combination of the above. Non-binary code includes all forms of byte code or scripts, regardless of whether they are executed by an interpreter or a JIT compiler. P2 infers non-binary code execution anytime a script or byte code file is loaded into the memory of a matching interpreter or JIT compiler, i.e. if the Perl interpreter loads a valid Perl script or a JAVA virtual machine loads a JAVA class file. P2 can also detect unpatched resource and configuration files, provided that they match the unpatched, vulnerable version exactly – P2 does not identify vulnerabilities caused by custom configurations. Second, P2 works for any commodity OS. We have validated P2 on two widely used OS environments, Windows XP and Fedora Core Linux.

OS	Total changed files	Documents	Binaries	Non-binaries
FC 10	73800	61037	4979	7784
FC 11	29019	24331	1307	3281

Table 4.3: Summary of Fedora Core security patch RPMs.

4.4.2 Patch Survey

To be OS-agnostic, P2 uses architectural introspection, which restricts hypervisor monitoring to the interaction between the guest VM and virtual hardware. As a result, P2 must make a distinction between applications implemented in native binary code and applications implemented in an interpreted language. On the one hand, binary code must execute directly on the processor and thus is observable through interactions with the memory management unit (MMU). On the other hand, interpreted code can only execute when it is read and executed by the appropriate interpreter or JIT, which itself is usually a native binary. To understand the importance of these two forms of monitoring, we conducted a survey of historical data to characterize the composition of security-sensitive patches.

We collected all security patches for Fedora Core 10 and 11 from the Fedora Project's admin website [1]. The Fedora Core 10 updates consist of security patches between its release date on November 25, 2008 until October 1, 2009 when we conducted the study. The Fedora 11 updates were for the same period starting from its release date on June 6, 2009. To identify which patches contain security fixes, we used the Fedora Project's update classification scheme, which classifies patches as pending, testing, stable or security. To be classified as security, the patch must contain at least one security critical fix.

To characterize the types of files patched, we compared each security patch with the previous version of the application that the patch replaced. Fedora uses Red Hat's RPM package format to distribute patches. RPMs contain entire binaries, which will overwrite the binaries of the previous version when the patch is installed. In addition, patch RPMs can also be installed onto a system where no previous version of the application exists. Thus, patch RPMs also contain files that did not change from the previous version. Thus, to identify which files are patched, we compare the files in each patch RPM with the RPM of the previous version. From this, we can characterize the makeup of files that changed in response to a security vulnerability. We note that while a security patch must fix at least one security vulnerability, the patch may also contain fixes for non-security sensitive bugs. Unfortunately, neither the the Fedora Project page, nor the RPM package format contain sufficient information to remove files that contain

[1] `https://admin.fedoraproject.org/updates/`

Extension	**Description**	**Modified files**
pyo	byte-compiled Python (optimized) code	1324
pyc	byte-compiled Python code	1305
php	PHP scripts	1231
js	JavaScript files	519
tmpl	Bugzilla template files (include templates for scripts)	293
desktop	Window manager configuration files	278
elc	byte-compiled Emacs Lisp code	276
info	Drupal configuration files	265
inc	PHP scripts	246
jar	Java bytecode archives	238

Table 4.4: 10 most common non-binary file types by extension appearing in security updates for Fedora Core 10.

non-security sensitive bug fixes from our study, so we conservatively assume that any file that changed due to a security patch contains a security fix.

Table 4.3 summarizes the data obtained from this survey. Tables 4.4 and 4.5 detail the 10 most common non-binary file types by extension for Fedora Core 10 and 11, respectively. "none" denotes miscellaneous files that had no extension. We analyzed 446 out of the 495 security patches in Fedora Core 10 and 130 out of the 134 security patches in Fedora Core 11. Not all patches could be analyzed because we could no longer obtain the matching previous RPM packages in some cases. For each file in an RPM that was changed due to a security patch, we determined if it was a binary file or a non-binary file using the `file` utility, which classifies any ELF file as a shared object or as an executable, and any other file as a non-binary file. To get a fair assessment of the percentage of non-binary files that were modified by security updates, we filtered out `man` pages, document files stored in `/usr/share/doc`, `/usr/share/info`,... as well as `locale` files that contained localization data. We further classified the remaining non-binary files based on their extension.

The results illustrate two important findings. First, 64% of the updates across both distributions were to non-binary files. This indicates that P2's ability to monitor the execution of non-binary code in an OS-agnostic way is critical to its success as a patch audit solution. Second, a significant portion of the top 10 non-binary files consist of executable scripts and byte code: `pyo` and `pyc` are compiled python code, `php` and `inc` contain php code, which is often used to implement websites, `jar` and `js` are for JAVA and JavaScript respectively, and

Extension	**Description**	**Modified files**
php	PHP scripts	676
pyo	byte-compiled Python (optimized) code	578
pyc	byte-compiled Python code	522
jar	Java bytecode archives	390
js	JavaScript files	203
inc	PHP scripts	109
zip	ZIP archives	80
gif	GIF bitmap images	78
none	Miscellaneous	73
py	Python script	61

Table 4.5: 10 most common non-binary file types by extension appearing in security updates for Fedora Core 11.

`elc` files contain Emacs Lisp code.

The top 10 non-binary file types make up 77% and 84% of all non-binary, non-document files in Fedora Core 10 and 11 respectively, forming a significant portion of the patched non-binaries. Thus, many of the non-binary updates are to executables that are run with the aid of an interpreter or a JIT compiler. In particular, much of this code appears to be used to implement web applications. While P2 also has the ability to identify unpatched configuration or application resource files, patches to these types of files are not common in practice.

4.4.3 System Architecture

We now describe the design and architecture of P2, which provides accurate, continuous and OS-agnostic detection of unpatched code in a VM. P2 is most naturally implemented in the virtualization infrastructure that manages the cloud VMs or a company's VMs. As a result, its operation is administered by the cloud provider in the public cloud setting, or by the system administrators in charge of the computing infrastructure in an enterprise private cloud setting. The architecture of P2 is illustrated in Figure 4.2. While P2 leverages Patagonix as a component, all other components in the figure are new contributions of P2. We begin by describing the database of software information that P2 needs and then describe how P2 uses these databases to detect the execution of unpatched binary and non-binary software.

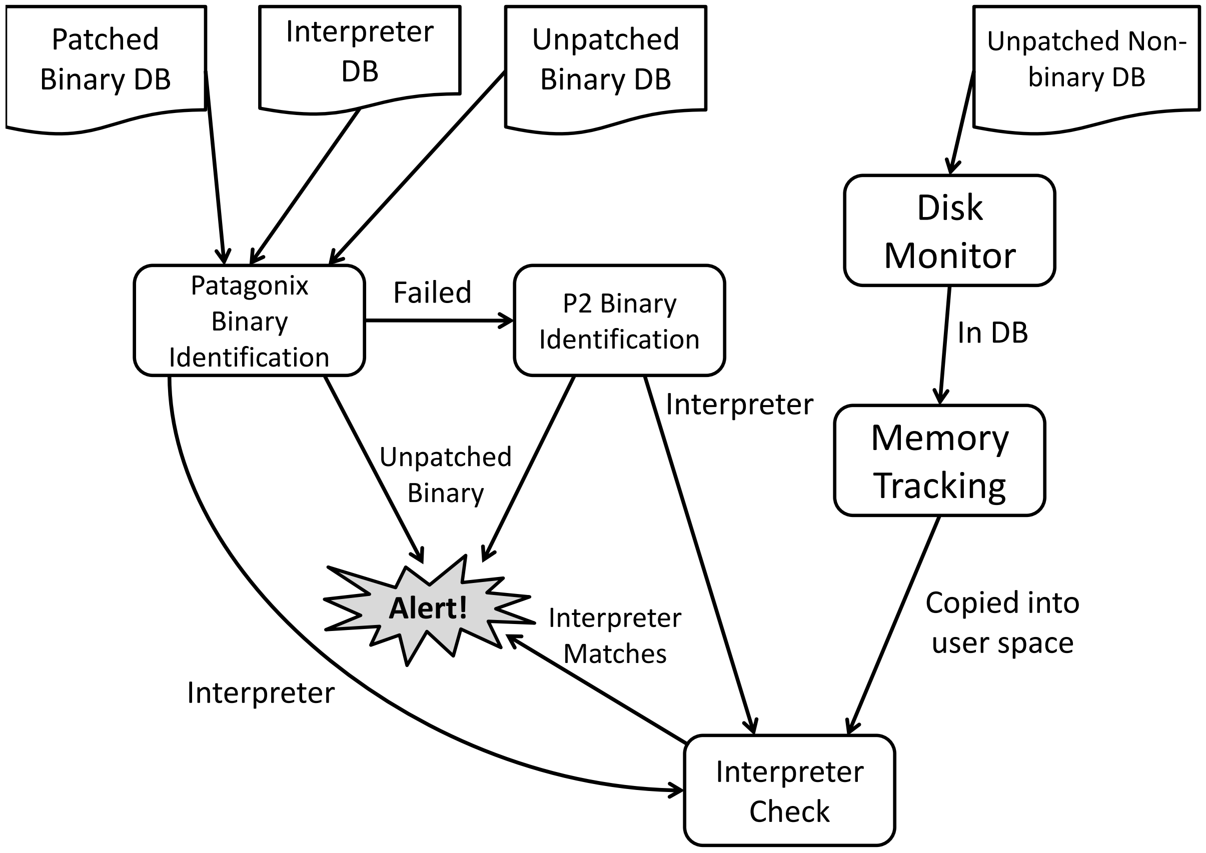


Figure 4.2: P2 architecture.

Software databases

To detect and report the execution of vulnerable, unpatched applications, P2 requires a database of information to identify these applications. Such an *unpatched database* can be derived from the files that make up the application. A list of files that need to be patched can be easily obtained by monitoring the updates applied by automatic update systems, or manually monitoring security vulnerability distribution lists. P2 requires the files that each patch modifies, which can be obtained by comparing the files in the patch with the software version it was patching. The patch survey described in Section 4.4.2 was performed using scripts written over a course of 6 days by one of the authors of this paper. Based on this experience, we believe that building and maintaining a database of unpatched software would require only modest effort and time. P2 also requires an *interpreter database*, which lists applications that may load and interpret unpatched non-binary files. We use the term *interpreter* to denote all interpreters and JIT compilers on the system, as well as any application that can load unpatched configuration files. Each non-binary file in the unpatched database must be associated with at least one binary in the interpreter database, which can load the non-binary file and exercise the vulnerability.

P2 also requires a *patched database* of files that do not need to be patched. Such a database would be considerably larger than the database of unpatched files since there are many more of such files. Maintaining such a database in a homogeneous enterprise setting where a single entity controls the software environment represents a tractable undertaking. In a more open setting, building a list of patched applications is a more difficult task, but not intractable. For example, anti-virus vendors already maintain large databases that keep track of information about malicious software. Some anti-virus vendors have suggested that it may actually be easier to maintain a white list of software instead of the current industry practice of maintaining a black list of malicious software [59]. The administrator can also leverage existing databases of software that currently exist. For example, NIST maintains the National Software Reference Library (NSRL), a database of hashes for a large number of applications [58] and VersionTracker [95] keeps track of available software updates. Linux vendors also maintain large repositories of software and track updates to this software.

In a public cloud setting, the cost of the work required from the cloud provider can be spread amongst cloud customers. The task of maintaining the database can also be outsourced to third parties. The exact details of how such databases would be managed are outside the scope of this work. For the rest of this paper, we assume that the cloud provider has access to the files that correspond to the software that need to be monitored.

Monitoring

The database of unpatched files is divided into two databases: one containing binary files and the other containing non-binary files. The database of unpatched binary files is then combined with a database of interpreters and the database of patched binaries. This combined database is then used for identifying executing binaries.

When P2 detects an executing binary, it first invokes Patagonix to identify the binary. If Patagonix is able to identify the binary it takes actions depending on which database the binary is identified in. If the binary is unpatched, P2 raises an alert and, if running in reporting mode, reports the alert to the cloud administrator. If P2 is running in prevention mode, it will also prevent the unpatched code from running using the technique described in Section 4.4.3. If P2 matches the executing binary with an entry in the database of interpreters, then it notes the address space the interpreter occupies for use in detecting unpatched non-binary file use.

P2's mechanism for identifying unpatched non-binary files combines file monitoring and execution monitoring. First, P2 uses file monitoring to track any unpatched non-binary file that the monitored VM accesses. When an access to such a file is detected, P2 leverages information from the binary execution monitoring – if it detects that data from the unpatched non-binary file flows into the memory space of a matching interpreter for the file, then it raises an alert and takes action depending on its operating mode.

For file monitoring, P2 uses a prefix length of 64 bytes. As a result, only files smaller than 64 bytes cannot be handled. In our patch study, there were 32 files out of a combined 17351 files across both Fedora Core 10 and 11 that were fewer than 64 bytes in length. Due to their short length and simplicity, we also believe they are unlikely to contain security vulnerabilities.

It is crucial that P2 detects that an unpatched non-binary file is actually loaded into the appropriate interpreter before raising an alert. For example, consider a simpler system where P2 does not track data flow through memory, but instead raises an alert whenever an unpatched file was read off disk. Such a naïve approach will raise alerts even when the file is accessed in a way that will not exercise the vulnerability in the unpatched file. Execution of a file by an interpreter or as a configuration file constitutes *vulnerable accesses*, as it has the potential to exercise the vulnerability in the file. On the other hand, access by applications such as anti-virus software performing a scan, desktop search software that is indexing the system, or backup applications constitute *non-vulnerable accesses* as the accessing applications do not interpret the file data in a way that can exercise the unpatched vulnerability. Without memory information flow tracking, P2 will lose accuracy and report non-vulnerable accesses as alerts. Unpatched files may be present on a system and accessed by such applications for a variety of reasons. For example, Windows keeps copy of old libraries after an update to allow software rollback should the update fail or result in unforeseen problems.

Prevention mode

When running in prevention mode, P2 is in a position to prevent any unpatched code from executing. P2 does this with as few side-effects as possible. If the unpatched code is a binary, then P2 replaces the instruction at the faulting address with an illegal instruction. When the guest VM is resumed, the application will execute the illegal instruction causing an OS fault, and the guest OS will terminate the application cleanly.

