VMM-based Systems for Enhancing Application Security

Koichi Onoue

Submitted to Department of Computer Science,
Graduate School of Information Science and Technology,
The University of Tokyo on December 15, 2009
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Thesis Supervisor: Akinori Yonezawa
Professor

Abstract

This thesis presents three security systems for controlling the behavior of application programs and protecting data relevant to those programs by using a virtual machine monitor (VMM), or more precisely, through control from outside virtual machines (VMs). The goal of this work is to design and implement security systems that simultaneously meet the following two requirements: (i) providing fine-grained control and protection mechanisms, and (ii) making subversion and evasion of these mechanisms difficult.

Security systems running on the OS or application level (e.g., sandboxing systems and anti-virus tools) can control the behavior of untrusted programs and prevent attackers from leaking and tampering with security-sensitive data. However, these security systems are potentially vulnerable because they run in the same execution space as untrusted programs. If an attacker can hijack OS kernels and privileged programs, it is not hard for the attacker to compromise security systems residing in the same execution space.

A VMM virtualizes execution of separate computing environments for each VM and allows users to run any programs, including OS kernels, inside their VMs. In terms of security, a VMM has two advantages. First, it can isolate the control and protection mechanisms of security systems from untrusted VMs. Second, it can control accesses to physical resources (e.g., physical memory and disks) at a higher privilege level than that of the OS kernels and application programs running inside untrusted VMs. However, there are two key challenges in improving application program security from outside VMs. The first challenge is to identify and manipulate in-VM states in OS-level semantics from the hardware-level states that a VMM can observe. The second challenge is to enable security systems to intercept events that a VMM cannot capture.

This thesis proposes three types of VMM-based security systems with different security concerns. The proposed systems have control and protection functionalities at the application program (target program) granularity while maintaining the above two VMM security properties.

The first proposed system, *ShadowVox*, controls the system call execution of target programs. The security concern is to control the behavior of target programs in user mode. ShadowVox can control untrusted VMs at process and system-call granularity from out-

side those VMs. To overcome the first challenge noted above, all three of the proposed security systems provide a technique for obtaining OS-level states by using information on the process management and system calls of OS kernels. To overcome the second challenge, the three systems incorporate a technique for allowing a VMM to intercept an arbitrary processor instruction. Experimental evaluation demonstrated that ShadowVox could control several types of application programs on different processor architectures. In addition, the system's effectiveness against an attack from a compromised program was demonstrated, and the overhead introduced by the system was measured.

The second proposed system, *Shadowall*, protects memory and virtual disk data involved with target programs. The security concern is to protect target program data in the user memory space and on virtual disks. Shadowall prevents compromised OS kernels and privileged programs from divulging or corrupting such data related to target programs running in the same execution space. Experimental results demonstrated Shadowall's effectiveness against malicious operations in existing application programs, and the performance penalties incurred by the system were measured.

Finally, the third proposed system, *ShadowXeck*, controls the behavior of OS kernels running on VMs. The security concern is to control the behavior of target programs in kernel mode. ShadowXeck addresses three problems with existing anti-malware systems: restriction on the use of kernel extensions, bypassing of system functionalities, and significant performance degradation. ShadowXeck does not conservatively impose any restriction on kernel extensions, and it reduces the performance impact on untargeted programs. Experimental evaluation demonstrated ShadowXeck's effectiveness against attacks by actual kernel-level malware, and the runtime overhead imposed by the system was measured.

Acknowledgements

First of all, my deepest appreciation goes to my thesis supervisor, Professor Akinori Yonezawa. I would like to thank him for his invaluable support. I am especially indebted to him for spurring my interest in this research area and giving me the opportunity to work in a first-class research environment. He has also been a great mentor and has always supported me during difficult times.

I am grateful to Research Associate Toshiyuki Maeda for his insightful comments and criticisms since I became a member of the Yonezawa Group. He supported me with his brilliant ideas and broad-ranging background. I have learned much from him, on various topics ranging from implementing programs to giving presentations and designing slides.

I am also deeply grateful to Associate Professor Yoshihiro Oyama. He suggested the direction of my work and supported me with his many bright ideas. From him, I have learned much about writing papers and having the right attitude toward research, among many other things. He significantly contributed toward improving this work, and it would not have been possible without his help.

I also profoundly thank Dr. Eric Chen for his valuable advice on both my research and my life. I have learned much from my interesting discussions with him.

