Hypervisor Support for Identifying Covertly Executing Binaries

Lionel Litty H. Andrés Lagar-Cavilla
Dept. of Computer Science
University of Toronto
{llitty,andreslc}@cs.toronto.edu

David Lie
Dept. of Elec. and Comp. Eng.
University of Toronto
lie@eecg.toronto.edu

Abstract

Hypervisors have been proposed as a security tool to defend against malware that subverts the OS kernel. However, hypervisors must deal with the *semantic gap* between the low-level information available to them and the high-level OS abstractions they need for analysis. To bridge this gap, systems have proposed making assumptions derived from the kernel source code or symbol information. Unfortunately, this information is *non-binding* – rootkits are not bound to uphold these assumptions and can escape detection by breaking them.

In this paper, we introduce *Patagonix*, a hypervisor-based system that detects and identifies covertly executing binaries without making assumptions about the OS kernel. Instead, Patagonix depends only on the processor hardware to detect code execution and on the binary format specifications of executables to identify code and verify code modifications. With this, Patagonix can provide trustworthy information about the binaries running on a system, as well as detect when a rootkit is hiding or tampering with executing code.

We have implemented a Patagonix prototype on the Xen 3.0.3 hypervisor. Because Patagonix makes no assumptions about the OS kernel, it can identify code from application and kernel binaries on both Linux and Windows XP. Patagonix introduces less than 3% overhead on most applications.

1 Introduction

Malicious software, otherwise known as *malware*, continues to be a serious problem in today's computing environment. Malware is becoming increasingly difficult to detect and remove because it commonly comes bundled with a *rootkit* [12], which abuses administrative privileges to hide the execution of malware binaries and their resource usage from the system administrator. Rootkits accomplish this by attacking the administrator's ability to obtain information about a system. For example, rootkits will subvert execution-reporting utilities, such as `ps` and `lsmod` on Linux systems and the `task manager` and `Process Explorer` [27] on Windows, which administrators rely on to query the operating system (OS) about running binaries and kernel modules. Rootkits may also subvert the OS kernel itself so that any queries to the kernel will receive a response that has been appropriately distorted by the rootkit. In this way, rootkits have been able to elude even the most experienced system administrators and sophisticated malware detection tools [11]. Even if the rootkit's presence is discovered, it is difficult to determine whether an attempted removal is successful or not, as the rootkit's ability to hide executing code enables it to trick the administrator into believing that it has been removed. As a result, best practice states that when a rootkit is even suspected to be present, the administrator must re-install the entire system from scratch to be sure that the rootkit is removed – a costly and undesirable solution. Trustworthy execution-reporting utilities, which would enable a system to detect hidden malware processes and determine if an attempted removal was successful or not, would save administrators a great deal of effort and reduce system downtime.

In this paper, we present Patagonix, a system that denies rootkits the ability to hide executing binaries from the system administrator. Patagonix does this by addressing two shortcomings of current execution-reporting utilities. First, these utilities all depend on the integrity of the kernel, both as a source of information and for protection against tampering. However, since rootkits can 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 [17], may detect tampering of on disk binaries,

but will miss tampering of binaries once they are loaded in memory.

Unlike existing execution-reporting utilities, Patagonix does not depend on the OS. Instead, Patagonix uses a hypervisor, allowing it to retain its integrity even if the rootkit has compromised the OS kernel. The challenge to implementing an execution-reporting utility in a hypervisor is the *semantic gap* [6] between the information available to the hypervisor and the actual state of the system. Other work has bridged this gap by using and trusting information about the OS kernel, such as the kernel source code or kernel symbol information [3, 10, 13, 23, 25]. However, such information cannot be trusted because it is *non-binding* – the rootkit is not bound to maintain the semantics implied by source and symbol information, allowing it to escape detection. For example, if the hypervisor uses non-binding information about the format or location of kernel data structures, the rootkit may evade detection by adding fields to the data structures or moving the data structures to a memory location that is not being monitored. Similarly, assumptions about the code structure of the kernel can be exploited by a rootkit that modifies OS kernel execution to avoid code paths monitored by the hypervisor. Patagonix does not rely on any non-binding information about the OS kernel and relies only on the behavior of the hardware, which cannot be altered by malware.

