Using Hypervisors to Secure Commodity Operating Systems

David Lie
University of Toronto

Lionel Litty*
VMware Inc.

ABSTRACT

Hypervisors are an excellent tool for increasing the security of commodity software against attack. In this paper, we discuss some of the lessons and insights we gained from designing and implementing four research prototypes that use hypervisors to secure commodity systems. We also compare our findings with other approaches to implementing security in a hypervisor.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive Software,Security kernels

General Terms

Security

Keywords

Hypervisors,Isolation,Integrity,Proxos,Patagonix

1. INTRODUCTION

Traditionally, the responsibility for enforcing system security has largely fallen on shoulders of the operating system (OS). For example, it is the responsibility of OS code to enforce process isolation and access control between principals and objects. Unfortunately, commodity OSs are large and complex, and because of this, they contain a large number of flaws that make them vulnerable to compromise. Once compromised, an attacker can violate all the security guarantees that the OS is supposed to enforce, giving her unfettered to access to the system. As a result, systems that require a higher amount of security and assurance have traditionally turned to hardware-enforced security to gain an additional level of protection should the measures in the OS fail. For example, a variety of software copyright and integrity solutions use "hardware dongles", which provide security in the form of a token, since the ability to copy hardware components is independent of the integrity of the software on the system. Highly secure systems may also use a physically secure co-processor card such as the IBM 4758, which contains a simple but complete computer system encased in a tamper-resistant package [16]. Recent Intel TXT, AMD SVM and TCG TPM functionality incorporated into commodity PCs provides additional hardware security to protect systems in the event of a failure of OS integrity to an attack. This recent trend in commodity hardware illustrate that the main threat to systems today are OS vulnerabilities, since they assume physical security against an attacker, and are designed only to protect the system in the event of a breach of security of the software in the OS.

However, hardware support for security has some disadvantages. First, implementation in hardware is generally more expensive and difficult that in software. As a result, the functionality is often restrictive and simple, thus constraining its power and effectiveness. Overly complex functionality requires more silicon area, which if only rarely used, does not justify its cost. Second, once functionality is defined in hardware, it is inflexible and difficult to change. Any bugs or vulnerabilities in a hardware design generally have fairly severe consequences and may require a software work-around. If a software workaround is not possible or weakens the security, a product recall is required [1]. In addition, since hardware cannot be changed after it is deployed, the functionality must be general and OS-agnostic, since hardware designers cannot predict what changes will occur in the OS after the hardware is deployed.

A hypervisor is a layer of software below the OS that runs at a higher privilege level than the OS. Because of the higher privilege level, the integrity of the hypervisor remains intact even if the OS is compromised. As a result, an alternative to implementing security in hardware is to implement security functionality in a hypervisor. On one hand, a hypervisor is not as secure as hardware against an adversary who has physical access to the system because she may tamper with the boot process to remove the hypervisor. However, in cases where physical access is not granted to the adversary, the security properties of a hypervisor and hardware are equivalent. On the other hand, a hypervisor offers significant advantages over hardware for implementing low-level security. Because it is in software, the security functionality can be arbitrarily complex and may even be specifically tailored to the guest OSs it is supporting. Flaws found in the security system or hypervisor can be patched just like any other software. Finally, implementation in software allows such functionality to be highly configurable and customizable, allowing end users to specify policies that suit their needs.

We have taken two broad approaches to improving the security of commodity OSs using hypervisors. In the first approach,

*The majority of this work was done while Lionel Litty was at the University of Toronto

[1]There have been documented cases of BIOS bugs causing some machines to become unbootable after TXT is used. One of the authors has experienced this personally.

STC'10, October 4, 2010, Chicago, Illinois, USA.

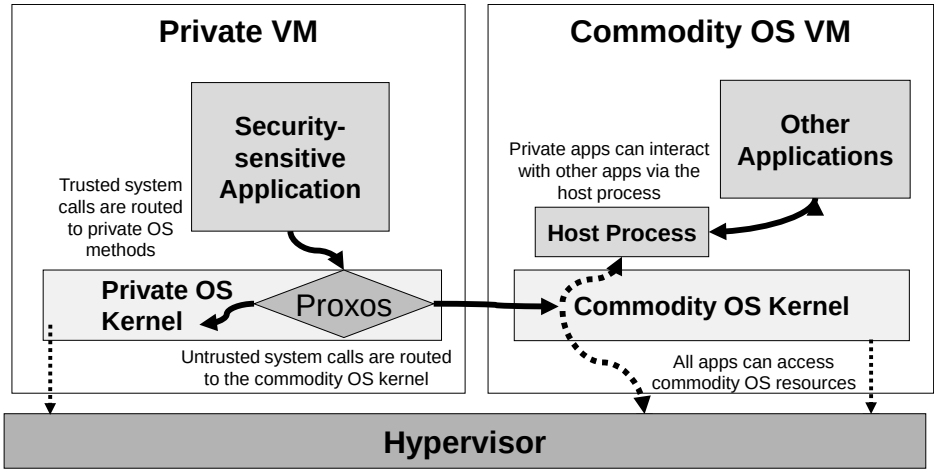

Figure 1: The Proxos Architecture

we implement secure execution for applications using a system called *Proxos*. Proxos allows applications to control how much trust they must place in the OS, thus limiting the damage an attacker who compromises the OS can cause to applications and their data. While OSs typically consist of millions of lines of code, Proxos allows applications to minimize how much that code base contributes to the application's trusted computing base (TCB). In our second approach, we use hypervisor introspection to attempt to detect and mitigate flaws in commodity OSs. The advantage of using a hypervisor is that the detection system is immune to tampering by the OS, and will continue to function correctly even if the attacker controls the OS.