If the unpatched code is a non-binary file, P2 must terminate the associated interpreter that is making a vulnerable access to the file. If the access occurred from user space because the file is mapped into the interpreter address space, P2 inserts an illegal instruction at the address that caused the trap and restarts the process just as above. However, if the accessed occurred from within the kernel, P2 cannot insert illegal instructions into the kernel as this would terminate the entire VM. Instead, P2 allows the access to complete but sets the entire user space address range to non-executable. When the guest OS returns to user space, a fault will be generated, at which time P2 can inject an illegal instruction and have the guest OS terminate the interpreter.

Chapter 5

Attestation Contracts

In distributed applications, a party would often like to have some guarantees about the integrity of the other party before sharing information or resources with them. For example, a bank may wish to ensure that a user's machine is free of spyware before divulging sensitive information about the user's account. Similarly, the provider of a multi-player online game would like to have some assurance that players have not tampered with their clients in such a way as to attain an unfair advantage. In each case, the ability for the user to make some guarantees about the integrity of their machine will help the remote party determine whether it should continue the dialog with the user's machine.

Currently, a user may try to make such guarantees via the Trusted Computer Group's Trusted Platform Module (TPM) [91], which is a tamper-resistant coprocessor that is soldered onto the motherboard of many commodity machines. The TPM measures the integrity of a system by computing cryptographic hashes of executable code and data on the system. The user can then ask the TPM to vouch for the integrity of the measurements by having the TPM sign these hashes, and then sending the signed hashes to the remote party – a process that is called *attestation*. The remote party verifies the authenticity of the hashes by verifying the TPM's signature, and then determines, based on the measurements, whether or not the user's system meets a level of integrity that is satisfactory to the remote party. Although the TPM can be used to certify the measurements of both code and data, current attestation methods generally focus on measuring the integrity of the binary code running on the system. This is because the effect that low-integrity data has on the overall integrity of a system is difficult to determine and is largely dependent on the semantics of the software handling the data. As a result, the task of determining whether the programs that are running on the user's system can be compromised by low-integrity data is left up to the remote party.

Current attestation methods have several shortcomings. First, the guarantees they provide are only valid at the time of attestation. The TPM's measurements reveal nothing about any software that is loaded after the attestation has taken place, thus weakening the guarantees provided to the remote party. Second, these methods record and report all software that has run since the machine was started by the user. This makes current attestation methods inflexible, since the user is forced to reboot her system if she has run any software that the remote party does not find acceptable, even if that software is no longer running. Recently, both Intel and AMD added a mechanism called late launch [32] to their CPUs. Late launch removes the need to reboot, but it is still disruptive because it must suspend the system while it dynamically loads a hypervisor beneath the operating system. Other systems that have used late launch have found it to be challenging to use, both because it requires careful interaction with the OS to avoid corrupting device state and because it is slow [51, 52].

The static property of current attestation methods also infringes on the user's privacy by reporting all running software to the remote party. In most cases, the remote party only needs to know that the user is running a certain class of software, or is not running another class that is undesirable [76]. For example, the remote party may wish to ensure that a user is running a fully patched version of Windows, but does not care exactly what flavor of Windows the user is running. As a result, current attestation methods report more information to the remote party than is actually required.

To address these problems, we propose an attestation mechanism that is both continuous and dynamic. Rather than passively certifying the authenticity of the measurements, our mechanism relies on a *Trusted Intermediary* on the user's machine to actively and continuously enforce the integrity requirements of the remote party. The Trusted Intermediary's enforcement can be activated by the user when she connects to the remote party and then deactivated when their interaction has terminated.

This Trusted Intermediary usage model can be formalized as an *attestation contract*. Attestation contracts are motivated by the observation that in many scenarios where attestation is beneficial, the remote party typically grants the user access to some service in exchange for the user proving the integrity of their machine to the remote party. Before service is granted, the remote party and the user negotiate an attestation contract, which states that for a certain period of time, the remote party will grant the user access to a service, and that the user will only execute software within a restricted set chosen by the remote party, as shown in Figure 5.1. Because the set of permitted software is specified by the remote party, the user need not report what software she has run in the past on her machine, nor reveal which programs within the remote party's specified set she is running.

One concern with this approach is that the remote party may abuse the Intermediary to mount a denial of service attack on the user by having it enforce unsatisfiable requirements. This concern was one of the primary motivations of the Trusted Computing Group to use

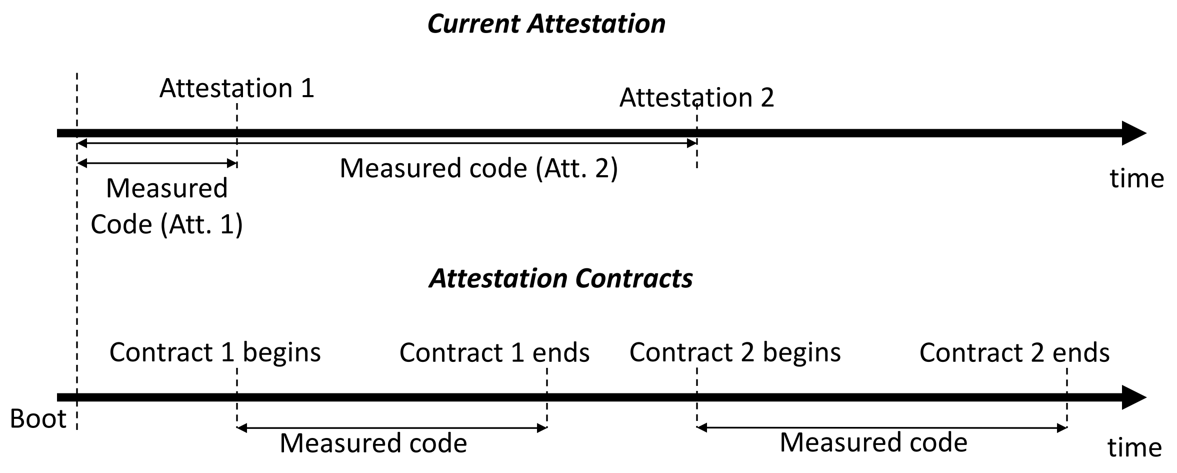


Figure 5.1: Time line illustrating the difference between current attestation methods and attestation contracts: Contracts measure and restrict code executed for a time interval while current attestation measures all code loaded between boot time and the time of attestation.

trusted boot, which only reports what OS is running, as opposed to secure boot, which limits what OSes may boot. To prevent denial of service attacks, the Trusted Intermediary allows the user to disable a contract whenever she wishes, but ensures that the remote party is always notified of the termination of the contract.

We have implemented the Trusted Intermediary as a combination of a hypervisor and a trusted VM. The hypervisor can be measured and certified by the TPM and the hypervisor can in turn measure and certify the trusted VM. The Trusted Intermediary uses the execution monitoring techniques described in Chapter 3 to control what software binaries can be executed on the system. Unlike an operating system, hypervisors generally do not require run-time loading of new code. Thus, basic load-time attestation is sufficient to guarantee that the hypervisor has not been tampered with. In addition, because the hypervisor is located below the operating system kernel in the software stack, the hypervisor is able to enforce an attestation contract that includes the operating system (OS) kernel. Execution monitoring also ensures that code that has been verified cannot be subsequently tampered with.

In this chapter, we make three main contributions:

- **Attestation Contracts.** We introduce the concept of attestation contracts, and detail the design of an active Trusted Intermediary that enforces them. We also discuss contract *refinements*, which can be used to provide more flexibility and power to attestation contracts.

- **Usage model.** We show that attestation contracts do not impact the availability of the user's system by showing how either party in the contract can terminate the contract at any time, but not without notifying the other party first.

- **Prototype.** We demonstrate the practicality of attestation contracts by describing a prototype implementation. We also evaluate the impact on the Trusted Computing Base (TCB) of attestation contracts.

We do not claim that attestation contracts solve all the difficulties with using attestation. For instance, attestation contracts, just like current attestation methods, require the remote party to perform the difficult tasks of determining what set of software may run on a system without compromising its integrity, as well as determining what input data can safely be used on the system. On the other hand, we believe that attestation contracts are a viable solution for making attestation continuous and dynamic.

5.1 Attestation background

Current attestation methods rely on the TPM to report the state of the user's machine in a trustworthy manner. The TPM accomplishes this by taking and storing measurements of the machine state in a set of Platform Configuration Registers (PCRs) that are isolated in the TPM's tamper-resistant packaging. These measurements are performed by computing hashes over code that is executed during system start-up, such as the boot loader and the OS kernel – a process referred to as *trusted boot*. When the remote party challenges the user to prove that her machine is running a certain operating system, she can request the TPM to sign the values in the PCR registers with an Attestation Identity Key (AIK) that is only available to the TPM. Because the remote party trusts the TPM, this allows the remote party to verify that the user is indeed running the OS she claims she is.

While this utilizes the TPM to attest to the state of the OS kernel, a good description of a mechanism that extends the current attestation to the state of all software running on a system is provided by Reiner et al [79]. They utilize the TPM to implement an *Integrity Measurement Architecture* (IMA), which uses a trusted OS kernel to take measurements of application binaries that it runs. To do this, the OS is modified to measure all application code and kernel modules as they are loaded and to keep these measurements in the PCRs. These additional measurements are added to the PCR values after boot by using the operation: $TPM\_extend(PCR, value) : PCR_{new} = SHA1(PCR_{old}||value)$. If some malicious software has been loaded, it can then load other software and bypass the measurement step, but it cannot set the PCR back to the old value before the first piece of malicious software was loaded. The remote party will thus be aware that the malicious software has executed at some point when the user attests their machine.

Modifying the OS kernel to make load-time measurements only enables the IMA to make attestations to code directly loaded by the kernel, such as code in application binaries and dynamically linked libraries. *Structured data*, which the authors define as data that has integrity semantics for the application, is more problematic for the IMA. Examples of structured data include configuration files and scripts. To attest to the integrity of such data, the applications that use the data must be modified to take measurements of the data and pass the measurements to the kernel. For example, Reiner et al. modified the `bash` interpreter to take measurements of all shell scripts that are executed.

In general, attestation contracts are able to provide the same guarantees to both the user and the remote party that IMA provides, while at the same time improving upon current methods by making the attestation both continuous and dynamic. As a result, contracts can provide integrity guarantees continuously, allow dynamic installation and removal of contracts without reboots, and protect the user's privacy by revealing less information about what software the user is running.

5.2 Attack Model

We have introduced the advantages that attestation contracts have over current load-time attestation described in Section 5.1. To make these advantages concrete, consider the following three scenarios that illustrate the need for attestation contracts.

5.2.1 Delayed tampering

A user playing an online multi-player game tries to cheat by tampering with her client. The online game server requires the user to attest to the integrity of her machine before connecting to the server. To cheat despite the attestation check by the server, the user starts the legitimate client and attests to the integrity of the client to the server. After the attestation, she replaces the client with the tampered one. The server is completely unaware of the switch as both the legitimate and tampered client have exactly the same network behavior. By using an attestation contract, the server can ensure that the user's client remains authentic throughout the gaming session.

5.2.2 Connecting to a corporate network

To protect against infection of its wireless network by machines harboring malware, a corporation forces all guests who access the network with their own machines to attest that they are only running a limited set of applications such as a web browser, e-mail client, and anti-virus scanner. A consultant who visits the corporation frequently connects to the network with his laptop to check e-mail, but finds it annoying to have to reboot his machine every time. Attestation contracts can be enabled and disabled dynamically without the need to reboot, allowing the consultant to simply enable a contract that restricts his machine to the appropriate software when he connects, and then disable the contract when he disconnects, allowing him to regain full use of his machine.

5.2.3 Application privacy

Online poker sites frequently require their customers to download clients so that they can detect and prevent collusion among players. One particular customer uses his home machine to play poker, but is concerned about the effect his hobby, if discovered, will have on his reputation at work. Unfortunately, when he connects to his work from home to read e-mail, his machine attests to the mail server and reports all the applications he has run, including the poker client. An attestation contract would bind the user to only using software that the mail server approves of. Even if the mail server approves of the poker client, the server does not know whether or not the user has actually run the poker client.

5.2.4 Summary

In each of these cases, current attestation methods would not have been adequate to provide the functionality or guarantees required by the scenario.

The above scenarios illustrate that there are at least three possible types of adversaries that attestation contracts have to deal with. In the first scenario, the adversary is the user, who has administrative control of the machine, and may even have physical access to the machine. In general, an adversarial user's goals are to execute software on her machine without notifying the remote party. Her abilities enable her to make arbitrary changes to the software used by the machine, such as applications or the OS kernel. However, an adversary who tampers with the TPM chip is outside of our attack model, since the raison d'être of the TPM is to provide a hardware root of trust that is tamper-resistant.

In the second scenario, the adversary is malware running on the user's machine and the user is simply an unwitting carrier of the attack. Like the adversarial user, the goal of the malware is also to execute code without notifying the remote party. While malware has the ability to arbitrarily modify the software running on the machine, it does not have physical access to the machine making it a strictly weaker adversary than an adversarial user. We note that the previous two adversaries are identical to the adversaries that current attestation tries to defend against.

The third scenario contains an adversarial remote party who tries to violate the user's privacy and learn what applications a user is running. However, the Trusted Intermediary only gives the remote party the ability to propose and terminate contracts. A possible attack is one where the remote party iteratively proposes different contracts and notes which ones the user is able to make forward progress on. With this information, the adversary may be able to deduce some of the programs that are installed on the user's machine and thus violate her privacy. However, the remote party cannot learn all programs that the user has installed with this attack because the user need not run all programs while under the contract. In addition, because the adversary must repeatedly convince the user to accept a number different contracts, the user is likely to become suspicious and terminate all contact with the adversary, making this attack difficult to mount in practice.

Attestation contracts introduce an adversarial remote party whose goal is to deny the user the use of her machine. To accomplish this, the adversary may trick the user into accepting a contract that cripples her machine by not allowing it to run some critical software. Current attestation methods are not vulnerable to such an attack because the remote party does not specify what the user may run, but instead receives a list of all software the user has run. In exchange for the privacy offered by attestation contracts, our solution must defend against such an attack by allowing the user to arbitrarily terminate a contract that is too restrictive. However, we explicitly exclude the case where the remote party is able to execute privileged code on the user's machine. This is because a remote party with this ability can both violate the user's privacy or deny her service without the need to abuse attestation contracts in the first place.