Patagonix also verifies the integrity of all executing binaries before giving their identity to the administrator. Rather than verifying the contents of binaries on disk, Patagonix inspects the code as it executes in memory. As a result, Patagonix cannot be fooled by rootkits that avoid tampering with files on disk by injecting malicious code into binaries as they run. On the other hand, systems make modifications to code at run-time, causing it to differ from its image on disk when it is executed. Patagonix can differentiate legitimate modifications from malicious ones. The executing code is identified using a trusted external database that contains cryptographic hashes of binaries, such as the National Software Reference Library (NSRL) [20].

In this paper 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 [4], 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 OSs 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 and Evaluation.** 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 [10]. We tested Patagonix on 9 rootkits and found that it was able to identify code hidden by every one of them. In addition, our Patagonix prototype introduces less than 3% performance overhead on most applications.

We do not claim that Patagonix can detect all rootkits since Patagonix focuses on detecting covertly executing binaries – a rootkit that does not hide executing binaries, but only hides files and network connections, would not be detected. Fortunately, techniques to detect such rootkits, which do not depend on non-binding information, already exist. For example, using direct access to a raw disk image can detect hidden files [13] and a network-based intrusion detection system can detect hidden network connections. However, to the best of our knowledge, all techniques to detect hidden processes depend on non-binding information, making Patagonix useful in those circumstances.

In Section 2, we describe the problem with trusting non-binding information, the assumptions that Patagonix relies on, and the guarantees and limitations it has. Section 3 gives an overview of the Patagonix architecture, while Sections 4 and 5 detail our identity oracles and our prototype implementation. In Section 6 we describe the two usage modes of Patagonix: reporting and lie detection. Section 7 evaluates Patagonix's effective-

ness at detecting covert processes and performance overhead. Section 8 discusses related work and we conclude in Section 9.

2 Security Model

2.1 Problem Description

Systems that monitor OS-level events from a hypervisor must wrestle with the semantic gap between the state of the OS and the information available to the hypervisor. Previous systems have bridged this gap using non-binding information derived from source code and symbol information, but acknowledge that in doing so they make themselves vulnerable to a rootkit that is aware of their monitoring technique [3, 10, 13, 23, 25]. For instance, if the hypervisor monitors the system call table by using location information derived from non-binding sources, the rootkit can evade detection by altering the kernel's system call dispatch handler to use a table placed at a different location, and filled with pointers to malicious system call handlers. The hypervisor-based monitor would continue to monitor the original, unchanged system call table, which is no longer being used by the kernel. Unfortunately, preventing this attack by simply disallowing modification of kernel code will cause false positives because kernels employ self-modifying code. Manipulating the dispatch handler is only one example; similar assumptions based on non-binding information about data types or function entry-points are equally prone to subversion. More sophisticated techniques take a systematic approach to analyzing the Linux kernel memory state for tampering by malware, but they require ad hoc rules written with expert knowledge [24] or source code annotations that provide only partial protection [25]. Further, all the aforementioned approaches use a sampling approach, creating a window of vulnerability that may be exploited by malware to remain undetected.

Patagonix securely addresses the semantic gap problem by avoiding reliance on non-binding information. Rather it relies only on information from the processor hardware about pages containing executing code. In addition, Patagonix detects and validates run-time code modification and ensures that they conform to the modifications permitted in the binary format specification. Finally, by utilizing the processor MMU hardware, Patagonix provides continuous monitoring and detection with very little overhead.

2.2 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 uses the NSRL [20]. 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 [11]. 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.

2.3 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 [32]. 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 JITs 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, it will ignore pages it observes executing in the JIT address space that are not present in the trusted database (JITs 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 and escape any sandboxing enforced by the JIT, Patagonix's guarantees still hold because the rootkit will not be able to hide the execution of the JIT, nor can the rootkit cause Patagonix to misidentify the JIT 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.

3 System Architecture

3.1 Overview

The architecture of Patagonix is illustrated in Figure 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 7, 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 Patagonix 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 OS-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 OS agnostic.