This paper is organized as follows. We begin by giving some background on hypervisors in Section 2. We then describe systems we have built implementing our two approaches to hypervisors support for security in Section 3 and Section 4. We discuss lessons and insights we gained from our experience in Section 5 and conclude in Section 6.

2. BACKGROUND

A hypervisor is a layer of software that interposes between the OS and the hardware. As a result, it runs at a privilege level higher than that of the OS. Often, hypervisors use their privileges to virtualize the underlying hardware and give the illusion of multiple identical hardware instances. When used in this way, hypervisors are also sometimes referred to as virtual machine monitors, as they can export multiple virtual machine objects that are used to run multiple OS images simultaneously. We will refer to an OS image running on the hypervisor as a *Guest OS*. For the purposes of this paper, whether the hypervisor is used to support multiple or just one OS image is not important – the points made in this paper apply equally to cases where there are multiple guest OSs running concurrently and to cases where there is just one guest OS.

Aside from exporting multiple virtual copies of the underlying hardware, hypervisors can also be used to modify the behavior of the underlying hardware from the point of view of the guest OS. For example, a hypervisor can be used to emulate a different instruction set, or emulate hardware functionality that does not actually exist in software. Hypervisors provide three important properties that make security hardware that is emulated in the hypervisor as secure against an attacker who has compromised the OS as real security hardware. First, hypervisors run in a separate protection domain from the guest OS. Thus all security functionality in the hypervisor is strongly isolated from any vulnerabilities or bugs in the guest OS. Second, the interface between the hypervisor and the guest OS is very narrow. It is roughly equivalent to the interface between a regular OS and the underlying hardware. For example, the Xen hypervisor exposes only four types of virtual hardware primitives – privileged instructions, devices, interrupts and page tables. In comparison, Linux exports more than 300 or so system calls to processes. This narrow interface gives an attacker who compromises a guest OS fewer opportunities to compromise the hypervisor than an attacker who compromises a process in a traditional OS. Finally, hypervisors are generally small and simple because their job is to emulate hardware, which in itself is generally simpler than software. Most complex policy and resource allocation decisions are implemented in software by the OS, leaving only simple operations that must either be fast or isolated from the OS to be implemented below in the hypervisor or in the hardware.

3. SECURE EXECUTION WITH PROXOS

For various, usually non-technical reasons, commodity OSs have all too often emphasized utility over security. As a result, they export broad and powerful interfaces to user-level processes, giving application programmers a large amount of freedom. Unfortunately, this power and freedom come at the cost of security. While OSs generally restrict the most sensitive capabilities to "privileged" applications, the fact remains that those capabilities are so useful and powerful that many applications end up being privileged. Researchers have recognized this problem and proposed some solutions, such as privilege separation [2, 14] and fine-grain access control [11]. However, these solutions either require non-trivial application porting or very complex policies to effectively limit privileges, so while effective, they have not experienced wide-spread applicability.

In commodity OSs, compromising a privileged process gives an attacker the ability to add code to a running kernel in the form of drivers or modules, to start or disable other processes, or to inspect and change the state of any process or file. As a result, even though this privileged code is not actually in the kernel, it is still in the TCB of the system because its privileged status make it as powerful as code in the OS kernel. Thus, to compromise an application,

the attacker need not find a vulnerability in the application or the systems code it uses – she need only to find a vulnerability in any privileged process running on the same system.

Unfortunately, because of the many capabilities given to privileged processes, a great deal of useful code often does run with full privileges in a commodity OS. For example, services that manage devices, fix file system corruption and perform backups run with full privileges. In addition, many network services, such as network file systems, http, e-mail, and remote login services have some component that require full privileges. Because of this, application developers find it beneficial to use these services to speed up application development and simplify their tasks. On an unvirtualized machine, there can only be one OS image, forcing unrelated applications to share a single OS image. An unfortunate result of this is that the TCB of an application becomes the OS kernel and the union of all other privileged code required by any application on the same OS. For a security-sensitive application, privileged code that is required by an application that is completely unrelated to it becomes part of the security-sensitive application's TCB.

The goal of Proxos is to allow an application to reduce the trust it has in this large TCB to the bare set of functionality it needs from the OS. The architecture of Proxos is illustrated in Figure 1. Rather than run the security-sensitive application in the same OS as other privileged processes, each security-sensitive application is run in its own "private" VM with its own private OS, which has only the bare minimum functionality required to execute the application. All binaries and sensitive data required to execute the application are placed in this private VM, while other applications remain on the "commodity OS". In this way, unrelated code is removed from the TCB of the security-sensitive application. The security-sensitive application has the option of forwarding certain system calls to be executed in the commodity OS by a "host process". We will discuss the reasons for doing this below.