5.3 Attestation Contract Operation

A contract consists of a list of permitted software that the user can run. As in current attestation, software is identified in an attestation contract through a cryptographic hash of the software binary. Using a cryptographic hash not only identifies the software , but also ensures that the software has not been modified in any way. Unfortunately, the remote party cannot simply send this list of hashes to the user's OS and expect it to enforce the contract. This is because the remote party has no guarantee that the user's OS is in a trustworthy state, and will not be able to bring it into such a state without forcing a reboot. As a result, we require an intermediary, which is trusted by both the user and the remote party, to enforce the contract. In our prototype implementation, the *Trusted Intermediary* fulfills this role. The Trusted Intermediary should be implemented in a manner that protects it from tampering by both the user and any malware on the user's machine. This is accomplished in our prototype

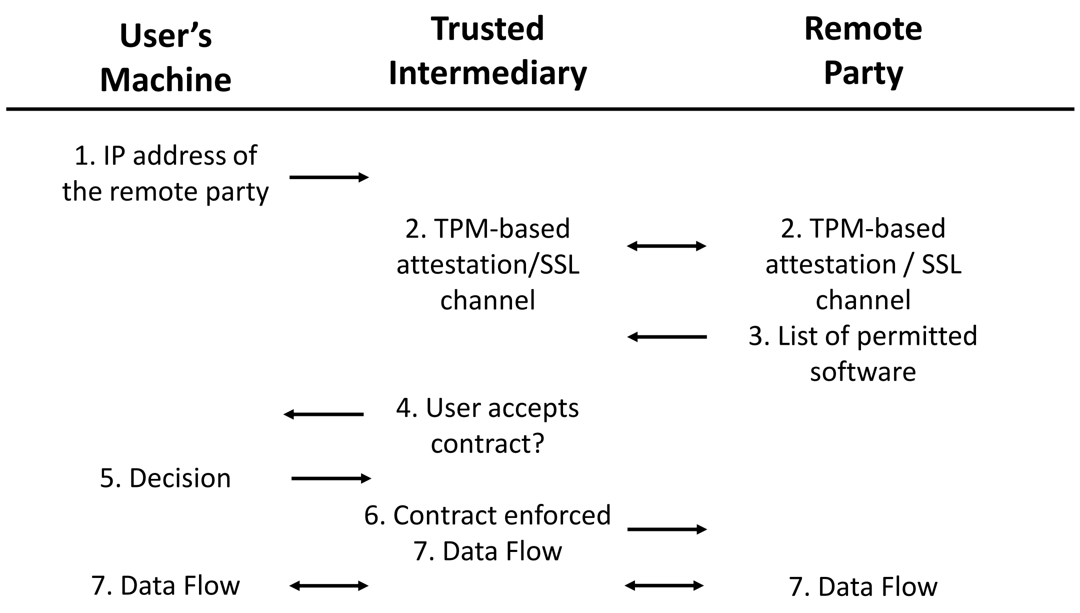

User's Machine
Trusted Intermediary
Remote Party
1. IP address of the remote party
2. TPM-based attestation/SSL channel
2. TPM-based attestation / SSL channel
3. List of permitted software
4. User accepts contract?
5. Decision
6. Contract enforced
7. Data Flow
7. Data Flow
7. Data Flow

Figure 5.2: Steps in successful contract negotiation

by using execution monitoring. As described in Chapter 3, Patagonix cannot be tampered with or mislead by code executing in the monitored VM.

Before the Trusted Intermediary begins enforcing a contract, it must first perform a contract negotiation between the user and the remote party. The process of contract negotiation is illustrated in Figure 5.2. A contract negotiation is initiated when the user attempts to connect to a remote party that requires guarantees on the integrity of the user's machine. In Step 1, the user's machine enlists the Trusted Intermediary to establish a contract by providing the network address of the remote party. The Trusted Intermediary then contacts the remote party, and uses the TPM on the user's machine to attest to its integrity in Step 2. Provided that the Intermediary is able to successfully attest itself to the remote party, it then negotiates a shared secret with the remote party and establishes a secure communication channel. This secure channel is implemented in our prototype with SSL, but could have also been accomplished with a network layer protocol such as IPSec. In Step 3, the remote party sends the names and hashes of permitted software binaries, as well as all required libraries, to the Trusted Intermediary. Along with this list, the contract may also specify some *refinements*, such as the inclusion of heartbeat applications. We will elaborate further on these refinements in Section 5.4. In Step 4, the Trusted Intermediary displays the names of software permitted under the contract, along with any refinements, to the user and the user is asked to either accept or reject the contract. If she accepts in Step 5, then the Trusted Intermediary notifies the remote party in Step 6 and immediately enforces the terms of the contract on the user's machine.[1] At this point, all traffic flowing from the user's machine to the remote party is tunneled through the Trusted Intermediary who signs every packet sent to the remote party with a key negotiated over the secure channel to authenticate it as shown in Step 7. All applications that are currently running on the system and are permitted to run by the contract will continue to execute unaffected. Applications that are not listed in the contract are immediately suspended and will only be resumed when the contract terminates. Similarly, if the user attempts to start an application that is not permitted by the contract, the application will be immediately suspended before it can execute a single instruction.

Recall that a malicious remote party may attempt to mount a denial-of-service attack on the user's machine by using a severely restricted attestation contract. To prevent this, the Trusted Intermediary ensures that a user can always terminate a contract after accepting it, should they find it too restrictive. A user can do this via a communication channel that the user's machine has with the Trusted Intermediary. Even if this channel is made unusable by a restrictive contract, the Trusted Intermediary can intercept all keystrokes and the user can use a particular key sequence to cause the Trusted Intermediary to end the contract.[2]

The Trusted Intermediary guarantees that if the contract is terminated or broken by the user, the remote party will be aware of this fact before it receives any more network packets from the user, including packets which may have been generated by code not present in the contract. The Trusted Intermediary will only sign packets sent by the user's machine to the remote party while the contract is still in effect. Thus, when the remote party receives an unsigned packet from the user, it knows the user has terminated the contract. Even if the user takes drastic measures to forcibly end the contract by power cycling the machine, or physically disconnecting the network cable and plugging it into another machine, subsequent communications sent by the user to the remote party will not be signed by the Trusted Intermediary and the remote party will be aware that the contract has been broken.

Regardless of which side terminates the contract, once the contract is terminated, the user is free to enter into another contract, and doing so does not require rebooting the user's machine. In our current model and implementation prototype, we restrict the user to having only one active contract at any given time. We defer the discussion of how to negotiate and enforce multiple contracts simultaneously to future work.

[1]This notification is not necessary and is only included here for clarity. The remote party trusts the Trusted Intermediary not to send data from the user's machine until the contract is in place.

[2]This is similar to the mechanism used by Windows to activate the login screen.

5.4 Contract Refinements

In addition to a list of software, the attestation contract can be augmented with additional refinements. The first two refinements are restrictions that remote parties may find useful to place on the user. If specified, these refinements are displayed to the user at the same time as the list of permitted software, and must be approved by the user before the contract is accepted. The last refinement allows users to amend a contract that is too restrictive without having to renegotiate a new contract.

When proposing a contract, the remote party may specify a list of *heartbeat applications*, which the Trusted Intermediary will ensure are executing during the contract. For example, the remote party may use heartbeat applications to guarantee that the user's machine running a virus scanner or a memory scrubber. Along with the heartbeat applications, the remote party also specifies a maximum *heartbeat period*. If the period between OS scheduling for these heartbeat applications exceeds the heartbeat period, the Trusted Intermediary will terminate the contract and inform the remote party. This prevents a malicious user or malware from pausing or lowering the priority of the heartbeat application. Naturally, the remote party should specify a period that is generous to account for unexpected load on the operating system.

The remote party may also specify *network restrictions*, which limit the allowed source or destination of network traffic. These may be used by the remote party to restrict network flows while a contract is in place. For example, a participant in an online game may try to cheat by running a tampered client on a different machine and then use user's machine that is under contract as a network proxy that will accept packets from the tampered client and then forward them to the Trusted Intermediary. In this case, the Trusted Intermediary cannot tell if the packets originated from the user's machine, which is under contract, or the machine running the tampered client. To mount this attack, the malicious user must have negotiated a contract that includes the necessary software to create such a proxy on the user's machine. Rather than try to ensure that no combination of software in the contract can be used to create such a proxy, it is much simpler to restrict the set of hosts that the user's machine can receive packets from.

Finally, specifying the complete list of legitimate programs may make the list impractically long. Instead the remote party may just specify the most commonly found legitimate programs and combine that with a refinement that allows the user to *extend* the contract dynamically. Thus, if after accepting the contract, the user wants to run a program that is not in the initial contract, she may make a request to the remote party to extend the contract to include the new software. If the remote party agrees, hashes are sent and added to the current list allowed software. Note that to do this, the user must reveal to the remote party the program she is attempting to run. If the user does not want to expose such information, she may either terminate the contract, or choose not to run the program.

5.5 Security Guarantees

In summary, attestation contracts provide strong security guarantees to both the remote party and the user. The strength of the guarantees that attestation contracts provide to the remote party are either equivalent or stronger than those of current attestation methods. We will analyze these guarantees during three periods of attestation usage. The first period is before the attestation is performed. Current attestation reports a list of all software that has run on a machine to the remote party before the attestation occurs. On the other hand, attestation contracts do not report any information about what software has run before the attestation. Current attestation must report all software that has run because it only measures software as it is loaded – once loaded, malicious software can modify other applications or load new code without measuring it. Thus, current methods conservatively assume that malicious code that has run in the past may still be running when the attestation occurs. In contrast, attestation contracts can detect whether certain software is running or not, so they do not require all software that has run since boot to be reported to the remote party.

Before a contract has been put into effect, it is possible for a malicious program to tamper with another program while it is running through an interface that allows arbitrary access to another program's address space, such as the debugging facility. If the malicious program tries to alter instructions in the victim program, the Trusted Intermediary will detect the tampering and prevent the altered victim program from executing. However, if the malicious software restricts itself to only tampering with the data in the victim program's memory, and the victim program is on the list of permitted software in a contract, then it will continue to run with the tampered data when the contract is in effect. This can be prevented by having the remote party request programs permitted by a contract to be restarted when the contract is invoked, thus eliminating the tampered data.

The next period is when the user is interacting with the remote party after attestation has occurred. With current attestation, the remote party has no guarantees that the user's machine remains in a legitimate state after the attestation has occurred. Even if the remote party requires the user to attest periodically, there is still a window of vulnerability between the attestations. Attestation contracts provide stronger guarantees, since the Trusted Intermediary guarantees to the remote party that the user will continuously adhere to the terms of the contract.

The third and last period is after the remote party and the user have terminated their communication. Neither current attestation nor attestation contracts make any guarantees to the remote party during this period. In most settings where attestation will be used, events that occur after the remote party has stopped interacting with the user are of no consequence.

In addition, the Trusted Intermediary also protects the user against a malicious remote party who abuses contracts to deny the user the use of her machine. This is accomplished by allowing the user to terminate a contract at any time, and thus regain full control over her machine. Furthermore, no information about what software is running on the user's machine is divulged. A remote party can only be sure that the user has not run any software that is not on the list of permitted software.

Since all guarantees that attestation contracts provide are enforced by the Trusted Intermediary, the guarantees are only as good as the Trusted Intermediary itself. To securely implement attestation contracts, the Trusted Intermediary is a *closed system*, meaning that the user can neither alter its configuration, nor add any code to its binary. The remote party verifies this attribute, as well as the integrity of the Trusted Intermediary during the TPM-based attestation phase of the contract negotiation.

Like current attestation methods, attestation contracts have limitations. For example, neither method prevents attested software from being used for malicious purposes, and neither method prevents attested software from being compromised or hijacked as a result of flaws in the software.[3] In both cases, the onus of selecting what software can be trusted is placed on the remote party. In addition, neither attestation method will verify the integrity of data on the system. In particular, if JIT compilers are allowed to execute, the code they generate cannot be verified. In addition, both systems rely on the integrity of the TCB, which in current attestation schemes is the operating system kernel (or a hypervisor in the case of Terra [26]). Likewise, attestation contracts rely on the Trusted Intermediary which is implemented with a hypervisor.

5.6 Implementation

To demonstrate the practicality of attestation contracts, we have implemented a system prototype, which is depicted in Figure 5.3. We begin by describing the overall architecture, and then follow with details on the individual components. Finally, we will also discuss how contract refinements are implemented.

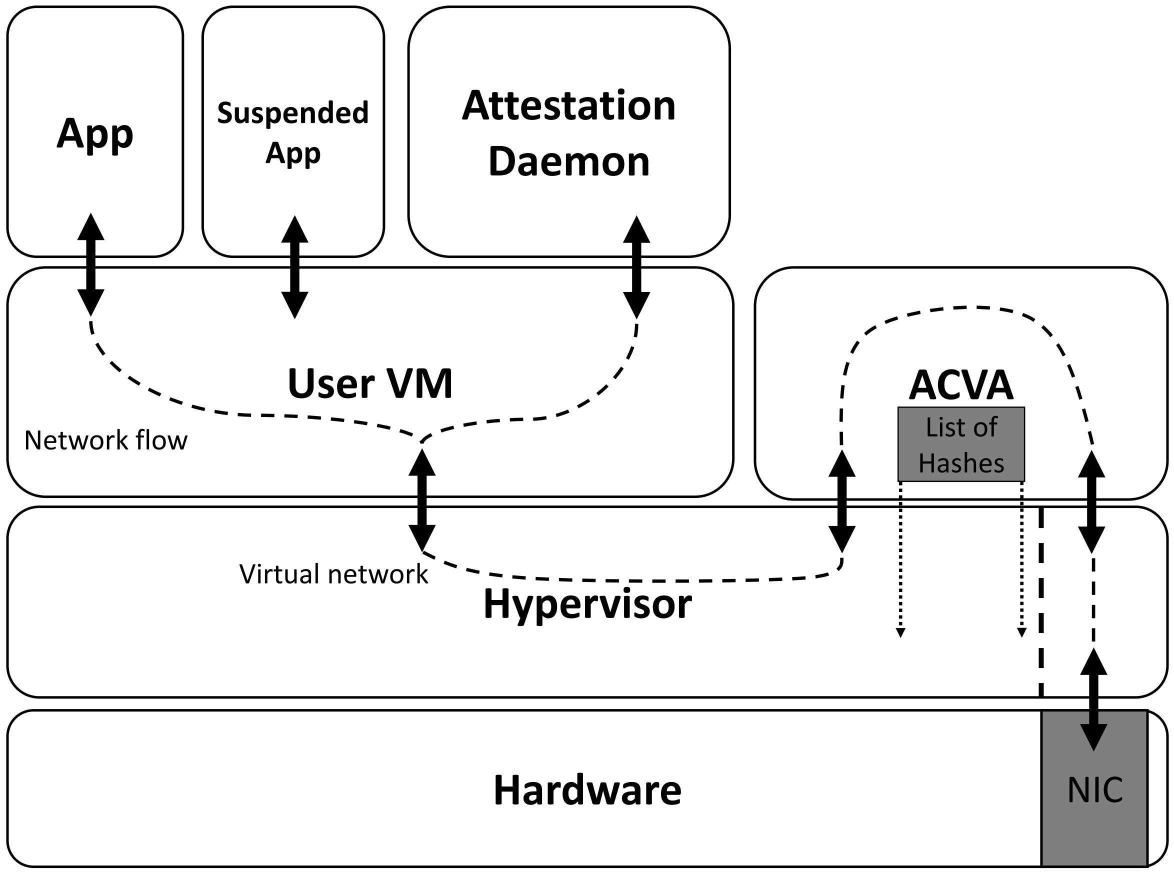


Figure 5.3: Attestation Contract System Architecture. Our prototype implements the Trusted Intermediary by using the Xen hypervisor and an Attestation Contract Virtual Appliance (ACVA). The user's machine is implemented in a VM called the User VM, and an Attestation Daemon, which runs inside the User VM, facilitates interaction between the user and the ACVA. While a contract is in effect, all network traffic to and from the User VM is tunneled through the ACVA.