I would also like to thank my colleagues in the Yonezawa Group for our daily research discussions and their assistance. This thesis owes much to their thoughtful comments. In particular, I gratefully acknowledge helpful discussions with Dr. Motohiko Matsuda, Junya Sawazaki, and Tomohiro Suzuki on several points in this thesis.

I dearly thank Office Administrators Yumiko Sakanishi and Kagari Watanabe for kindly supporting my day-to-day requests.

Finally, I would like to thank my thesis committee, who made this thesis possible. Their perceptive reviews greatly improved the quality of the thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

With the growing use of personal computers and the Internet, users have been increasingly interested in the security of their computing environments. The increasing interest in computer security has driven the development and growth in use of various security systems running on the operating system (OS) or the application layer. Security systems *control the behavior of untrusted application programs (untrusted programs)* and *protect application-specific or system-wide data from malicious programs*. To control the behavior of untrusted programs, security systems interrupt specific events that they focus on, such as system call execution and Web page access, and prevent and detect unintended behavior of untrusted programs. On the other hand, to protect application-specific or system-wide data, security systems prevent data leakage and tampering.

For example, researchers have proposed sandboxing systems [11, 31, 42, 45, 86] and intrusion detection/prevention systems [21, 37, 38, 98, 107] to control the behavior of untrusted programs running at the user level, or to detect anomalies in their behavior. Commercial anti-virus tools with real-time checking functionality [9, 70, 102] block untrustworthy program execution. For program code and data, to analyze, detect, and prevent leakage and tampering, there are security systems that check tainted data on memory [95, 118] or check file-system integrity [60, 80]. There are also systems for detecting stealthy, malicious software and preventing it from residing at the user or kernel level [110]. Furthermore, security-enhanced operating systems [4, 13, 46, 76] provide access control mechanisms at the kernel level.

Unfortunately, attackers also take existing security systems into consideration, and they attempt to take over programs, create leaks, and tamper with system-critical and personal data. If attackers hijack programs running with administrator privileges or OS kernels, they can subvert and evade security systems residing in the same execu-

tion space without difficulty. For example, an attacker can send a signal to a security system running on a compromised system to terminate or abort the security system, and thus eliminate it from the compromised system. Attackers can also corrupt data, on memory, related to security systems controlling the behavior of untrusted programs, so that the security systems cannot interrupt events generated by the untrusted programs. Furthermore, attackers can tamper with files used by security systems, such as policy and database files. Attack methodologies for hijacking computing environments have become more diverse and complicated over the past several years [25, 79, 108]. Even worse, attackers can abuse OS kernel extensions for enhancing security in the execution space [19, 59]. For all these reasons, the threat of attacks on computer systems remains a serious problem.

As an approach to addressing the problem of attacks that are aware of security systems, we focus on leveraging virtual machine monitors (VMMs), also called hypervisors [20, 35, 67, 71, 77, 78, 101, 105, 116]. A VMM multiplexes and virtualizes the underlying physical resources, such as physical memory and disks, and allows multiple OSs to run on one physical machine. With a VMM, each OS runs on a virtual machine (VM). Such an OS is called a *guest OS*. In recent years, various VMMs have been developed, and their performance and functionalities have been significantly improved. The technology also has received much attention as an infrastructure for cloud computing [47, 106]. From the security perspective, VMMs have two main advantages. First, they provide stronger isolation between VMs than isolation between processes. This is called *VM isolation*. Even if an attacker hijacks an OS kernel or programs with the administrator privilege inside a VM, it does not get easier for the attacker to take over the VMM or other VMs. Second, VMMs can control access to physical resources, such as physical memory and network devices, at a higher privilege level than that of VMs. This is called *highest privilege control*. Consequently, security systems based on a VMM pose more formidable barriers to attackers. For these reasons, it is effective and efficient to leverage VMMs to address problems associated with attacks on security systems.

1.2 Goal and Approach

The goal of this thesis is to design and implement systems that enhance the security of application programs, meeting the following two requirements.

Difficulty in disabling and abusing security systems: The systems should provide control and protection mechanisms that are hard to corrupt and circumvent. Moreover, the systems' control and protection mechanisms should not be exploitable by attackers. For example, the systems should not protect malicious programs.

Fine-grained control and protection for a wide variety of application programs: As con-

ventional security systems running on the OS or application layer, the security systems should control the behavior of application programs and protect related data in terms of OS-level semantics, i.e., with process and file granularity. In addition, the systems should provide security for a wide variety of existing application programs including legacy code. In other words, the systems should not require modifying the source code of application programs, compilers and assemblers [22, 30, 72, 75, 109, 120].