However, an application running on its own in a VM cannot interact with other processes normally found in a commodity OS, making it less useful. For example, one of the applications we chose to protect in our prototype is the *ssh* remote login service. By placing the ssh server in its own VM, we protect important information such as the user passwords and cryptographic keys used by the ssh server from the underlying OS if it gets compromised. However, the ssh server would not be very useful to a remote user since once they log into the private VM, there would be no services or applications to access. While these other applications and services need not be in the TCB of the ssh server, the ability to interact with them is still important to the user of the ssh server. Proxos protects the security-sensitive application and its data while at the same time allowing the application to interact with other commodity applications by having the security-sensitive application declare the interface it can safely export to other applications through a set of system call routing rules. In this way, rather than giving unlimited access to privileged code in the commodity OS, only the minimum interface that is necessary for the utility of the application is exposed.

To illustrate, we compare a the ssh server running in a commodity OS with that of an ssh server protected by Proxos in Figure 2. In the commodity OS, all operations are performed on the single commodity Linux OS. However, with Proxos, when the ssh server opens and reads sensitive data like the password file, Proxos routes these system calls to the private OS where that sensitive data is stored. System calls that require interaction with the commodity OS, such as those used to spawn a shell and establish input and output pipes with the shell, are forwarded by Proxos to the commodity OS, where it is executed by the ssh host process. Since the

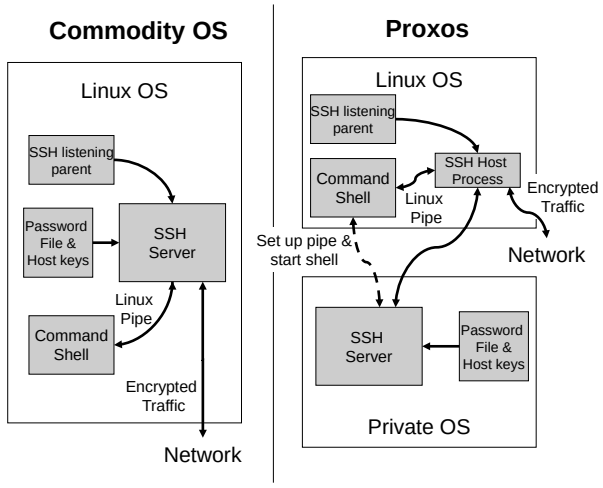

Figure 2: Protecting SSH using Proxos

shell is spawned on the commodity OS, it gives remote users the illusion that the ssh server is actually running on the commodity OS, even though the ssh server and all sensitive data is protected in the private VM and running on a private OS. An attacker who controls the Linux OS would only be able to affect the ssh server through the limited attack surface presented by the pipes between it and the command shell it spawned on the commodity OS. To access the password file or host keys, the attacker would have to find another vulnerability in the ssh server itself and exploit it via these pipes. In contrast, an attacker who compromises the Linux OS in the commodity OS side would have full access to the password file and host keys by virtue of having root privileges.

Other systems have also tried to protect applications in the face of a hostile or compromised OS. For example, XOM [8], Aegis [17] and SP [7] all use processor support to protect applications while executing. By encrypting and signing data stored in memory, the untrusted OS is allowed to manage the resources used by these applications, but cannot view or tamper with the data without violating the guarantees provided by cryptographic protections. Like Proxos, Overshadow [4] also uses a hypervisor to protect applications from a compromised OS. However, unlike Proxos, which places the protected application in a separate VM, Overshadow takes an approach much more similar to the processor-based systems mentioned earlier. Whenever the security-sensitive application is executing, Overshadow ensures that the memory pages belonging to the application are not mapped into any other address space that doesn't belong to the application. If a context other than the security-sensitive application executes, such as the OS kernel or another application, then Overshadow unmaps the application's pages from its address space. If the other context attempts to access pages belonging to the security-sensitive application, Overshadow encrypts and signs the content of the pages and then permits access. As a result, the security-sensitive application and commodity OS execute in the same VM, but the data and execution context of the application are protected from compromise through cryptographic means, just like in the processor hardware-based proposals listed above.