5.6.1 System Architecture

The hypervisor in our prototype controls the underlying processor and presents a virtual hardware interface to the VMs running above it. The Trusted Intermediary is implemented through a combination of functionality within the hypervisor, and functionality within a special virtual machine (VM) called the *Attestation Contract Virtual Appliance (ACVA)*. The user's machine is also placed within another VM called the User VM.

In our prototype, the ACVA is a VM containing a Linux OS (Fedora Core 4). It includes the same components as the Patagonix VM, namely identity oracles and an extended control logic to implement attestation contract functionality. The ACVA is strictly a virtual appliance [80], meaning that even though it runs Linux, it acts more like a device than a full operating system – the ACVA has a predetermined set of behaviors, and cannot be extended or configured by the user of the machine. Thus, both the ACVA and the hypervisor are closed systems. Similar to execution monitoring, the Trusted Intermediary could have been implemented entirely within the hypervisor.

The ACVA interacts directly with the remote party as well as with the User VM, which runs an unmodified OS. Attestation contracts do not require modifications to the kernel, but do require an application-space *Attestation Daemon* that is running in the User VM during contract negotiation. This daemon has two main responsibilities. First, it is responsible for displaying contracts to the user, and recording whether the user has accepted or rejected the contract. Second, if contracts are accepted, the Attestation Daemon is also responsible for sending the binaries of the software permitted by the contract to the ACVA. The Attestation Daemon and the ACVA communicate via a virtual network that the hypervisor creates between the the User VM and the ACVA.

5.6.2 ACVA and Attestation Daemon

The ACVA and Attestation Daemon are responsible for two high-level contract operations – negotiating contracts and converting the list of permitted software into a set of code hashes that the underlying hypervisor can use. To begin a contract negotiation, the user provides the IP address of the remote party to the Attestation Daemon, which passes it through the virtual network to the ACVA. The ACVA uses the external network to contact the remote party. A TPM-based attestation procedure is used by the remote party to verify the integrity of the ACVA and the hypervisor. The ACVA then sets up a secure communication channel using SSL. Next, the remote party sends the ACVA the proposed contract, which contains a list of cryptographic hashes and application names which are allowed to execute. In our prototype, we

[3]However, attestation contracts prevent code injection attacks.

use SHA-256 hashes. The ACVA passes the contract terms to the Attestation Daemon, which presents the list of permitted applications, along with any refinements listed in the proposed contract to the user and prompts the user to accept or reject the contract. After reviewing the contract, the user responds with her choice to the Attestation Daemon, which passes the user's response to the ACVA. If an adversarial remote party is able to intercept and tamper with the channel between the Attestation Daemon and the user, she will be able to force the ACVA to accept a contract and deny service to the user. However, this falls outside of our attack model, since a remote party that can prevent the user from interacting with applications can prevent the user from using her machine without having to abuse an attestation contract or the Trusted Intermediary.

If the user accepts a contract, the ACVA and the Attestation Daemon must generate the database of binary information that is required for execution monitoring: page hashes, relocation information, entry points and hashes of sequences for the address inference algorithm. The proposed attestation contract only provides the ACVA with the per-binary hashes of permitted software. To convert this into a list of per-code-page hashes, the Attestation Daemon must transmit the binaries of the permitted software to the ACVA. The ACVA verifies that these binaries correspond to cryptographic hashes in the contract to ensure that the binaries in the User VM have not been tampered with, and then extract the required information from the binaries. The Attestation Daemon only helps the ACVA communicate with the user and provides the ACVA with software binaries. Because the integrity of the software binaries is verified independently by the ACVA, the Attestation Daemon does not have to be trusted for the attestation contract guarantees to hold.

All network traffic between the User VM and outside world is routed through the ACVA while a contract is in place. The ACVA signs all packets transmitted by the User VM to the remote party. There are two possible attacks that a malicious user can mount if they have control of another VM. First, she may try to use another VM that is not under contract to send traffic through the ACVA and get it to sign the packets. To prevent this attack, we utilize the Xen reference monitor [77] to ensure that the User VM that is under a contract may send network packets to the ACVA to be signed. Second, a malicious user can try to impersonate the ACVA and interact directly with the remote party after the contract is in place. However, this would require stealing the key that the ACVA uses to sign packets, which was negotiated as part of the SSL session established during the TPM-based attestation with the remote party. The hypervisor ensures that the ACVA is isolated from other code running on the machine, thus making it impossible for others to obtain the signing key.

5.6.3 Hypervisor Modifications

Implementing attestation contracts functionality only requires a minor change to the hypervisor in addition to the changes required for execution monitoring and discussed in Chapter 3. This modification allows the hypervisor to recognize and intercept a terminate-contract key sequence from the user. Since the Xen hypervisor virtualizes all devices including the keyboard, it can watch for the terminate-contract key sequence. When it sees the key sequence, it terminates the current contract and immediately informs the ACVA.

5.6.4 Implementing Attestation Contract Refinements

To verify that heartbeat applications are running, the ACVA uses the refresh mechanism. The hypervisor is periodically instructed to mark all code pages of the heartbeat application non-executable. As a result, the next time the application is executed, a page fault will occur. For each heartbeat application, two parameters are specified: t and t'. t specifies the interval at which the hypervisor will check that the heartbeat application is executing, and $t + t'$ specifies the maximum interval that a heartbeat application may not have executed before the hypervisor considers the contract violated. If no page fault occurs for that application for more than t seconds, possibly because all the code pages used by the application have already been checked and marked executable, the hypervisor clears the executable bit for all code pages of that application. If no page fault occurs for more than t' seconds after all code pages of the application have been marked non-executable, the hypervisor terminates the contract.

Network restrictions are enforced by using the sHype reference monitor [77] in Xen to force all traffic to and from the User VM to go through the ACVA. After this, the ACVA acts as a router and enforces firewall rules on the User VM's traffic to satisfy the network restrictions imposed by the contract.

Contract extensions are fairly straight forward to implement, and their operation can be viewed as a simplified version of a full contract negotiation. The main difference is that both the Xen hypervisor and the ACVA have already been attested, so there is no need for them to attest again. The ACVA then forwards the hashes and identity of the application the user wishes to add to the contract, and the remote party responds by either accepting or denying the request.

5.7 Impact on the TCB

Our system model assumes that the user is running her applications on the User VM. Thus, the TCB in our system consists of the code that the security of the User VM depends on. Our prototype does not require any modifications to be made to the kernel of the User VM. In addition, the ACVA and the Attestation Daemon are not part of the TCB – the Attestation Daemon runs as an unprivileged process on the User VM, and the ACVA runs in a separate VM. Thus, a vulnerability in these components could affect the proper operation of the attestation contracts, but could not be used by the adversary to elevate privileges on the User VM. Because the termination key sequence is handled entirely in the hypervisor, the ACVA has no influence on the availability of the User VM either. However, the hypervisor is part of the TCB of all VMs running on the system, so we attempted to keep the modifications to this component to a minimum when designing our prototype. Our modifications to the hypervisor were mainly limited to the memory management subsystem and only represent 1256 lines of code.

5.8 Conclusion

Attestation contracts improve on current attestation methods by enabling attestation guarantees continuously over a period of time, freeing the user from having to reboot frequently or invoke a late launch hypervisor, and enabling the user to keep what software she runs on her machine private. The key difference that enables these improvements is that while current attestation is a finite statement about the state of a user's machine at a particular point in time, attestation contracts dynamically and continuously impose restrictions on a system throughout the duration of a contract via an active Trusted Intermediary. The key mechanism that enables this is execution monitoring. We have shown that attestation contracts can provide all the guarantees that current attestation provides, while at the same time providing additional guarantees and functionality. We also show that a malicious remote party cannot abuse attestation contracts to mount a denial-of-service attack on a user by unfairly limiting what code can be run on a user's machine.

Chapter 6

Evaluation

In this chapter, we evaluate the effectiveness and performance impact of both execution monitoring, as implemented in Patagonix, and file monitoring, as implemented in P2. For execution monitoring, we test Patagonix on 9 rootkits and verify that it is able to identify code hidden by every one of them. In addition, we determine the overhead introduced by our modifications to the hypervisor through both microbenchmarks and macrobenchmarks.

For file monitoring, we validate that the approach is accurate by running our prototype to monitor specially designed workloads. For these workloads, we obtain the list of files they access through logging. We then compare the list of files accessed as reported in the logs with the list of file accesses reported by the File monitor. Finally, we evaluate the performance overhead introduced by the monitoring.

6.1 Execution Monitoring

All experiments were carried out on a machine with an AMD Athlon 64 X2 Dual Core 3800+ processor running at 2GHz, with 2GB of RAM. We used the Xen 3.0.3 hypervisor and allocated 512MB of RAM to the monitored VM and 1GB of RAM to the `Domain 0` VM, which also doubles as the Patagonix VM. Unless stated otherwise, the monitored VMs contain either Windows XP SP2 or Fedora Core 5 with a 2.6.19 Linux kernel.

6.1.1 Effectiveness

To evaluate the effectiveness of Patagonix at identifying covertly executing binaries, we used Patagonix to monitor VMs containing the nine rootkits listed in Table 6.1. These rootkits target the Windows kernel and Linux kernel versions 2.4 and 2.6. For this experiment, they were installed in VMs running Windows XP SP2, version 2.4.35.4 of the Linux kernel, and version 2.6.14.7 of the Linux kernel (The rootkits that targeted Linux 2.6 kernels did not work

Target OS	Rootkits
Linux 2.4	Adore, Adore-ng, Knark, Synapsys
Linux 2.6	Adore-ng-2.6, Enyelkm
Windows XP	Fu, Hacker Defender, Vanquish

Table 6.1: Rootkits detected by Patagonix. In reporting mode, Patagonix is able to identify processes hidden by these rootkits and/or detect tampering of processes by these rootkits. In lie detection mode, Patagonix detects that the OS is under reporting the binaries that are running.

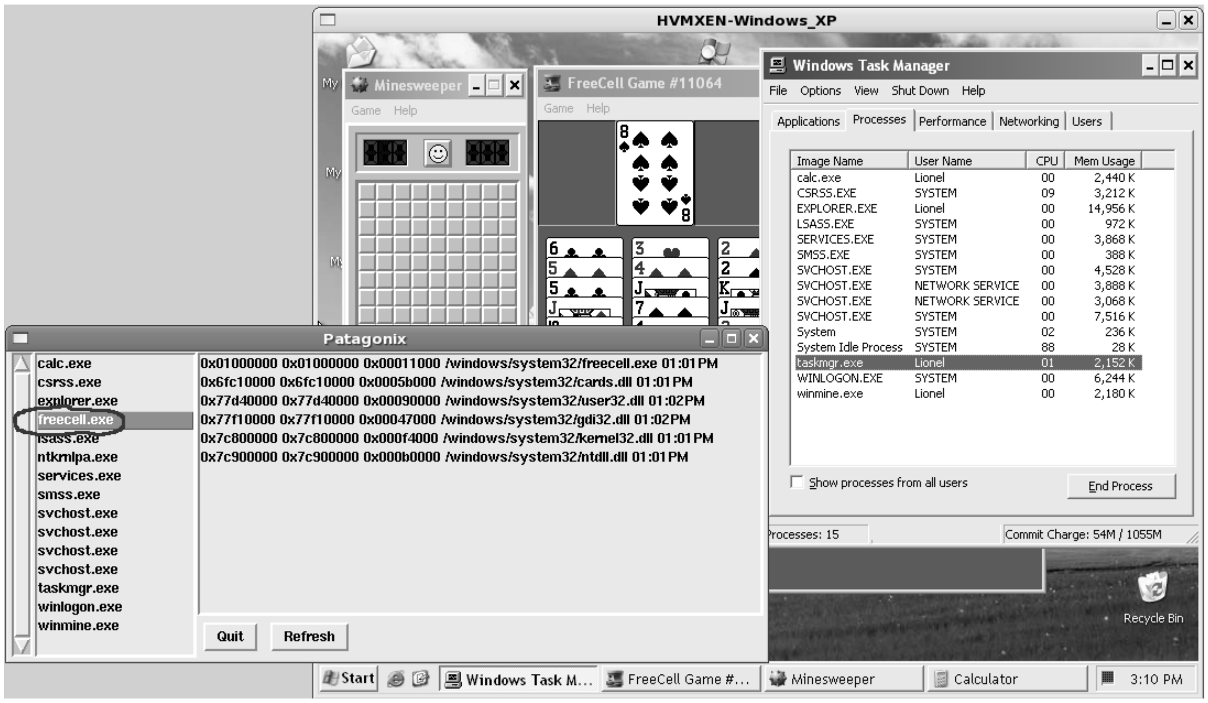
with version 2.6.19 of the kernel). We evaluated Patagonix in both reporting and lie detection mode.