Toward this goal, we present three VMM-based systems that enhance the security of application programs: *ShadowVox*, *Shadowall*, and *ShadowXeck*. More specifically, the three security systems control the behavior of application programs and protect data relevant to them from outside VMs in which the application programs reside. We refer to the VMs in which the application programs reside as *target VM*s. The major concern for ShadowVox is to identify and control application programs in terms of OS-level semantics from outside target VMs. The major concern for Shadowall and ShadowXeck is to protect application programs from untrusted OS kernels running inside target VMs.

The three security systems address different security concerns. ShadowVox's security concern is controlling the behavior of application programs running in user mode. ShadowVox controls system calls executed by application programs. For these programs, it provides the same security guarantee as that of traditional sandboxing systems based on system call interposition [45, 86]. Unlike such sandboxing systems, ShadowVox also provides a security guarantee for the security system itself, i.e., ShadowVox, by using the VMM properties described above: VM isolation and highest privilege control.

Shadowall's security concern is protecting data manipulated by application programs running in user mode. It prevents leakage and tampering with respect to memory and virtual disk data related to application programs. For these programs, even if an attacker hijacks OS kernels running inside target VMs, Shadowall prevents the attacker from inducing leakage and tampering with program-related data. As with ShadowVox, Shadowall can also enhance its own security by using VM isolation and highest privilege control.

ShadowXeck's security concern is controlling the behavior of application programs running in kernel mode. More specifically, it controls OS kernel behavior related to application programs. For these programs, even if an attacker takes over OS kernels running inside target VMs, ShadowXeck can the control OS kernel behavior of only user-specified application programs. As with ShadowVox and Shadowall, ShadowXeck can also enhance its own security.

This "out-of VM" scheme not only makes subverting and evading the protection mechanisms harder but can also prevent attackers from abusing the protection mechanisms to protect malicious programs. On the proposed security systems, the user specifies which application programs should be protected from outside target VMs. More

concretely, the systems execution control commands and manage security policies in a separate, trusted VM. We refer to such user-specified application programs as *target program*s. The security systems do not require the source code of target programs.

The security ensured by the proposed systems is assumed to depend on a VMM as part of the systems' trusted computing base (TCB); that is, the systems must have a trusted, and reliable VMM. Virtual-machine-based-rootkits (VMBRs) [33, 51, 61] are malicious software running at the VMM layer. SubVirt [61] modifies the system boot sequence. King et al. also proposed running security mechanisms at a layer below VMBRs as a countermeasure against them. The security mechanisms this lower layer include a secure boot mechanism [16] and secure hardware mechanisms based on Trusted Platform Module, such as Intel TXT [28] and AMD Platform for Trustworthy Computing [15]. Blue Pill [51] and Vitriol [33] are VMBRs that migrate a running OS kernel into a VM and run a malicious VMM underneath the OS kernel on the fly by using hardware-assisted virtualization, through means such as the AMD-V [14] and/or Intel VT [29] architecture extensions. The usage scenarios of the systems proposed here are different from those of Blue Pill and Vitriol. The proposed systems apply to OS kernels running permanently on virtual execution environments, whereas the OS kernels hijacked by Blue Pill and Vitriol run directly on hardware, and Blue Pill and Vitriol dynamically insert their malicious VMMs underneath the OS kernels. Carbone et al. proposed GuardHype, a concept for preventing the operation of VMBRs, including Blue Pill and Vitriol [23]. GuardHype's VMM runs underneath other VMMs and controls the operations that they execute.

1.3 System Overviews

This section gives overviews of each security system.

1.3.1 Controlling System Calls

Simply isolating execution environments with a VMM does not provide sufficient security. A VMM cannot prevent exploited program parts from being used for malicious purposes. For example, if a Web server hosted on a VM is taken over, sensitive information kept in the server is revealed to the attacker. Moreover, the attacker can modify or delete the information stored in the OS kernel by abusing the server's privileges. An effective countermeasure to these problems is to combine the VM isolation scheme with security systems such as sandboxing systems. Specifically, a security system running outside a target VM controls the behavior of target programs inside the target VM, while VM isolation makes attacks on the security system harder.

Unfortunately, to control the behavior of target programs from outside target VMs, we cannot simply take advantage of existing security systems, with little or no modification

of their code. There are two challenges in enabling the functionalities of security systems from outside target VMs. The first challenge is to identify the inner states of target VMs at application program granularity from outside. A VMM obtains the execution states and events inside target VMs in terms of hardware-level semantics, e.g., register values, memory data, and interrupts. Security systems, however, need to interpret execution states and events of target programs in terms of OS-level semantics, e.g., processes and system calls. Therefore, we need to bridge the gap between the OS-level and hardware-level semantic views. This gap is known as the *semantic gap* [24].