The different designs of Proxos and Overshadow present some interesting trade-offs. Despite the different approaches, both systems are in reality quite similar. Because the OS kernel cannot directly access the address space of the security-sensitive applica-

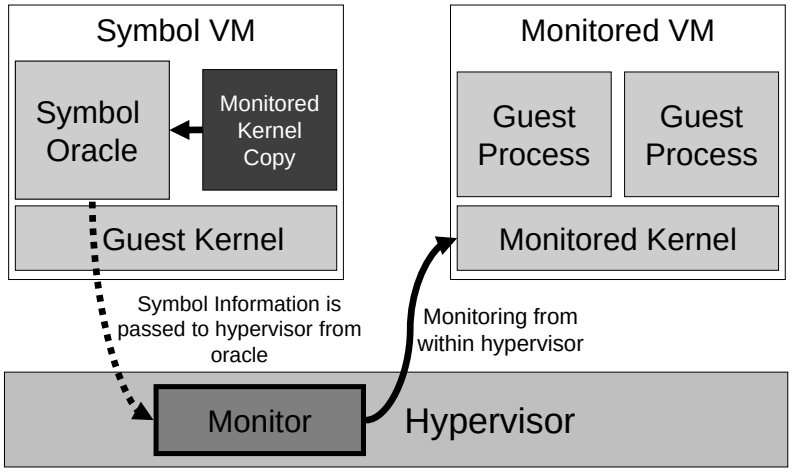

Figure 3: Sensor Architecture

tion, system call arguments must be marshaled and unmarshaled in both systems. In addition, page protections prevent the OS kernel from maliciously accessing the address space of the application so a full TLB-flush is required on each system call, while a trusted commodity OSs does not incur this cost. However, because Overshadow places the pages within the VM of the commodity OS, it must additionally encrypt and decrypt the pages of the security sensitive application whenever the OS wishes to access them. While this is an additional expense, it permits the OS to manage the resources used by the application. For example, Overshadow allows the OS to swap the security-sensitive application's pages to disk. In contrast, by putting the security-sensitive application in a separate VM, Proxos essentially gives the security-security sensitive application its own set of resources, which are managed by the private OS in that VM. Any re-allocation of resources between the security-sensitive application and the commodity OS would have to be handled by the hypervisor. Thus, for the cost of the encryption and decryption overhead, Overshadow permits the commodity OS kernel to handle resource sharing between the security-sensitive application and the rest of the commodity OS instead of relying on the hypervisor.

Both systems also require OS-specific code to be added to the TCB of applications. On the one hand, certain OS operations had to be emulated in the Overshadow hypervisor to be secure. For example, to support protected pipes between applications, Overshadow re-implements pipes outside of the commodity OS. Proxos does not need to do this because security-sensitive applications that need to communicate would already have their own private OS over which they could communicate. On the other hand, the original design of Proxos required some porting of applications to target the private OS, which was intended to be minimal and thus had less functionality than the commodity OS. For example, the private OS of Proxos did not support multiple address spaces or threads. Our more recent incarnations of Proxos utilize a standard commodity OS kernel as the private VM, but simply have unnecessary privileged code removed, reducing the attack surface. In addition, the private OS has no direct network access, further reducing the attack surface. While one always hopes to add as little complexity to the TCB as possible, by increasing the size of the hypervisor slightly, both in the case of Overshadow and Proxos, they are able to remove the majority of a commodity OS from the TCB of an application while at the same time permitting the controlled use of services in the commodity OS by the application. A promising approach that can further reduce the TCB is to remove the commodity OSs altogether and create smaller secure OSs with reduced functionality [12, 15]. An interesting open question is whether these smaller but brand-new computing bases are more secure than the larger but more heavily tested computing bases found in commodity software components. We will examine this issue further in Section 5.

4. USING INTROSPECTION TO DETECT OS COMPROMISES

An alternative approach to protecting applications against a compromised commodity OSs is to detect when an attacker has successfully compromised the OS and prevent damage at that time. Traditionally, intrusion detection systems (IDS) were implemented either in the network or on the host. Host-based IDSs are more accurate than network-based IDSs due to their visibility into the monitored OS, but suffered the disadvantage that they could be disabled if an attacker were to evade the intrusion detection system and compromise the host without the administrator being alerted. In addition, host-based detection systems present a larger administrative footprint since they must be installed and maintained on each individual machine. An exciting alternative to host-based and network-based monitoring for intrusions is to do the monitoring in a hypervisor by having it *introspect* on the guest OSs running on it. Implementing intrusion detection in the hypervisor holds the promise accuracy equivalent to a host-based IDS, but at the same time protecting the IDS from tampering by isolating it from the monitored OS in the hypervisor [6]. In addition, a single IDS implemented in the hypervisor, can simultaneously monitor several guest VMs in a coordinated way.

However, the benefit of being able to achieve host-based accuracy without the risk of tampering by an adversary is hampered by a problem called the "semantic gap" [3]. Guest OSs typically are not aware that they are running in a VM, and so make no special effort to communicate the meaning of events or data structures to the underlying hypervisor. As a result, the hypervisor must indirectly infer the significance of events and data structures and assign se-