First, we ran Patagonix on monitored VMs that have been infected with the rootkits. Each rootkit (except Vanquish) was configured to hide a process on the monitored OS: an instance of `Freecell` on Windows and an instance of `top` on Linux. We then verified that the hidden processes were not visible to the standard execution-reporting utilities on the respective OSes. In reporting mode, Patagonix was able to neutralize all the rootkits and report the execution of the covert code to the administrator, as illustrated in Figure 6.1. Likewise, in lie detection mode Patagonix is able to detect the tampering performed by each of the rootkits without fail. The Vanquish rootkit does not hide processes like the other rootkits. Instead, it tampers with applications by injecting code into the address space of executing processes. In these cases, the executing code of the tampered binaries is correctly identified as "not present" since it no longer matches any binary in the database. This warning should be interpreted as a likely rootkit infection by the administrator since the only other cause would be a missing binary in the trusted database.

Second, we ran Patagonix on VMs that did not have any rootkits installed to see if Patagonix reports any false positives. We exercise the VMs using the various application and microbenchmarks described in the following sections. During these tests, all executing code was correctly identified. When run in lie detection mode on an uninfected VM, Patagonix reported no discrepancies between the processes reported by the monitored OS and that detected by Patagonix.

6.1.2 Microbenchmark

To understand the overheads introduced by Patagonix, we devised *chain*, a microbenchmark that touches a new page of code on every instruction by chaining together a series of jumps, each targeting the beginning of the next page. Chain represents the worst case scenario for Patagonix: every instruction requires Patagonix to identify the new page of executable code.



```
calc.exe
csrss.exe
explorer.exe
freecell.exe
lsass.exe
ntkrnlpa.exe
services.exe
smss.exe
svchost.exe
svchost.exe
svchost.exe
svchost.exe
taskmgr.exe
winlogon.exe
winmine.exe
```

```
0x01000000 0x01000000 0x00011000 /windows/system32/freecell.exe 01:01PM
0x6fc10000 0x6fc10000 0x0005b000 /windows/system32/cards.dll 01:01PM
0x77d40000 0x77d40000 0x00090000 /windows/system32/user32.dll 01:02PM
0x77f10000 0x77f10000 0x00047000 /windows/system32/gdi32.dll 01:02PM
0x7c800000 0x7c800000 0x000f4000 /windows/system32/kernel32.dll 01:01PM
0x7c900000 0x7c900000 0x000b0000 /windows/system32/ntdll.dll 01:01PM
```

Image Name	User Name	CPU	Mem Usage
calc.exe	Lionel	00	2,440 K
CSRSS.EXE	SYSTEM	09	3,212 K
EXPLORER.EXE	Lionel	00	14,956 K
LSASS.EXE	SYSTEM	00	972 K
SERVICES.EXE	SYSTEM	00	3,868 K
SMSS.EXE	SYSTEM	00	388 K
SVCHOST.EXE	SYSTEM	00	4,528 K
SVCHOST.EXE	NETWORK SERVICE	00	3,888 K
SVCHOST.EXE	NETWORK SERVICE	00	3,068 K
SVCHOST.EXE	SYSTEM	00	7,516 K
System	SYSTEM	02	236 K
System Idle Process	SYSTEM	88	28 K
taskmgr.exe	Lionel	01	2,152 K
WINLOGON.EXE	SYSTEM	00	6,244 K
winmine.exe	Lionel	00	2,180 K

Processes: 15 Commit Charge: 54M / 1055M

Figure 6.1: Output of both Patagonix and the Task Manager when the FU rootkit is used to hide `freecell.exe`. Patagonix identifies all processes including `freecell.exe`, while the Task Manager does not display the hidden process. Patagonix identifies “System” as `ntkrnlpa.exe`, the name of the Windows XP kernel binary.

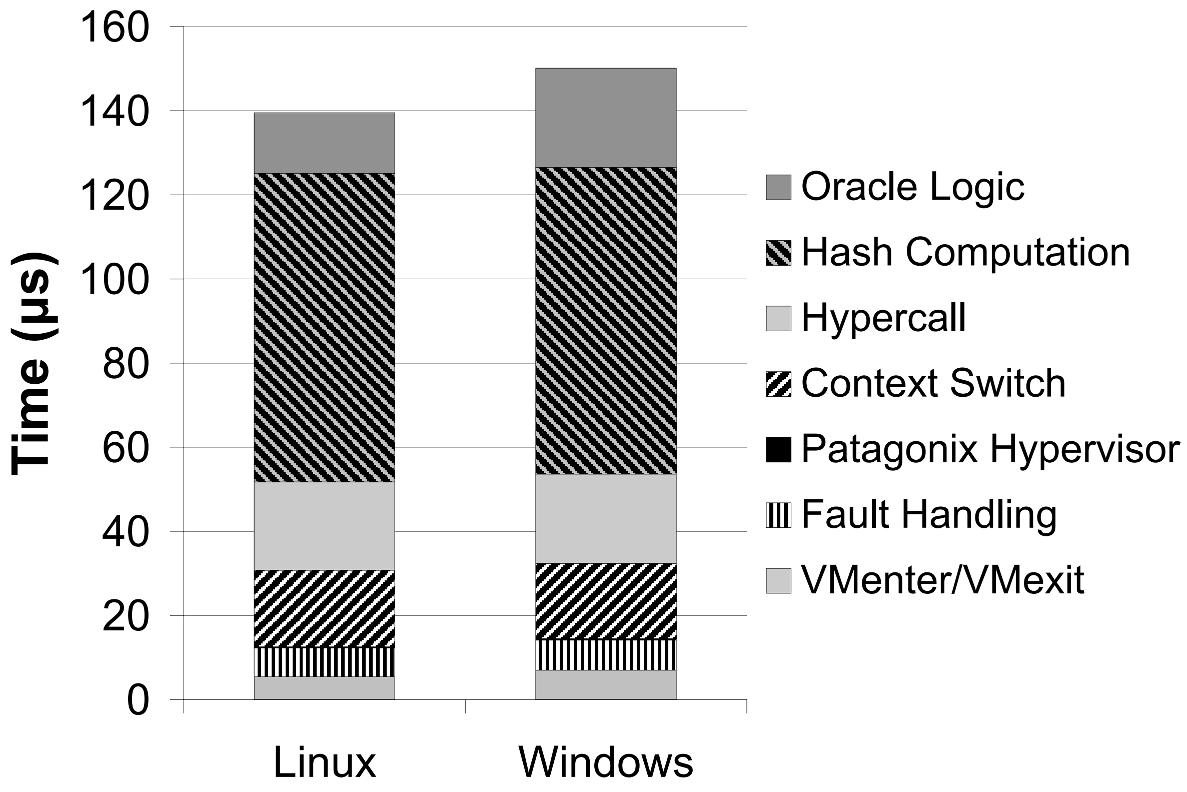


Figure 6.2: Execution time for various components of the identification operation. The total height of the bars represents the average time required to identify the origin of an executing code page.

Benchmark	**Linux (%)**	**WinXP (%)**	**WinXP-hw (%)**
Apache Build	1.68	2.62	1.99
Boot	2.05	30.39	10.63
SPECINT 2006	0.03	2.32	0.25
perlbench	2.06	23.01	1.42
gcc	13.75	12.43	3.48

Table 6.2: Application benchmark results. Results are the average of ten runs and are given in percent overhead over vanilla Xen. All standard deviations were less than 3% of the mean. WinXP-hw is estimated performance with hardware support for sub-page permissions.

We instrumented our prototype to break down the page identification process into its different components. Figure 6.2 details the overhead incurred when identifying one page of code; the values presented are the average of 10,000 Patagonix invocations, and the standard deviations for each component were consistently less than 5% of the average.

When reaching a new page of code, a page fault is triggered by the MMU. This results in an unavoidable hardware cost due to the VMexit and VMenter operations in and out of the hypervisor. After a VMexit, a software page fault handling cost is incurred that is specific to Xen's shadow page table implementation; we expect it to change with other hypervisor implementations. The Patagonix's hypervisor code is then executed; running this code is extremely brief (approximately 0.3μs), attesting to its minimal impact on the hypervisor. This code triggers a context switch into the Patagonix VM, where a hypercall is executed to retrieve the executing page information. These two operations cost a total of 40μs, but enable 2080 out of a total 3544 lines of code to be implemented in the Patagonix VM instead of the hypervisor. The hash computation necessary for all oracles accounts for 73μs, nearly half of the page identification time. As expected, the PE oracle logic takes slightly more time than the ELF oracle logic. We note that the case in which the PE search function has to match an entry-point page against several candidates will be more expensive, as each candidate binary requires a hash computation; we have observed times as high as 538μs. Fortunately, this only happens very rarely and the search is only performed once per binary mapped in memory.

6.1.3 Application Benchmarks

Since Patagonix is only invoked when code is executed for the first time, we expect this to coincide with page faults that load code from the disk. Because disk operations are expensive to begin with, we expect Patagonix overhead to be minimal in practice. To confirm this, we ran several application benchmarks in both the Linux and Windows VMs in our prototype.

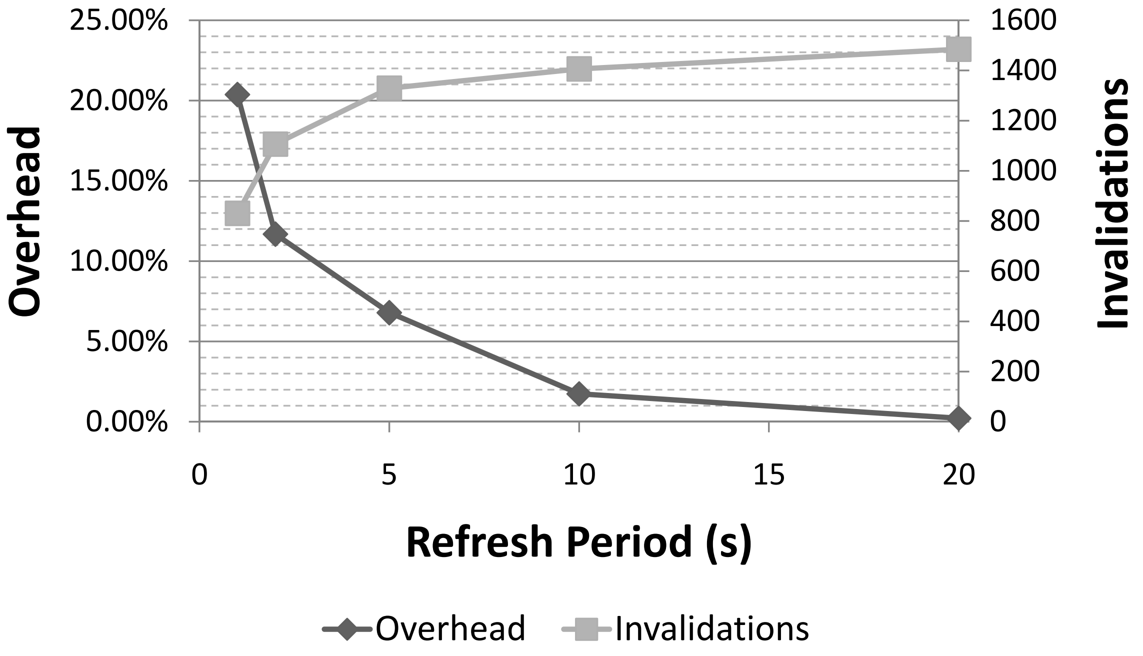


Figure 6.3: Overhead and Invalidations vs. Refresh Period. Apache Build on Linux. Averages of five runs with standard deviations below 2% of the average.

Computationally intensive applications are represented by the benchmarks from the SPECINT 2006 suite. For workloads with larger code footprints, we also measured the time Patagonix takes to boot Windows and Linux, as well as to build Apache. We compare the execution time for each benchmark against a vanilla Xen system running the same benchmark on the same monitored VM and report the overheads in Table 6.2. Since the PE oracle uses sub-page emulation, we also ran benchmarks without the emulation and sub-page checks (WinXP-hw column) to approximate what the performance might be if hardware support were available.

We report the SPECINT benchmarks as an aggregate because overheads for all benchmarks where less than 3% for the three configurations except for `gcc` and `perlbench`, whose performance we report separately. The Windows boot and `gcc` have large code footprints in comparison to their execution time: Windows initializes several services, drivers and interrupt handlers during boot, while SPEC drives `gcc` with a set of tests that exercises a large number of code paths. `perlbench` does not experience high overhead except in the Windows XP configuration because it spends a high portion of its time running code on mixed code/data pages, motivating architectural support for sub-pages in such cases. As expected, the overhead for all other benchmarks is low. This is because their code footprint is small relative to their execution time.

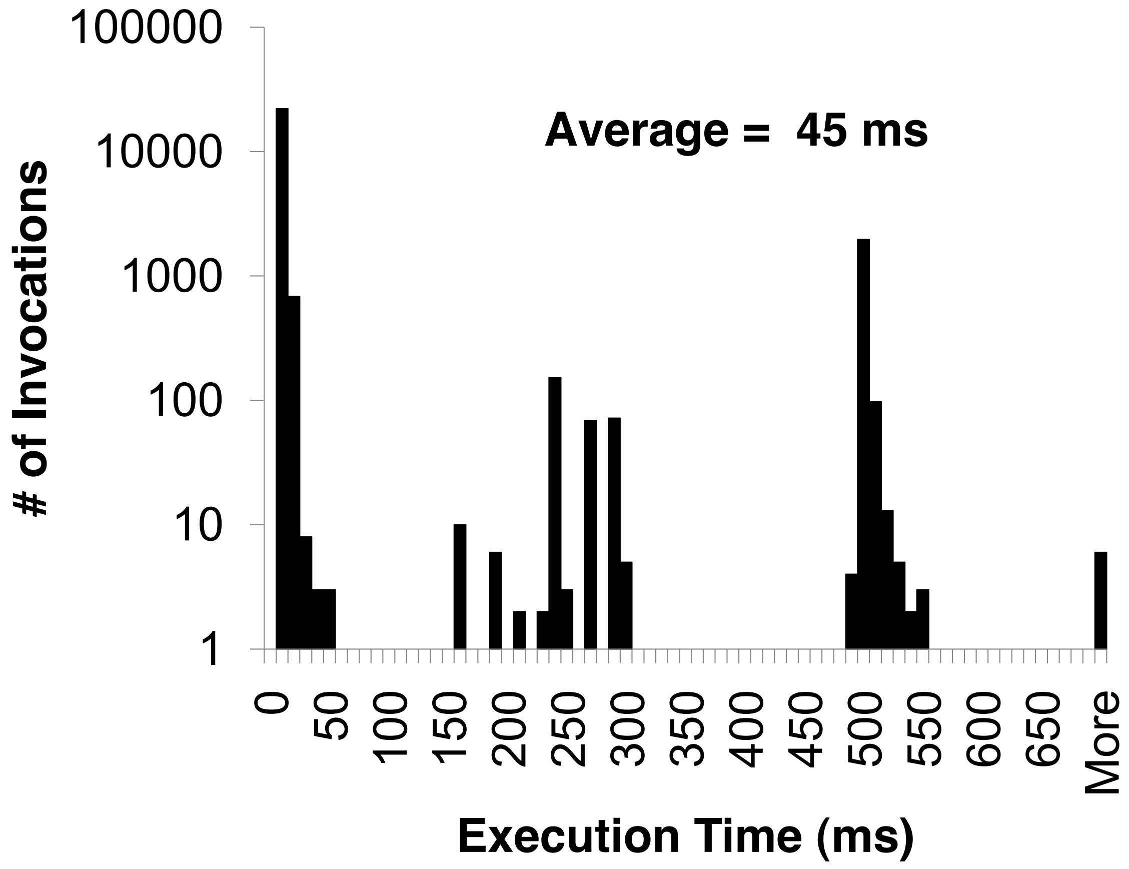


Figure 6.4: Histogram of address inference algorithm execution times.