3.2 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

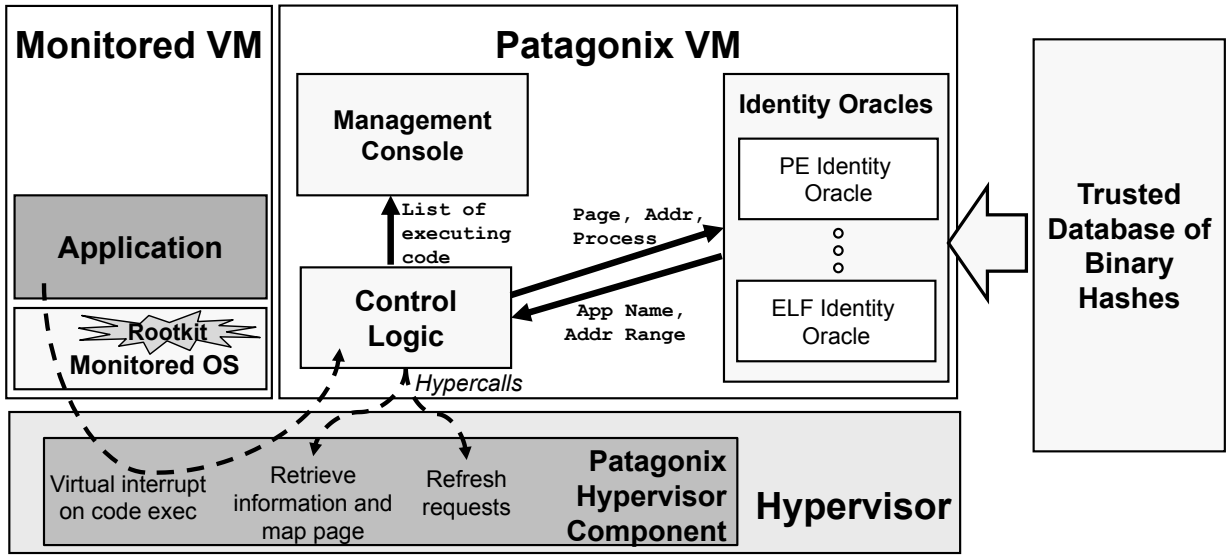

Figure 1: The Patagonix architecture.

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 5.2.

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 [14] 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 se-

curely 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.

3.3 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

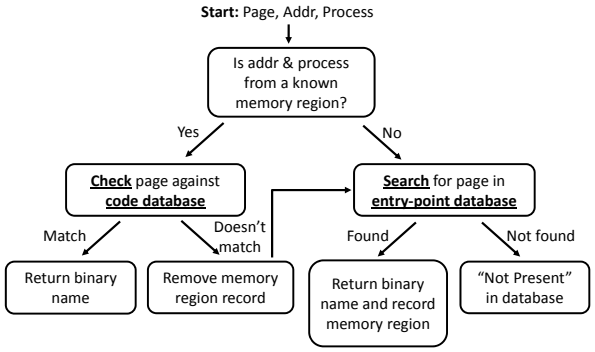

Figure 2: Identity Oracle framework. The functions and databases that are loader specific have been underlined.

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 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 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.

4 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.

4.1 Application Binary Oracles

ELF Oracle. The Executable and Linkable Format (ELF) [33] is used by Linux, as well as other OSs 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 SHA-256 is collision-resistant and difficult to invert, any tampering of the binary will result in the binary being identified as not present.

PE Oracle. The Portable Executable (PE) format [19] 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.

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.

4.2 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 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 [28].

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.

4.3 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 [21]. To support additional packers, Patagonix only needs to be provided with an unpacker. For example, Patagonix could use PolyUnpack [26] to automatically support a large number of executable packers.

Finally, we observed two non-JIT binaries 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 JITs – we can identify that they are executing, but do not examine other code pages in their address space.

5 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.

5.1 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.

5.2 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 [35] or Transmeta's Crusoe processor [8]. 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 [34]. 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, this functionality 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.

5.3 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 because 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.

6 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

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 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.

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 [10,13]. 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.

7 Evaluation

We evaluate two aspects of Patagonix: its effectiveness at detecting and identifying hidden processes and rootkits and the performance overheads introduced by adding Patagonix to the hypervisor.

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.

7.1 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 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 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 OSs. 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 3. 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.