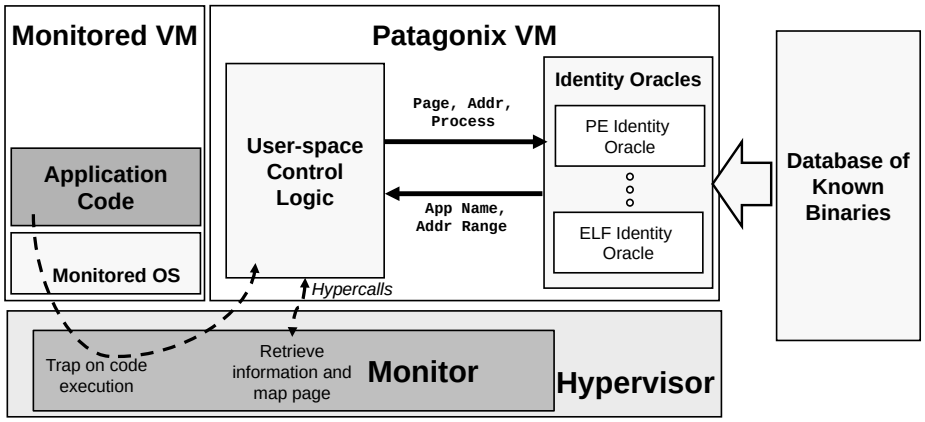

Figure 4: The Patagonix Architecture

mantic meaning to them. For example, if an attacker abused stolen privileges to alter the password file in an OS, all the hypervisor would perceive is a series of hardware interrupts caused by system calls, the reading and writing of certain locations in memory and perhaps some accesses to the virtual disk device. Without information mapping certain interrupts to read, write and open system calls, and certain disk blocks to the password file, the hypervisor cannot distinguish this event from other benign file accesses or events. This "gap" between events in the guest OS and what the hypervisor is able to reliably interpret from the resulting hardware events hampers the accuracy of any hypervisor-based monitoring system.

We have developed two systems that take different approaches to bridging the semantic gap between the hypervisor and the guest OS. In the first, we used symbol information extracted from the kernel compilation process to be able to interpret the meaning of bits in kernel memory [1]. Our *Sensors* system accomplished this as shown in in Figure 3. A user-space oracle extracts the symbol information from a copy of the kernel binary from the monitored VM and passes the information to a monitor implemented in the hypervisor. Since extraction of the symbol information is only done once at start-up, implementing this component outside of the hypervisor is efficient and keeps the hypervisor and TCB of the VMs small. The hypervisor then uses this information to interpret events and data in the monitored VM. Using symbol information was also implemented in the Livewire [6] and the two systems share many similarities. Both systems can be classified as "white-box" introspection since they assume the hypervisor introspection system has detailed knowledge, such as symbol tables, about the system it is monitoring.

With our Sensors, we were able to get high-fidelity information on events in the OS. For example, by applying this intrusion detection system to monitor several honeypots we had connected to the Internet, we were able to accurately and automatically detect when real attackers had exploited vulnerabilities in the honeypots and compromised the monitored OSs. With the information collected by our hypervisor and the kernel symbol table to serve as a map by which we could interpret the information, we could deduce in great detail the actions taken by those attackers.

However, white-box monitoring is vulnerable to deception by an attacker who is aware of what type of information the monitor uses to bridge the semantic gap. For example, suppose that one uses the hypervisor to monitor for the execution of certain system calls. One way to do this is to use kernel symbol information to find the location of the system call handler functions that will execute when an application makes a system call. Then, the hypervisor can replace the initial entry-point instruction of these functions with a trap instruction, which will notify the hypervisor when the function is executed. An attacker who is aware of this can evade detection simply by making a copy of the monitored function elsewhere and creating an alternate path to the copied and unmonitored function. For example, if monitoring the *open* system call, the attacker could make a copy of the *open* system call handler that the hypervisor is not aware of and thus does not place a trap in, and then redirect the system call table to point at the copy instead of the original. While the Sensors could also monitor the system call table, in general, an attacker who is aware of what is being monitored can always change code paths or references to evade the detection. Such evasion is also possible by tampering with memory data-structures so as to mislead the monitor.

The fundamental problem with white-box introspection is that it relies on *non-binding* information about the monitored system. Non-binding information is information that is true in an uncompromised OS, but can be arbitrarily changed by an attacker who has compromised the OS kernel. In other words, white-box systems depend on assumptions implied by the non-binding information, but the attacker is not bound to uphold these assumptions and can break them to evade detection by the monitor. Our conclusions from this work are that while the monitor logic is isolated from the monitored VM at all times, and thus cannot be tampered with by an attacker, the information that the monitor logic reports from the monitored VM is vulnerable to tampering by the attacker. Other than the fact that the introspection cannot be outright disabled by the attacker, white-box introspection is no more tamper-resistant than a host-based IDS system that logs all its monitoring results in a way that is isolated from the monitored VM, such as to a remote host.

To avoid this pitfall, we have also explored the design and implementation of two "black-box" introspection systems – an initial prototype called Manitou [10], and a more complete system called Patagonix [9]. Black-box introspection systems treat the monitored OS as an opaque system and only use externally visible events to infer what is going on in the OS. Both our black-box systems were experiments in trying to determine how accurate an introspection system that did not depend on any non-binding state could be. In this paper, we will focus our attention on Patagonix. The architecture of Patagonix is illustrated in Figure 4. Patagonix identifies all

executing binaries in a VM, and detects the execution of unknown binaries without trusting anything except the processor hardware and binary format specification. As with the Sensors system, Patagonix also implements a monitor in the hypervisor and implements non-performance critical components in a separate VM. As input, the Patagonix VM requires information about the binaries it will identify in the form of a list of known binaries. Any binaries not found in the list are identified as "unknown".