Finally, the Patagonix VM needs to request periodic refreshes from the hypervisor. A shorter refresh interval means more accurate information about when a process was last observed executing, but also incurs more overhead. Figure 6.3 plots the additional overhead the Apache build benchmark in Linux experiences for various refresh periods, as well as the number of executable pages that are invalidated (set non-executable) each time. More frequent refreshes mean less time for the application to execute various pages, resulting in fewer invalidations.

The address inference algorithm is never invoked in the benchmarks above. This is because it is only needed when resuming a suspended VM. To evaluate its overhead separately from the overhead of the PE oracle, we instrumented the PE oracle to measure both the number of times it is invoked after a VM resumes and the amount of time it takes to identify a binary each time it is invoked. We then perform the Apache compile and suspend the execution of a Windows VM compiling Apache at 1 minute and 2 minute intervals. We found that on average, the P2 algorithm was invoked an average of 46 times with a standard deviation of 2.5 when using a 1 minute interval, and an average of 48 times with a standard deviation of 3 when using a 2 minute interval. This illustrates that few invocations of the address inference algorithm are necessary when resuming a suspended machine. This is because at most one invocation is necessary per binary file that was in the process of being executed when the machine was suspended.

The execution time of the address inference algorithm is essentially determined by the number of iterations of the two nested for loops in the algorithm, which are bounded by the number of candidate addresses that are wrongly inferred and the number of offsets that must be attempted. We graph a histogram of execution times in Figure 6.4. Note that the y-axis is logarithmic. While the execution times vary widely, there are two dominant cases, illustrated by the two peaks centered around 31 ms and 501 ms. The overall average execution time of the P2 binary identification algorithm is 45±143 ms. As a result, combining the average run time of the address inference algorithm and the average number of times it needs to be invoked, we surmise that Patagonix adds an average of 3 seconds of overhead every time a VM is resumed.

6.2 File Monitoring

In this section, we evaluate the effectiveness of file monitoring at detecting unpatched applications and the performance overhead introduced by our architectural introspection based approach to file monitoring. To evaluate effectiveness, we verify that the monitor can accurately identify which files are being accessed and correctly ascribe these accesses to processes while avoiding false positives. To evaluate the performance impact of monitoring, we assess the overhead introduced by our modifications to the Xen hypervisor and virtual block driver using both microbenchmarks and macrobenchmarks.

All experiments were conducted on an AMD Athlon 64 X2 Dual Core 3800+ processor running at 2GHz, with 2GB of RAM. We used the Xen 3.3.0 hypervisor and allocated 512MB of RAM to the monitored VM and 1GB to the `Domain 0` VM. `Domain 0` is a Fedora Core 9 distribution based Linux system running the 2.6.18.8 kernel with the Xen `dom0` patches applied. We pinned the `Domain 0` VM to the first core and the monitored domain to the second core to minimize VM scheduling effects. To demonstrate the OS-agnostic quality of file monitoring, unless otherwise stated, all tests were run on both Windows and Linux VMs. The Windows VM runs Windows XP SP2. The Linux VM is a Fedora Core 9 distribution with a 2.6.27.25 Linux kernel.

Timing was recorded using an external time server to eliminate clock skew introduced by the hypervisor. The time server is a second physical machine that runs a simple network daemon that answers `gettimeofday` queries. The various experiments query that daemon over TCP to

Setting	Vanilla	Execution monitoring	File monitoring and Execution monitoring
Linux	436.9s	455.8s (4.3%)	460.9s (5.5%)
Windows	581.3s	605.2s (4.1%)	617.7s (6.3%)

Table 6.3: Compilation time for Apache (overhead).

Setting	Vanilla	Execution monitoring	File monitoring and Execution monitoring
Linux	30.1s	30.3s (0.9%)	30.4s (1.0%)
Windows	34.2s	37.8s (10.6%)	37.9s (10.8%)

Table 6.4: Time to boot the system (overhead).

obtain wall-clock time measurements of benchmark execution times.

6.2.1 Effectiveness

We evaluate the ability of file monitoring to properly detect access to monitored files and to correctly attribute these accesses to the accessing process. To this end, we place a unique ruby script in each directory on the file system. This represents 4161 ruby scripts on our Windows file system and 10268 ruby scripts on our Linux file system. We then ran an anti-virus scan of the entire disk on an otherwise idle system and confirmed that all files were accessed by the anti-virus by examining the logs of the anti-virus. On Windows the anti-virus scanner was the Symantec Anti-Virus 10.1.5. On Linux, the anti-virus scanner was ClamAV 0.95.2. We also ran a specially crafted application that would randomly execute one of the ruby scripts every second. In both cases, the file monitor successfully detected that the files were being accessed and correctly attributed the accesses to either the anti-virus scanner or the ruby interpreter.

To test whether stress on the OS's buffer cache might affect the accuracy of the file monitor, we generated load on the system. The load was generated by a script that navigated the Firefox web browser, visiting randomly selected links starting from a seed website (`http://del.icio.us`), thus creating churn in its web cache. We did this at the same time as running the test above. Again, the file monitor attributed all accesses correctly. In addition, no access to any script was incorrectly attributed to Firefox or any other application running on the system.

6.2.2 Performance

File monitoring requires execution monitoring. As a result, it incurs overhead when new code is executed, when data is read from disk into the OS buffer cache or the address space of a process and when data in the buffer cache is read by a process. To evaluate the overall performance impact of file monitoring, we measured the time required to compile the Apache web server on both Windows and Linux. In each case, we ran file monitoring with a database containing 20,532 entries corresponding to files being monitored, and a comprehensive database of binaries to allow execution monitoring to identify executing processes. We ran the compilation of Apache three times: with no monitoring, with only execution monitoring and with both execution and file monitoring.

We also measured the time required to boot both Windows and Linux under the same three settings. To measure the time to boot Windows, a simple "start-up" script was added to the start-up folder. When it is executed, this script pings the benchmark script on Domain 0. The benchmark script on Domain 0 starts measuring time when it creates the Windows VM and stops measuring time when it is pinged by the start-up script. The experiment is conducted in the same way on Linux, except that we added the "start-up" script at the end of the System V initialization procedure.

When only execution monitoring is enabled, the system is equivalent to Patagonix. However, the system tested in this section is implemented on a newer version of Xen and some internal details of the execution monitoring have changed. We verified that the measured overhead for the execution monitoring component are consistent with the overhead reported in the previous section.

The results for the compilation of Apache are tabulated in Table 6.3. They are the mean of 4 runs, and the standard deviation was under 1% of the mean. In both the Linux and the Windows cases, the overall overhead of file monitoring was 4%. The overhead with only execution monitoring was actually slightly worse, but the difference is comparable to measurement error in our system. The results for booting the systems are tabulated in Table 6.4. They are the mean of 10 runs, and the standard deviation was under 2% of the mean. For Linux, the overhead was 1%. Similar to what we reported in the previous section, the overhead for booting Windows was higher, a little under 11%. Again the difference between only execution monitoring and both execution and file monitoring is within our measurement error. Thus, we surmise that the majority of the overhead is imposed by the auditing and identification of executing binaries and that file monitoring does not introduce measurable overhead.

To more accurately measure the overhead of file monitoring, we evaluate it in isolation. We ran a micro-benchmark that sequentially read 2 GB of data from the disk to measure the disk transfer rate. The vanilla transfer rate was 79 MB/s (mean of 10 runs, standard deviation under 1% of mean) and 78 MB/s with auditing. To assess the impact of the software database size on the file monitoring overhead, we reran the benchmark with a larger database containing 1,252,977 entries. The transfer rate in this scenario dropped slightly to 63 MB/s since more time was required to query the database on each disk access. The modest overhead incurred

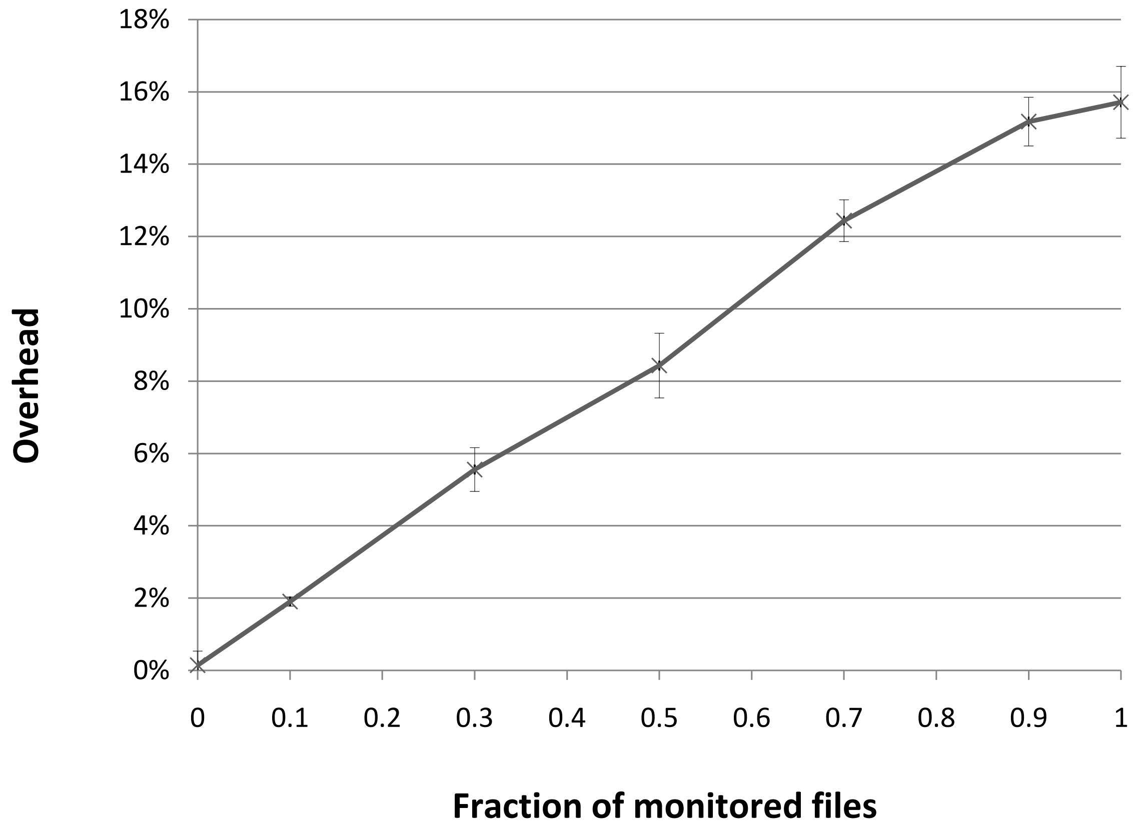


Figure 6.5: Overhead of scanning a directory containing 100,000 files as a function of the fraction of these files that are monitored. Overhead is relative to scanning the directory without File Monitoring. Averages of five runs, with error bars.

by file monitoring on transfer rates explains why no noticeable overhead is introduced on a macro-benchmark such as compiling Apache.

Finally, we evaluate the overhead caused by the page faults required to track files in memory. In practice, this overhead should be negligible. This is because we anticipate that only a small fraction of the files used by the machine will need to be monitored. Moreover, the monitoring of a file can stop as soon as it is being accessed by an interpreter. However, overhead will be incurred when an application other than the interpreter associated with a file accesses that file. To evaluate this overhead, we measure the time necessary to perform an anti-virus scan using ClamAV on Linux of a directory containing 100,000 files, each of size exactly one block (4 KB). We plot the overhead as a function of the fraction of monitored files in Figure 6.5. The overhead ranges from negligible (0.14%) when file monitoring is enabled but none of the files in the directory are monitored to 15.7% when all the files in the directory are monitored. To provide a point of reference, one machine in our lab used for general desktop computing and software development has 666,792 files. Of these files, 20,074 have an extension that appears in Table 4.4 or Table 4.5 and that corresponds to non-binary code. Even if all these files needed to be monitored, this would only represent 3% of the files on the system, resulting in an expected overhead of less than 2% on a workload similar to a virus scan.

Chapter 7

Related Work

7.1 VM introspection

VM introspection was first proposed in 2003 by Garfinkel and Rosenblum [27]. Their system, called Livewire, implements an intrusion detection system (IDS) with the help of a hypervisor, building on the idea by Chen and Noble [17] that this will isolate the IDS from the monitored system while providing it with greater visibility than a Network IDS. In addition to isolation and visibility, Garfinkel and Rosenblum emphasize that a VMM also provides the IDS with the ability to interpose on events of interest and that their introspection infrastructure enables event-driven policy modules to be invoked on certain hardware events. For example, Livewire can detect that the network card is being put in promiscuous mode, which may indicate an attempt to sniff network traffic. Other policy modules implemented by the authors include scanning memory for the presence of strings that contain known rootkit signatures, verifying the integrity of immutable sections of known binaries, and detecting the use of raw sockets by guest processes.

As emphasized earlier, the main limitation of Livewire is that it requires an *OS Interface Library* to parse the content of the memory used by the monitored OS. This means that an attacker may tamper with the guest OS data structures to hide his actions. While an attacker cannot disable the IDS itself, he can make it blind to what is actually occurring inside the VM. Using architectural introspection for execution monitoring enables the monitoring system to eschew parsing OS data structures.