7.2 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. We instrumented our prototype to break down the page identification process

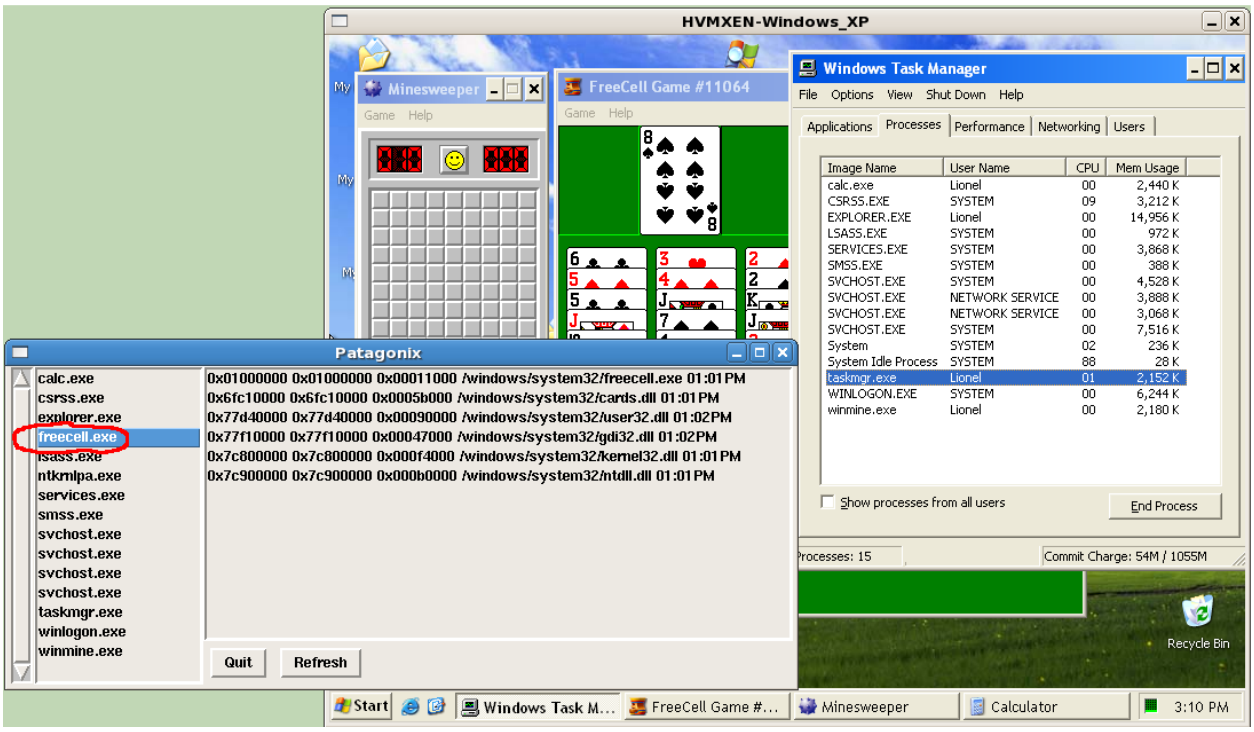

Figure 3: Output of both Patagonix and the Task Manager when the FU rootkit is used to hide `freecell.exe`. Patagonix identifies all processes including `freecell.exe`, while the Task Manager does not display the hidden process. Patagonix identifies "System" as `ntkrnlpa.exe`, the name of the Windows XP kernel binary.

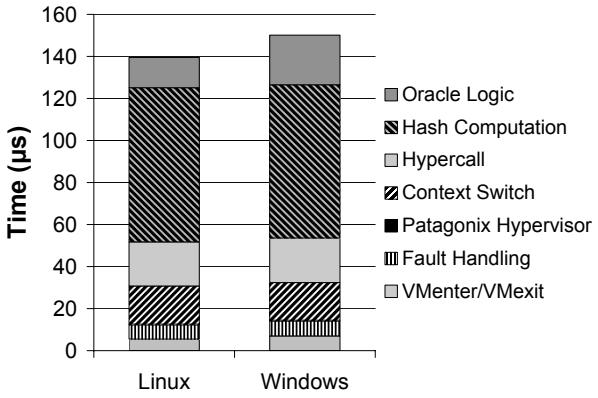

Figure 4: Execution time for various components of the identification operation. The total height of the bars represents the average time required to identify the origin of an executing code page.

into its different components. Figure 4 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.

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 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.