Patagonix operation is divided into two major tasks. First, it must detect when binary code executes. Then, it must identify the binary code or indicate that it is of unknown identity. To detect code execution, Patagonix's hypervisor monitor component leverages the processor's memory management unit (MMU) to detect code as it executes. The processor MMU performs virtual to physical translations and restricts execute and write access to memory pages according to page tables specified by the hypervisor. To efficiently detect the execution of code, Patagonix initially marks all pages in a VM as non-executable. When a page is executed, the hypervisor receives a trap and identifies the page. Once identified, Patagonix does not need to be informed of the further execution of instructions on that page unless the identity of the page changes, so Patagonix marks the page as executable, but non-writable. Since the page is non-writable, its contents and thus the identity cannot change. Any attempt to modify the contents of an executable page will trap to Patagonix, at which time Patagonix will again mark the page writable but non-executable. If the page is executed again after the modification, it must be re-identified before it is allowed to execute again.

To identify binaries, Patagonix uses *identity oracles*. An identity oracle is a function that takes a page, an address indicating where code was to be executed, and an address space identifier as inputs, and then returns the identity of the binary as the output. An individual oracle is needed for each type of binary that Patagonix is to identify. The reason is that the mapping between a binary on disk and an executable in memory is specific to the binary format of the executable. The oracle inverts this process to recover the original on-disk image of the page and thus match it to an entry in the known binary database. Certain types of binaries can be more difficult to invert than others. For example, Linux shared libraries, which are in the ELF binary format, are position independent and thus simple to identify because their in-memory format is the same as their on-disk format. In contrast, Windows 32-bit DLLs, which are in the PE format, are not position independent and must be relocated at load time, making their in-memory format different from their on-disk format. As a result, the location of all absolute references had to be added to the known kernel binary database. When identifying a page from a PE binary, the absolute references on the page needed to be relocated back to their default values before the page could be identified. However, without knowing the identity of a page, it was impossible to know which bytes on a page contain absolute references. To break this circular dependency, the Patagonix PE oracle exploits the fact that most binaries have a small number of entry points which must be executed before any other code in the binary and that binaries can only be relocated by a integral number of pages. As a result, the offset between the faulting address and the closest page boundary can be used as a key to quickly narrow down the number of candidate binaries a page could have come from. Using this method, Patagonix was able to relocate PE binaries back their original locations and then match them against their on-disk images without having to depend on any information from the OS.

When developing Patagonix, we had an intermediate design that depended on an untrusted *agent* that was installed in the OS. The agent gave "hints" to Patagonix that reported the identity and address at which every executing binary was located. Patagonix took this information and confirmed its truth by replicating the operations a legitimate loader would take in loading the binary from disk to a particular memory address. If the loaded memory image that Patagonix computes deviates from what is actually in memory, Patagonix knows that either the agent or the binary has been tampered with and raises an alarm. One drawback of an agent is simply that it is another piece of software that must be installed into all VMs and managed. Further, a version of the agent must exist for every OS to be monitored. As a result, we eventually abandoned this design in favor of an agent-less design. However, the use of an untrusted agent that provides hints has several advantages. First, it makes the Patagonix oracles simpler because instead of having to invert a complex loader operation, the oracle simply needs to reimplement what a legitimate loader would do. While we did not explore the possibility, it may have even been possible to use an existing loader whose integrity had been independently verified, thus reducing the effort needed to generate oracles for other binary formats. Second, we believe that in some cases, the use of an agent could even have performance advantages. The simpler and faster execution of the oracles may be able to offset the additional cost of transferring the agents hints into the Patagonix monitor.

Since Patagonix depends only on the binary format specification and the processor hardware, it is able to correctly identify binaries even if an attacker has compromised the OS kernel. This is because compromising the OS kernel neither allows the attacker to change the processor hardware nor the binary format specification used by the system, since both of these are independent of the kernel implementation. This explains why a single processor hardware specification or single binary format can be supported by different OSs. It is also worth noting that both processor behavior and binary formats are long-lived, and tend to remain unchanged even as kernel versions change. As a result, Patagonix works without modification on different OSs and different OSs versions.

5. DISCUSSION

From building these systems, we have learned some valuable lessons. We attempt to summarize these lessons under three points in this section.