Many other systems followed in the footsteps of Livewire. Copilot [66], ISIS [47], IntroVirt [43], VMM-based sensors [7], the use of specifications to detect semantic integrity violations [67], and SBCFI [65] were discussed in Section 2.3. These systems use introspection, either via a hypervisor or using a coprocessor to detect or monitor intrusions. Similar to a hypervisor-based IDS, a coprocessor-based IDS is isolated from the system it is monitoring and therefore cannot be tampered with by an attacker that has successfully compromised the monitored system. Unlike a VMM-based IDS, it cannot interpose on events occurring in the monitored system. Rather, it relies on periodic checking of the content of memory to detect anomalous changes of kernel data structures or virus signatures. The IDS can also access the disk and perform file integrity checking a la Tripwire. The use of periodic checking means that an attacker may evade detection for a small amount of time.

The idea was first explored by Zhang et al. [101]. Copilot [66] builds on their approach and offers a more thorough study of the approach, as well as techniques that can be applied using a hypervisor. Copilot is limited to taking hashes of immutable memory regions (kernel text segment, system call jump table and other kernel function pointer jump tables) in a known good state and then periodically ensuring that these memory regions have not changed. Copilot can only monitor data structures that are static or change infrequently. As a result, an attacker may inject code in the kernel and get it to execute by modifying function pointers in constantly changing kernel data structures.

In their follow-up work, Petroni et al. explored ways to monitor dynamic kernel data structures [67]. They use specifications describing consistency rules for important kernel data structures such as the `task struct` or the SELinux security decision cache that could be used by rootkits to hide their presence. These rules are written by kernel expert and the coprocessor periodically checks that they are not violated. The asynchronous nature of the checks render the checking difficult and prone to false alarms due to data structures being in an inconsistent state as a result of the kernel updating them. They emphasize that while their implementation leverages a coprocessor, the approach is also applicable to VMM-based intrusion detection. In addition, VMM-based IDSs may be able to check the data structures synchronously, overcoming some of the limitations of the coprocessor-based approach.

IntroVirt [43] leverages vulnerability-specific predicates to detect intrusions caused by specific vulnerabilities. The novelty of the system resides in the fact that it can leverage logs of past execution to detect intrusions that occurred before the distribution of the predicate by the software vendor. The key idea here is that an attacker may have taken advantage of a vulnerability before the software vendor released a patch, or between the time a patch was released and the patch was applied. By logging enough information to replay execution, a system administrator can replay past execution and check if an attacker took advantage of the bug fixed by the newly released predicate. A predicate is invoked at a specific invocation point within kernel or application code. When a breakpoint is hit, the VMM checkpoints the VM, executes the predicate code and then rollbacks the VM and resumes execution. In that way, the predicate can check pre-conditions for triggering a vulnerability right before the vulnerable code line is reached and generate an alarm if the pre-conditions are met. Rolling back the VM ensures that the predicate execution does not affect the VM. Inserting breakpoints require debugging information and, for breakpoint in application code, kernel instrumentation. In addition, writing a predicate requires detailed understanding of the application and may be hard to do without the source code of the application.

Lares [63, 64] and VMwatcher [39] are two additional examples of systems that use VM introspection to detect malware. Like Livewire, VMwatcher uses an OS interface library to reconstruct OS state from a snapshot of the memory of a VM. The authors present a detailed description of how the OS state is reconstructed that clearly highlights both the need for expert knowledge of the guest OS to implement such a system and the fragility of the approach to changes of the OS. They then demonstrate how the system can be used for passive outside-the-box monitoring, such as running an anti-virus scanner.

While VMwatcher takes a passive approach to monitoring, Lares [63,64] implements hooks similar to the ones used by Asrigo et al. [7] and focuses on trying to protect these hooks. The paper concludes that protecting such hooks is a challenging task that is specific to each implemented hook. In addition, the approach suffers from two common problems when using hooks: comprehensive hook placement is challenging, even for experts that have access to the source code of the monitored OS or application [90,100], and an attacker can simply create new code paths to avoid triggering the monitor.

Antfarm [40] is a system that was developed independently of our work and that is closely related in spirit to architectural introspection. Like architectural introspection, the authors describe a system that can bridge part of the semantic gap to detect how many processes are executed by a VM without information about the OS internals. To this end, the system monitors changes to the CR3 register. They then use this information to improve I/O performance via anticipatory scheduling, a technique that can be implemented by the VMM only if it can associate disk requests with particular processes. Antfarm could be improved by using the additional information that our more complete execution and file monitors gather.

7.2 Execution monitoring

Patagonix uses the principle of lie detection: comparing two views of the same data for discrepancies. This idea has been used by other systems for the same purpose. For example, Rootkit Revealer [20] and Strider GhostBuster [10] are systems that compare high-level and low-level views of the same system information. Rootkit Revealer detects differences between two views of the file system: one obtained by querying the Windows API and one obtained by reading the raw content of the disk. In this way, Rootkit Revealer can detect Hacker Defender and Vanquish, but not FU. This is because FU does not hide the presence of files on disk. Strider

GhostBuster uses the same technique to detect hidden files and extends it to detect hidden processes by comparing a view obtained using the Windows API and a view obtained by a driver loaded in kernel memory. However, in both cases, views are still derived from within the infected system, and a thorough rootkit can make both high-level and low-level views agree, thus eluding these systems. Both systems acknowledge this limitation and Strider GhostBuster suggests using outside-the-box scanning. Their outside-the-box scanning approach is different from the one taken by Patagonix: their hidden process detection approach requires specialized hardware to access memory from outside the OS and still requires parsing OS data structures.

Like Patagonix, other systems use a hypervisor to compare views taken from both within (i.e. inside-the-box) the infected system and outside (outside-the-box) the infected systems. Livewire [27] and VMWatcher [39] compare views of executing processes derived from the hypervisor with those gathered from within the monitored system. However, unlike Patagonix, these systems do not deal with asynchrony between the measurement times of the in-the-box and out-of-the-box views and will thus suffer from false positives.

Lycosid [42] also does lie detection by counting the number of address spaces in a VM. However, Lycosid does not identify which binaries the processes are executing. Besides, Lycosid [42] uses statistical inference techniques to detect hidden processes despite noisy input due to the aforementioned asynchrony. While the use of statistical inference helps overcome some of the difficulties encountered when comparing asynchronous measurements, short lived processes can still escape detection. As a result, Lycosid can only probabilistically detect when the number of address spaces does not match the number of processes reported by the OS. Because Patagonix identifies processes and registers callbacks with the OS, Patagonix is able to both precisely detect hidden processes, as well as identify which process is being hidden.

Independent to our work and using a similar low-level mechanism to detect code execution, SecVisor [82] restricts what code can be executed by a modified Linux kernel. SecVisor focuses solely on code that is executed in kernel mode. It uses a custom-made hypervisor, showing that execution control can be achieved with a small TCB. In contrast, Patagonix provides comprehensive guarantees for unmodified Linux and Windows OSes as well as the applications they execute, and demonstrates that these guarantees can be obtained by small extensions to a general-purpose hypervisor.

Control-flow integrity (CFI) [1] constitutes another line of research that tackles identifying and restricting what code can execute on a system. First, a control-flow graph (CFG) that represents the possible flows of execution of an application is constructed. An execution monitor is then in charge of ensuring that execution of the application conforms to the computed CFG. The execution monitor can be implemented inline [1] or using a dynamic binary rewriter [96]. CFI has the advantage that it can, at least in principle, address return-into-libc attacks, where an attacker does not inject new code but uses *stack-oriented programming* instead [84]. However, current implementation of CFI have substantial overhead and are hampered by the difficulty of obtaining an accurate CFG. When implemented inline, it is also challenging to add new code, such as drivers, to an application because generating correct inline execution monitoring code requires analyzing the entirety of the application beforehand. Finally, the feasibility of these approaches has not been shown for OS kernels.

State-based CFI (SBCFI) [65] is an attempt to apply an approximation of the CFI approach to operating system kernels. Rather than validate all control flow transfers, SBCFI periodically scans the VM memory to validate that the static kernel data has not been tampered with and that the kernel pointers point to valid targets. This requires acquiring information about the target OS using symbol information and expert knowledge. The approach is also limited because it does not handle return addresses on the stack and complex computed jumps. Moreover, transient attacks are possible due to the periodic nature of the memory scans. As a result, SBCFI provides weaker guarantees than our execution monitoring approach.

Other projects have manipulated the page tables used by the x86 MMU. For example, the PaX project [61] proposes manipulating these page tables to emulate the NX bit on older CPU that do no have hardware support for the feature. Finally, computer forensics experts [81] have demonstrated that PE binaries can be reconstructed by analyzing memory dumps. The starting point for the reconstruction is the data structure used by Windows to represent a process (EPROCESS). By using knowledge of the layout and content of this data structure, it is then possible to find where the header of the executable was loaded in memory and to then use information in this header to find where the code pages are loaded in memory. One can then write this content to a file to reconstruct the PE binary. The information acquired from the PE header is a subset of the information we acquire when constructing the binary database.

7.3 File monitoring

Dynamic information flow tracking (DIFT) and related taint tracking techniques have been extensively used to improve security. Suh et al. [89] and Taintcheck [57] both use taint tracking to detect corruption of critical pointers with external data. Suh's method requires specialized hardware but has low runtime overhead. Taintcheck uses a software-only method, but suffers from overheads of 30x or greater. Recently, Chang et al. [16] applied compiler analysis to optimize away some of the instrumentation required to perform taint tracking, achieving runtime overheads less than 13% without any specialized hardware. However, their compiler pass requires source code to perform the optimizations and instrument the program. Dalton et al. [22] introduce more complex security policies and a pointer identification algorithm to improve the accuracy of taint tracking. However, their technique requires specialized hardware to be added to the processor to be practical. P2 differs from previous uses of taint tracking for security in two respects. First, P2 uses taint tracking to determine if a non-binary file is accessed in an unsafe way while previous systems used taint tracking to detect attacks. Second, because P2's goals are different, P2 only needs to track taint at a page granularity, which allows it to leverage the processor MMU to monitor accesses to tainted data. This makes P2 far more efficient, allowing it to achieve low overheads without source code changes or special hardware.

Recently, information flow tracking was leveraged by Ho et al. [35] in a virtualization context to track data originating from the network in a virtualized environment. This allows the proposed system to determine if a VM ever attempts to execute data originating from the network. Like P2, page table manipulations are used to track tainted data at the page granularity. However, once tainted pages are accessed, the system switches to byte-level granularity and starts running VMs in an emulator, resulting in significant performance overhead. Because P2 uses coarse taint tracking, code can execute natively on the processor, resulting in modest performance overheads.

Finally, in a non-security context, others have also leveraged hypervisor virtualization of the MMU and disk to monitor systems. monitor the interaction between VMs and storage. Geiger [41] manipulates page table entries to detect evictions from the OS's buffer cache. This information allows optimizing memory allocations to VMs and helps implement a second-level buffer cache maintained by the hypervisor. Satori [54] hashes all disk content accessed by VMs to identify sharing opportunities between VMs to reduce memory consumption.

7.4 Attestation

In this section, we examine other approaches to attestation and contrast them with attestation contracts.

Both NGSCB [24] and Terra [26] allow partial attestation of software running on a system. NGSCB splits the system into two virtual machines, with one running a generic operating system and the other running Nexus, which is capable of attesting software running on it. The concept of separating applications in distinct virtual machines is generalized in Terra which permits any number of closed system VMs to be attested. In both cases, the paradigm of using a different virtual machine image for different applications is fundamentally different from our design, which allows the user to only have to interact with one system image. Attestation contracts allow a single system image to be dynamically switched between an open and a closed system. Further, neither of these systems consider the privacy of the user.

Sailer et al. proposed to combine their Integrity Measurement Architecture [79], which we have discussed in the previous section, with remote access security policies [78]. Here, the operating system kernel identifies all code loaded on the system, as well as information flows on the system. With Prima [38], Jaeger et al. revisit this approach using the more formal Clark-Wilson Lite integrity model [85]. Enforcing a MAC policy means that only the integrity of applications that can influence information flows the remote party cares about need to be measured. While these works share our goal of not having to attest to every single piece of software run on the platform since boot, they use standard attestation and therefore suffer from the aforementioned limitations of load-time only attestation, the need to reboot, and infringement on the user's privacy.

Griffin et al [33] advocate an abstract high-level language that a party involved in a distributed transaction can use to describe properties that other parties involved the transaction should have. Attesting virtual environments are then used to guarantee that involved parties indeed satisfy the required properties. However, the trusted virtual domains described by Griffin et al. are abstract constructs and they do not discuss which of the suggested properties can practically be attested to. We view attestation contracts as a concrete step towards an instantiation of such a vision, with Trusted Intermediaries as possible attesting virtual environments.

Both the Trusted Computing Group (TCG) [91] and Intel's LaGrande Technology (LT) [32] provide hardware support for remote attestation. TPMs conforming to the TCG specification are present in many machines. Recent AMD and Intel processors support late launch via AMD-V and Trusted Execution Technology (TXT), respectively. If no TPM is present, Pioneer [83] proposes a mechanism that allows the user to perform attestation without special hardware. Thus, it can be used instead of TPM-based attestation to attest for the Trusted Intermediary in our system. *K* is a system that protects privacy by utilizing trusted hardware to hide data access patterns [37]. *K* and attestation contracts are complementary – *K* maintains privacy for data usage, while attestation contracts maintain privacy for program usage.

Similar to our work, several approaches allowing finer-grained attestation have been proposed. Monrose et al. proposed to break down the computation performed by clients into units of work that can be independently verified and to have the remote party selectively verify units of work [55]. However, this technique is specifically aimed at verifying the results of a distributed computation. Attestation contracts, on the other hand, are applicable to a much wider range of scenarios. BIND [86] allows a user to only attest for a single process on a system, and narrows the gap between time-of-attestation and time-of-use by invoking a small sandboxed environment in which the process will run. However, BIND is not generic and applications need to be rewritten to use of such a mechanism.

Using cryptographic hashes to identify the software running on a machine is common practice, and was first used in the Aegis architecture [5] to limit what code can be used to boot the machine. While Secure boot gives a platform owner guarantees that all code loaded during boot corresponds to authorized code, it does not enable remote attestation.