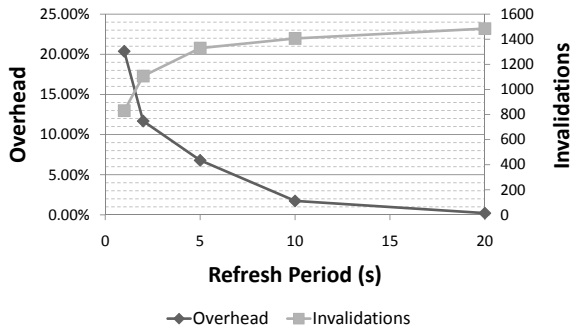

Figure 5: Overhead and Invalidations vs. Refresh Period. Apache Build on Linux. Averages of five runs with standard deviations below 2% of the average.

7.3 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. 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 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 WinXP 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.

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 5 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.

8 Related Work

The problems associated with the semantic gap between the hypervisor and guest VMs were first identified in a seminal paper by Chen and Noble [6]. Since then, there have been several attempts to bridge this gap using non-binding information derived from source code and symbol information. For example, Livewire [10], Copilot [23] and SBCFI [25] rely on symbol information in kernel binary or `System.map` file, while Asrigo et al. [3] and VMWatcher [13] rely on information derived from kernel source code. Because they make assumptions based on non-binding information, they are all prone to evasion by a rootkit that breaks those assumptions. Patagonix does not rely on any non-binding information.

The principle of lie detection – comparing two views of the same data for discrepancies – has been used in the literature. For example, Rootkit Revealer [7] and Strider GhostBuster [5] compare high-level and low-level views of the same system information. However, since both views are still derived from within the infected system, a thorough rootkit can make both high-level and low-level views agree, thus eluding these systems. Like Patagonix, other systems compare views taken from both within (i.e. in-the-box) the infected system, and outside (out-of-the-box) the infected systems. For example, both Livewire [10] and VMWatcher [13] compare views of executing processes derived from the VMM with those gathered from within the monitored system. However, unlike Patagonix, these systems do not deal with asyn-

chrony between the measurement times of the in-the-box and out-of-the-box views and will thus suffer from false positives. Lycosid [15] also does lie detection by counting the number of address spaces in a VM. However, because Lycosid does not identify which binaries the processes are executing and the hypervisor's measurements contain noise, it 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.

Like Patagonix, remote attestation systems also must identify and report executing binaries on a system. In addition, they may also report the integrity of the data in a system, and are often used to report this information to a remote party instead of the system administrator. However, these systems in general assume a weaker attack model since they in general rely on the integrity of the OS. For example, IMA [29], implements such functionality directly in the OS kernel, and thus depends on the integrity of the OS kernel to report correct results. An alternative is Terra [9] which performs attestation in a hypervisor. Terra attests the identity of the virtual disk used to initialize a "closed box" to a remote party. Closed boxes are VMs that are fully managed by a third party and usually cannot be extended in any significant way. Since Patagonix allows the monitored OS to be arbitrarily extended as long as the hashes of any new legitimate code are in the trusted database. A combination of Patagonix and Terra's abilities could enable support for attestation of open, extensible systems as well as individual programs executing in these systems.

Hypervisors have long been used as a means for implementing a secure trusted computing base, with which untrusted OS images could be made secure [16, 31]. While our prototype was implemented in the Xen hypervisor [4], the functionality required from the hypervisor is generic enough to allow Patagonix to be implemented on any virtualization system. To explore this point, we have obtained a source code license for VMware Workstation and are currently working on a port of Patagonix. We have found that VMware-specific functionality, such as its page table entry caching [2] and dynamic code translation [1], have not impeded the necessary functionality from being added.

Finally, Patagonix uses or extends ideas presented in other work. Patagonix is based on our earlier work called Manitou, which also uses hashes to identify running applications from a hypervisor [18]. However, Manitou is only able to identify applications for Linux guest OSs, making its treatment of the problem overly simplistic. It also does not perform synchronous lie detection. Independent to our work and using a similar low-level mechanism to detect code execution, SecVisor [31] 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 OSs as well as the applications they execute, and demonstrates that these guarantees can be obtained by small extensions to a general-purpose hypervisor. Other projects have manipulated the page tables used by the X86 MMU. For example, the PaX project [22] 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 [30] have demonstrated that PE binaries can be reconstructed by analyzing memory dumps. The PE identity oracle described in this paper uses similar techniques to identify binaries online.