Minimize TCB by implementing functionality outside of the hypervisor. In building these systems, we quickly learned to implement as much functionality outside of the hypervisor as possible. Initially, the reasons for this were purely practical. Implementing new code in the hypervisor leads to new bugs being added to the hypervisor, which usually crashed the development machine and were also much more difficult to debug. It also permits prototyping in scripting languages and the use of external libraries for cryptographic operations. However, while this conveniently facilitates development, a benefit that outweights the initial reasons is that the hypervisor is the most privileged component on the system, and is in the TCB of every VM in the system. While it is not always obvious whether a particular component should be implemented in the hypervisor or not, we believe that in general, there are only two reasons why one should implement security functionality directly in hypervisor. The first reason to implement functionality in the hypervisor is that it requires privileges that only the hypervisor has. For example, Patagonix requires the ability to manipulate executable and writable permissions on page table entries, as well as examine the contents of guest VM memory pages. As a result, the code that manipulated page tables needed to be added to the monitor in the hypervisor. On the other hand, the logic to examine the pages and perform lookups in the known binary database

are implemented as user processes in the Patagonix VM. To allow this code to access pages from another VM, the Patagonix VM was given special privileges to gain access to pages in another VM via the Patagonix monitor component in the hypervisor. In addition, the Patagonix VM was given access to special hypercalls we implemented as part of the Patagonix monitor that allows the user-space components to retrieve information about the address of the faults and the address space the fault occurred in. While this gives the Patagonix VM privileges that other VMs do not have, an attacker who successfully compromises the Patagonix VM does not get unfettered access to all VMs since the use of these privileges are still checked and controlled by the hypervisor, which remains in a separate and more privileged protection domain than the Patagonix VM.

The other reason to add functionality in the hypervisor is that it must communicate frequently with other components in the hypervisor or with components in other VMs. Communicating across protection domains is expensive so if it occurs too frequently, performance will suffer. As an example, a significant amount of logic is required in Patagonix to ensure that a page cannot be simultaneously executed and written at the same time. The key is to note that the permissions are applied to virtual pages, but a physical page may be simultaneously mapped to several virtual pages with different permissions. In other words, if Patagonix does not check for simultaneous inconsistent mappings, an attacker could simultaneously gain executable and write access to a page by constructing one set of page tables that maps a physical page as executable but non-writable, and constructing another set of page tables that maps the same physical page as writable but non-executable. To prevent this, each time a virtual page is switched from executable but non-writable to writable but non-executable or vice-versa the Patagonix monitor must ensure that all other virtual pages mapping to the same physical page are also switched. To make this operation efficient, Patagonix maintains a count of the number of mappings to each physical page. If the permissions on a virtual page change and Patagonix finds that there is more than one mapping to the physical page the virtual page maps to, then Patagonix must find all other page tables with mappings to the same physical page and adjust them accordingly. While both maintaining counts and page table-walking are complex operations, they are implemented in the hypervisor component of Patagonix because they need access to every page table belonging to the monitored VM.

Make hypervisor security events infrequent to keep performance overhead low. Security events are usually not simple and often require some complex check or operation. As a result, picking which events to check or secure can have an effect on the performance overhead of the hypervisor security system. As an example, the forwarding of system calls from the private VM to the commodity VM in Proxos involves communication across VMs, which requires five context switches. This turns many system calls, which normally take less than 1 μs to execute, into operations that take on the order of 10 μs to execute. Fortunately, system calls are fairly infrequent for most applications so in practice, we observed very little overhead on real application workloads. Similarly, when identifying a memory page, the cost of switching protection domains and transferring information from the Patagonix monitor in the hypervisor to the user components in the Patagonix VM took on the order of 40 μs. In addition, the user-space components took on the order of and additional 100 μs on top of the communication operations, resulting in an overall cost of approximately 140 μs. However, we were able to mitigate the overall cost of this operation by only requiring it to be paid when a page of code was executed for the first time. As a result, the overall overhead of Patagonix was still less than 5% in most cases. Security events that require frequent intervention are not good candidates for a hypervisor-based implementation because of the prohibitive cost of transferring events and information between the guest VM and the hypervisor.

Project	TCB Size (LOC)
Proxos [18]	10,478
Flicker [12]	4,415
HiStar [19]	15,350
Linux 2.6.34.15	8,356,048

Table 1: Size of recent small TCB projects in lines of code (LOC) compared to Linux.

Xen Version	Approximate Release Date	Size (LOC)
Xen-2.0.7	08/2005	175,942
Xen-3.0.2	04/2006	349,438
Xen-3.1.3	12/2007	567,117
Xen-3.2.0	01/2008	536,691
Xen-3.3.0	08/2008	910,066
Xen-3.4.0	05/2009	845,112
Xen-4.0	04/2010	919,129

Table 2: Growth in size of Xen hypervisor

The ability to support commodity code can result in a higher quality TCB. One of the advantages of using a hypervisor is that it is able to add security to existing commodity code. However, the addition of the hypervisor itself increases the size of the TCB since the system is now vulnerable to bugs in both the hypervisor and the original commodity OS. In our original Proxos design, we removed the commodity OS from the private VM in favor of a custom-crafted private OS that has less functionality, but as a result was considerably smaller. Recently, other proposals have made similar attempts to produce small but secure TCBs. We tabulate the sizes of Proxos, Flicker [12] and HiStar [19] and compare them against a commodity OS like Linux in Table 1. We note that the three research prototypes should not be compared to each other as they have different functionality: Flicker only contains support for cryptographic operations, TPM access and memory management, Proxos supports a good part of the Linux system call interface and HiStar actually contains drivers for hardware, as well as supporting its own system call interface. However, it can be seen that none of these security kernels approaches even 1% of the size of the Linux kernel.