Chapter 8

Conclusions and Future Work

In this thesis, we have introduced *architectural introspection.* We demonstrated through two systems, Patagonix and P2, that it can be used to provide useful abilities at the virtualization layer without having to resort to using knowledge about the internals of the OSes being virtualized. In addition, we evaluated both systems and showed that the performance overhead of architectural introspection is small enough to make it practical for real world deployment.

When we first started working on architectural introspection, we did not know how much knowledge about the monitored systems could be acquired without targeting a specific OS. The amount of information we were able to obtain exceeded our expectations. Exploring both Linux and Windows, we found out that while there are fundamental differences between the two OSes, they also have a lot in common. Undoubtedly, this is because the design of the hardware shapes how the OS will operate. As evidenced by even a cursory reading of CPU manuals, hardware designers have a very precise idea of how an OS should utilize the mechanisms they provide. Straying far from this idea will often result in inefficient and cumbersome designs in practice. As a result, distilling virtually unavoidable OS design principles into a way to perform introspection was a fruitful approach.

Besides, we found out that the performance impact of introducing additional hardware traps to detect monitored events was small. In our systems, the high cost of such traps is mitigated by the relative rarity of the monitored events under typical workloads. At the onset of our work, while we suspected that this may be the case, we only had a vague idea of what the overhead of the approach may be. By implementing prototypes, we learned that the overhead was small in practice and that through careful engineering it can be made negligible.

However, there appears to be limitations that are fundamental to the approach. While a reductionist strategy greatly facilitates system analysis, it is hard to assess the level of interaction between processes. Modern OSes provide mechanisms that allow one process to affect another process in ways that defeat independent analysis of the two processes. For example, a process can manipulate the computation of another process through the insertion of breakpoints followed by the manipulation of the registers or the stack of that process. As a consequence, in the presence of a malicious process, the usefulness of assigning events, such as file accesses, to specific processes or applications becomes limited. This explains why the file monitoring approach we proposed is best suited to scenarios where the monitored VMs are not intentionally malicious.

Even in a setting where monitored VMs are not intentionally trying to obfuscate their actions, many events cannot easily be made to trigger hardware traps. For example, file system data and meta data are often updated by the OS in memory before being written to disk. Depending on the file system in use, some updates will never be written to disk because subsequent updates eliminate the need to reflect the changes to persistent storage. A file could be created, written to, read and then deleted with no actual access to the physical disk. This makes it impossible to get a comprehensive picture of the interaction of a process with the file system solely by monitoring the disk interface. Another example of an event that is hard to monitor is inter-process communication. OSes offer many mechanisms that allow to processes to exchange information and while monitoring such exchanges would be useful, it is challenging to do so via architectural introspection. A more rigorous characterization of the limitations of architectural introspection is left as future work.

In the course of this work, we also learned that there is a fundamental tension between on the one hand VM introspection and on the other end obfuscation performed by monitored software. Software may use obfuscation to prevent the theft of intellectual property through reverse engineering, to enforce Digital Rights Management (DRM) policies, or to thwart malicious attempts to disable security mechanisms. For example, PatchGuard [25] is a technology introduced by Microsoft in 64-bit versions of Windows to detect kernel hooking. PatchGuard periodically modifies its memory location and code to make itself harder to detect and disable. This means that it is also challenging for an execution monitor to identify the PatchGuard code. In retrospect, this is not particularly surprising, since the goals of obfuscation and introspection are at odds. It is unclear whether reconciling these two goals is feasible.

Combining OS-specific mechanisms with architectural introspection to address the limitations discussed above is a promising avenue of future work. In settings where it is possible for the VMM to introduce code in the monitored VM, such code can monitor events that are not visible from the VMM and can also help reduce the performance overhead introduced by the monitoring. However, this code becomes vulnerable to manipulations by the monitored VM. OS-independent mechanisms that leverage architectural introspection could then be used to prevent tampering with the monitoring code introduced in the VM.

The presence of an executable permission bit in page table entries proved to be key to enabling a low-overhead, low-complexity approach to providing execution monitoring. In essence, this hardware mechanism forces the OS to make its intention explicit. As a result, it provides the layer below some visibility into the operations of the layer above. This means an opportunity for the layer below to monitor that the layer above is operating properly. It also means an opportunity to optimize the operations of the layer above using information that it may not have access to. This is an avenue that was explored in file system work in particular [6].

Exploring similar hardware mechanisms that could be introduced to facilitate introspection is another avenue of future work. Designing such mechanisms will not be easy, as making interfaces artificially restrictive has the potential to hamper innovation. Mechanisms that have multiple uses and whose use is optional, all features of the NX bit, certainly have the most potential to be successful. Fine-grained memory protection mechanisms as proposed by Witchel et al. with the Mondrian memory protection [99] are an example of an addition to existing hardware that would prove very useful to introspection systems.

Finally, one could incorporate AI techniques such as machine learning to infer more information about the monitored VM. Such techniques have been shown to hold promise [21] to interpret OS data structures. While these techniques are probabilistic and can often be defeated by motivated attackers, they could be used to infer aspects of the operations of the monitored VM that can then be deterministically confirmed, in a way similar to how our address inference algorithm functions.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 2005)*, November 2005.

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*, pages 2–13, October 2006.

[3] Amazon. Amazon web services customer agreement, 2006. `http://aws.amazon.com/agreement/` Last accessed: 4/2/2009.

[4] Austin Appleby. MurmurHash 2.0, 2006. `http://murmurhash.googlepages.com/`.

[5] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 1997.

[6] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Lakshmi N. Bairavasundaram, Timothy E. Denehy, Florentina I. Popovici, Vijayan Prabhakaran, and Muthian Sivathanu. Semantically-Smart Disk Systems: Past, Present, and Future. *Sigmetrics Performance Evaluation Review (PER)*, 33(4):29–35, March 2006.

[7] Kurniadi Asrigo, Lionel Litty, and David Lie. Using VMM-based sensors to monitor honeypots. In *Proceedings of the 2nd ACM/USENIX Conference on Virtual Execution Environments (VEE 06)*, June 2006.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, October 2003.

[9] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, and Adam Shostack. Timing the application of security patches for optimal uptime. In *Proceedings of the 15th Large Installation Systems Administration Conference (LISA)*, pages 233–242, November 2002.

[10] Doug Beck, Binh Vo, and Chad Verbowski. Detecting stealth software with Strider GhostBuster. In *International Conference on Dependable Systems and Networks (DSN)*, pages 368–377, April 2005.

[11] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference (FREENIX track)*, July 2005.

[12] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure software updates: Disappointments and new challenges. In *Proceedings of the 1st Workshop on Hot Topics in Security (HotSec 2006)*, July 2006.

[13] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9:131–152, 1996.

[14] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, May 2008.

[15] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[16] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, October 2008.

[17] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS 2001)*, May 2001.

[18] P. Chicoine, M. Hassner, M. Noblitt, G. Silvus, B. Weber, and E. Grochowski. Hard disk drive long data sector white paper. Technical report, The International Disk Drive Equipment and Materials Association (IDEMA), April 2007.

[19] Citrix XenClient, 2006. http://www.citrix.com/xenclient.

[20] Bryce Cogswell and Mark Russinovich. RootkitRevealer v1.71, November 2006.

[21] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 255–266, December 2008.

[22] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *Proceedings of the 17th USENIX Security Symposium*, pages 395–410, July 2008.

[23] James C. Dehnert, Brian K. Grant, John P. Banning, Richard J ohnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing^{TM}software: using specul ation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 15–24, March 2003.

[24] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *IEEE Computer Magazine*, pages 55–62, July 2003.

[25] Scott Field. An introduction to kernel patch protection, August 2006.

[26] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, October 2003.

[27] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Symposium on Network and Distributed System Security (NDSS 2003)*, February 2003.

[28] Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS 2005)*, May 2005.

[29] GoGrid. GoGrid cloud hosting: Acceptable use policy, 2006. http://www.gogrid.com/legal/aup.php Last accessed: 4/2/2009.

[30] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, June 1974.

[31] Kingshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262, 2000.

[32] David Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006.

[33] John L. Griffin, Trent Jaeger, Ronald Perez, Reiner Sailer, Leendert van Doorn, and Ramón Cáceres. Trusted virtual domains: Toward secure distributed services. In *Proceedings of the 1st Workshop on Hot Topics in System Dependability (HotDep 2005)*, June 2005.

[34] J. Alex Halderman and Edward W. Felten. Lessons from the Sony CD DRM episode. In *Proceedings of the 15th USENIX Security Symposium*, pages 77–92, August 2006.

[35] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection. In *Proceedings of the First European Conference on Systems (EuroSys)*, April 2006.

[36] IDC Exchange. IT cloud services user survey, pt.2: Top benefits & challenges, October 2008. http://blogs.idc.com/ie/?p210 Last accessed: 1/10/2009.

[37] Alexander Iliev and Sean W. Smith. Protecting user privacy via trusted computing at the server. *IEEE Security and Privacy*, 3(2):20–28, March 2005.

[38] Trent Jaeger, Reiner Sailer, and Umesh Shankar. Prima: Policy-reduced integrity measurement architecture. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT 06)*, June 2006.

[39] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 128–138, October 2007.

[40] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the 2006 Annual Usenix Technical Conference*, June 2006.

[41] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*, pages 14–24, October 2006.

[42] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the 4th*

ACM/USENIX Conference on Virtual Execution Environments (VEE 08), pages 91–100, March 2008.

[43] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, October 2005.

[44] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[45] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating system support for virtual machines. In *Proceedings of the 2003 Annual Usenix Technical Conference*, June 2003.

[46] Brian Krebs. Amazon: Hey spammers, get off my cloud!, July 2008. http://blog.washingtonpost.com/securityfix/2008/07/amazon_hey_spammers_get_off_my.html Last accessed: 1/10/2009.

[47] Lionel Litty. Hypervisor-based intrusion detection. Master's thesis, Department of Computer Science, University of Toronto, 2005.

[48] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th USENIX Security Symposium*, pages 243–258, July 2008.

[49] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Computer meteorology: Monitoring compute clouds. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS 2009)*, May 2009.

[50] Lionel Litty and David Lie. Manitou: A layer-below approach to fighting malware. In *Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 6–11, October 2006.

[51] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, April 2008.

[52] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. How low can you go? Recommendations for hardware-supported minimal TCB code execution. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*, March 2008.

[53] Microsoft. Visual Studio, Microsoft Portable Executable and Common O bject File Format specification, May 2006. Rev. 8.0.

[54] Grzegorz Miłoś, Derek G. Murray, Steven Hand, and Michael Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 Annual Usenix Technical Conference*, July 2009.

[55] F. Monrose, P. Wyckoff, and A. Rubin. Distributed execution with remote audit. In *Proceedings of the 6th Annual Symposium on Network and Distributed System Security (NDSS 1999)*, pages 103–113, February 1999.

[56] Nessus, Tenable Network Security, 2006. http://www.nessus.org.

[57] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Annual Symposium on Network and Distributed System Security (NDSS 2005)*, February 2005.

[58] NIST. National software reference library, 2006. http://www.nsrl.nist.gov/.

[59] Peter Nowak. Internet security moving toward "white list", September 2007. Available at http://www.cbc.ca/news/background/tech/privacy/white-list.html.

[60] M. Oberhumer, L. Molnár, and J. Reiser, 2006. http://upx.sourceforge.net/.

[61] PaX, 2006. http://pax.grsecurity.net/.

[62] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[63] Bryan D. Payne, Martim Cabrone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *23rd Annual Computer Security Applications Conference (ACSAC 2007)*, December 2007.

[64] Bryan D. Payne, Martim Cabrone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, May 2008.

[65] Nick L. Petroni and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 103–115, October 2007.

[66] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[67] Nick L. Petroni, Jr., Timothy Fraser, Aaron Walters, and William A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th USENIX Security Symposium*, August 2006.

[68] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third-generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[69] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An analysis of conficker's logic and rendezvous points. Technical report, SRI International, March 2009.

[70] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th IEEE/ACM International Symposium on Micro-architecture (Micro 2006)*, December 2006.

[71] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, pages 199–212, November 2009.

[72] John Scott Robin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[73] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing mal ware. In *22nd Annual Computer Security Applications Conference (ACSAC 2006)*, pages 289–300, December 2006.

[74] Mark E. Russinovich. Process Explorer, 2007. http://technet.microsoft.com/sysinternals/bb896653.aspx.

[75] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000.* Microsoft Press, 2005.

[76] Ahmad-Reza Sadeghi and Christian Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *Proceedings of the 2004 Workshop on New Security Paradigms (NSPW 2004)*, September 2004.

[77] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramón Cáceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a MAC-based security architecture for the xen open-source hypervisor. In *21st Annual Computer Security Applications Conference (ACSAC 2005)*, December 2005.

[78] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS 2004)*, October 2004.

[79] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[80] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA 2003)*, pages 81–94, October 2003.

[81] Andreas Schuster. Reconstructing a binary, April 2006. `http://computer.forensikblog.de/en/2006/04/reconstructing_a_binary.html`.

[82] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, October 2007.

[83] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, pages 1–16, October 2005.

[84] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, pages 552–561, October 2007.

[85] Umesh Shankar, Trent Jaeger, and Reiner Sailer. Toward automated information-flow integrity for security-critical applications. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006.

[86] Elaine Shi, Adrian Perrig, and Leendert van Doorn. BIND: A fine-grained attestation service for secure distributed systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[87] James Edward Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann Publishers, 2005.

[88] Snort, 2006. http://www.snort.org/.

[89] G. E. Suh, J-W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004)*, October 2004.

[90] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. AutoISES: Automatically inferring security specifications and detecting violations. In *Proceedings of the 17th USENIX Security Symposium*, pages 379–394, July 2008.

[91] The Trusted Computing Group, 2006. https://www.trustedcomputinggroup.org/home.

[92] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Form at (ELF) specification, May 1995. V1.2.

[93] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.

[94] Paul C. van Oorschot, Anil Somayaji, and Glenn Wurster. Hardware-assisted circumvention of self-hashing software tamper resista nce. *IEEE Transactions on Dependable and Secure Computing*, 2(2):82–92, Apr.-June 2005.

[95] VersionTracker, 2006. http://versiontracker.com/.

[96] Saman Amarasinghe Vladimir Kiriansky, Derek Bruening. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[97] VMWare ACE, 2006. http://www.vmware.com/products/ace.

[98] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 195–210, December 2002.

[99] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002)*, pages 304–316, October 2002.

[100] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[101] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th ACM SIGOPS European Workshop (EW 2002)*, September 2002.