9 Conclusions

Current OSs are vulnerable to subversion by rootkit and thus cannot be relied upon to provide trustworthy information about what code is executing on a system. Patagonix solves this problem by using the processor MMU to detect executing code from a hypervisor. It then uses identity oracles, which leverage information from the binary format specifications and loaders to identify the executing code. In this way, Patagonix is able to bridge the semantic gap between the hypervisor and the OS without having to trust non-binding information, which is vulnerable to subversion by the rootkit. We have found that binary formats across different OSs have similarities, enabling the creation of a universal oracle construction framework and the use of common techniques across various binary formats. Aside from the binary-specific formats, the Patagonix framework does not use any information about the OS, allowing the same framework to be used on diverse OSs such as Windows XP, Linux 2.4 and Linux 2.6, without any modification. Through the combined use of writable and non-executable page table bits, Patagonix is only invoked when code is executed for the first time, and as a result, has a modest performance overhead of less than 3% on most applications.

Acknowledgements

This paper was greatly improved by comments from Tom Hart, Ian Sin, Jesse Pool, Lee Chew, James Huang, Stan Kvasov, Niraj Tolia and Ashvin Goel. We would also like to thank the anonymous reviewers for their helpful suggestions. This work was supported in part by an NSERC Discovery Grant and a MITACS Grant.

References

[1] K. Adams and O. 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)*, pages 2–13, Oct. 2006.

[2] O. Agesen and P. Subrahmanyam. Method and system for performing virtual to physical address translations in a virtual machine. U.S. Patent 7069413, Dec. 2006.

[3] K. Asrigo, L. Litty, and D. Lie. Using VMM-based sensors to monitor honeypots. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, pages 13–23, June 2006.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, Oct. 2003.

[5] D. Beck, B. Vo, and C. Verbowski. Detecting stealth software with Strider GhostBuster. In *International Conference on Dependable Systems and Networks (DSN)*, pages 368–377, Apr. 2005.

[6] P. M. Chen and B. D. Noble. When virtual is better than real. In *8th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, May 2001.

[7] B. Cogswell and M. Russinovich. RootkitRevealer v1.71, Nov. 2006.

[8] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing™software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 15–24, Mar. 2003.

[9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206, Oct. 2003.

[10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Symposium on Network and Distributed System Security (NDSS)*, pages 191–206, Feb. 2003.

[11] J. A. Halderman and E. W. Felten. Lessons from the Sony CD DRM episode. In *Proceedings of the 15th USENIX Security Symposium*, pages 77–92, Aug. 2006.

[12] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison Wesley, 2005.

[13] X. Jiang, X. Wang, and D. 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)*, pages 128–138, Oct. 2007.

[14] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the 2006 Annual Usenix Technical Conference*, pages 1–14, May 2006.

[15] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE)*, pages 91–100, Mar. 2008.

[16] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A VMM security kernel for the VAX architecture. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 2–19, 1990.

[17] G. H. Kim and E. 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.

[18] L. Litty and D. Lie. Manitou: A layer-below approach to fighting malware. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 6–11, Oct. 2006.

[19] Microsoft. Visual Studio, Microsoft Portable Executable and Common Object File Format specification, May 2006. Rev. 8.0.

[20] NIST. National software reference library, 2008. `http://www.nsrl.nist.gov/`.

[21] M. Oberhumer, L. Molnár, and J. Reiser, 2008. `http://upx.sourceforge.net/`.

[22] PaX, 2008. `http://pax.grsecurity.net/`.

[23] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot–a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, Aug. 2004.

[24] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th USENIX Security Symposium*, pages 289–304, July 2006.

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

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

[27] M. Russinovich. Process Explorer, 2007. `http://technet.microsoft.com/sysinternals/bb896653.aspx`.

[28] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, 2005.

[29] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223–238, Aug. 2004.

[30] A. Schuster. Reconstructing a binary, Apr. 2006. `http://computer.forensikblog.de/en/2006/04/reconstructing_a_binary.html`.

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

[32] H. 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)*, pages 552–61, Oct. 2007.

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

[34] P. C. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, 2(2):82–92, Apr.-June 2005.

[35] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 304–316, Oct. 2002.