One big reason that these systems are small is the fact that they are research prototypes and do not contain complete support for all possible applications one might want to use them for. Thus, they are likely to grow in size and complexity if they were turned into commercial products. An interesting analog is the growth of the Xen hypervisor code base as it transitioned from a research prototype to a commercially deployed system, which we tabulate in Table 2. While many of the additions that contributed to the increase in Xen's code base are not security related, to be useful, one can assume that many of these security-oriented OSs would also need to support additional functionality to enable developers to produce useful applications. Thus, we might expect a similar and rapid increase in code size if the concepts were adopted in a commercial product.

Various studies have also shown that older code bases tend to have a lower density of bugs. This is to be expected, since bugs

are found and fixed through use of the code. Chou et al's study of kernel bugs [5] finds that many kernel versions seem to stabilize after some time, indicating that so long as no more new code is added, the bug density of a given code base will significantly smaller than when it is first developed. In addition, it is unlikely for security bugs to be quickly removed from a code base. Ozment and Schechter [13] find that security bugs had a median lifetime of 2.6 years, and that over a 7.5 year period 62% of the vulnerabilities found during that period were introduced right at the beginning of the period. This evidence since to suggest that older but larger commodity code bases can be significantly more secure than newer systems despite their larger size. Newer systems will have higher bug density, and will have to go through a period in which more new code is rapidly added to their code base. After that, they will have to go through a stabilization period as bugs introduced during that growth phase are found and removed. Finally, in that process, applications may have to be ported or rewritten, further introducing new bugs.

One should not read this as an argument that we should not develop new systems that support security, but simply that comparing the code size of a new system against that of a commodity system is not instructive as their is often a large gap in code quality, bug density and functionality. However, we do believe that this indicates that in many cases, retrofitting security into existing commodity systems may yield a better result than building a new system from scratch by minimizing the risk of adding new bugs and at the same time still deriving some if not all of the intended security benefit. Hypervisors fill this role well since they can be used to add security functionality to commodity system without having to trust the existing system, and without requiring changes that risk introducing new bugs to be made to these large and already complex systems.

6. CONCLUSIONS

Even though security is increasingly important, for economic reasons, users are often stuck with commodity software. In addition, older, widely used code bases are less likely to contain bugs that lead to vulnerabilities. Hypervisors offer a way of improving the security of commodity systems without having to trust the existing commodity code base. Thus, their functionality remains intact even if an attacker is able to compromise the OS kernel. Currently, there exist a number of mature, commercial-grade hypervisors to which security functionality could be added. Because hypervisors need only to virtualize the underlying hardware, their design and implementation can be considerably simpler than that of a full OS kernel, and their interface to the untrusted guest VMs can be much narrower than an equivalent OS system call interface.

We have built several systems that implement two different approaches to securing commodity systems with hypervisors. The first approach, embodied in a system called Proxos, provides better isolation for security-sensitive applications than a commodity OS, but without constraining the way those applications can interact with the OS or other applications. The second approach, embodied in our Sensor, Manitou and Patagonix systems, introspects on commodity guest OSs to detect compromises. We have learned that some of the major challenges to implementing security support in the hypervisor include bridging the semantic gap without using non-binding information, supporting legacy code without unduly increasing the TCB, and finding ways to implement as much functionality outside of the core hypervisor as possible without impacting performance. We think several interesting avenues remain unexplored. For example, the spectrum between white-box and black-box introspection is broad, and we believe that there may be benefits to using a system based on untrusted agents that provide hints. In addition, the trade-offs between building fresh new systems and retro-fitting existing commodity systems with hypervisors remains a rich area of future research.

Acknowledgements

We would like to thank Kurniadi Asrigo and Richard Ta-min, who were reponsible for a large part of the implementation of the Sensors and Proxos systems respectively. Funding for some of the work described in this paper was provided through an NSERC Discovery Grant and the ISSNet NSERC Strategic Network.

References

[1] 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.

[2] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, pages 57–72, Aug. 2004.

[3] 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.

[4] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, May 2008.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, Oct. 2001.

[6] 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.

[7] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 2–13, June 2005.

[8] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 168–177, Nov. 2000.

[9] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. pages 243–258, July 2008.

[10] 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.

[11] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *FREENIX Track of the 2001 USENIX Annual Technical Conference (FREENIX'01)*, pages 29–42, June 2001.

[12] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 2008 ACM European Conference on Computer Systems (EuroSys)*, Apr. 2008.

[13] A. Ozment and S. E. Schechter. Milk or wine: Does software security improve with age? In *Proceedings of the 15th USENIX Security Symposium*, pages 93–104, Aug. 2006.

[14] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242, Aug. 2003.

[15] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of the 2006 ACM European Conference on Computer Systems (EuroSys)*, Apr. 2006.

[16] S. W. Smith, E. R. Palmer, and S. Weingart. Using a high-performance, programmable secure coprocessor. In *Financial Cryptography*, pages 73–89, Feb. 1998.

[17] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, June 2005.

[18] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 279–292, Nov. 2006.

[19] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, Nov. 2006.