The second challenge is to efficiently and effectively intercept events that security systems need to mediate for controlling target programs. There are events that a VMM cannot intercept but security systems need to mediate. For example, a VMM cannot intercept the instruction of an optimized system call invocation and the instructions of system call exits.

Therefore, we propose *ShadowVox*, a system for controlling the behavior of application programs by introducing two basic techniques to overcome the above two challenges. The first technique restores inner states in OS-level semantics by using prior knowledge on a guest OS kernel. The second technique intercepts system calls at processor instruction granularity. These two basic techniques are commonly used by all three of the proposed systems.

1.3.2 Protecting Memory and Virtual Disk Data

Existing security systems implemented in the OS or application layer assume that programs residing in the same execution space as the security systems, such as the OS kernel and privileged programs, have not been compromised. If an attacker compromises a privileged program that is vulnerable, regardless of whether other program are vulnerable, the attacker can take over any other programs, and induce leakage, and tamper with any data. In addition, to hide malicious software, such as rootkits, the attacker can also tamper with system-critical data on memory or disk.

To protect programs running at the user level, without the previous assumption that security systems run in the OS or application layer, there are security systems based on a VMM [41, 103] and a microkernel [100]. Either a VMM or a microkernel can strongly isolate a VM for a trusted program [41] or parts of a trusted program [100, 103] from a VM for untrusted programs. However, These approaches based on isolating the VM have disadvantages in three respects: resource consumption, VM context switches, and binary compatibility. In terms of resource consumption, the user needs to provide one VM solely for the purpose of protecting one trusted program [41, 100, 103]. In terms of VM context switches, extra inter-VM context switches occur to switch execution contexts between a trusted program and untrusted programs [41, 100, 103]. In terms of binary compatibility, the user is required to modify the source code of a trusted program in or-

der to separate parts containing security-sensitive manipulations, such as authentication procedures, from other parts of the program [100, 103].

Given the above considerations, we propose *Shadowall*, a system for protecting data related to an application program running inside a target VM from outside the target VM. Shadowall's basic approach is to conceal target program data from the other programs including OS kernels. Specifically, Shadowall protects memory data by using highest privilege control and protecting virtual disk data by using VM isolation.

To protect memory data related to an application program, the VMM provides a VM with different views, depending on differences in execution mode (kernel and user modes). No compromised programs residing in a VM can disable the memory protection mechanism at the VMM layer. The VMM implements this memory protection without making any guest OS kernel aware of Shadowall's memory translation.

To protect disk data related to an application program, Shadowall manages files related to the program (target files), such as an executable, in a separate VM. When the program manipulates the target file's content, the VMM intercepts the system call involved in the file operation, and the VMM and the separate VM emulate the intercepted system call. No compromised program residing in a VM can subvert this file protection mechanism based on file management in a separate VM and emulation of the file operation without routing through system call procedures in a guest OS kernel. In addition to protecting a target file from being leaked and its content from being tampered with, the file protection mechanism can also prevent exploits that manipulate vulnerability to symbolic links and relative paths.

1.3.3 Controlling OS Kernel Behavior

Malicious software running at the kernel level, called *kernel-level malware*, poses a more serious security threat, because its attacks damage entire systems, through activities such as tampering with OS kernel code and data, installing backdoors, and escalating the malware privileges. Various systems to overcome the threats of kernel-level malware have been proposed and developed [59, 62, 63, 68, 74, 88].

However, all of these systems have drawbacks in terms of the use of legitimate kernel extensions, the performance impact introduced by applying their protection mechanisms, and the timing of analyzing, detecting, and preventing malware. Users cannot flexibly extend untrusted OS kernels, because these conventional systems [62, 88, 99] only permit the use of kernel extensions that have passed static or dynamic checks. For performance impact, coarse-grained control based on a VMM [82, 99], i.e., control at page-level granularity, requires excessive runtime resources because write-protected pages also include data that is not to be protected. Control based on a CPU emulator adds substantial overhead for emulating executed instructions [88, 117]. As for the timing of checking malware, attackers can evade or subvert the protection mechanisms of prior systems in order

to periodically analyze and detect them [55, 73] by using kernel-level malware when the anti-malware systems are inactive.

Hence, we propose *ShadowXeck*, a system for controlling the behavior of a guest OS kernel associated with target programs. Control is achieved in two ways: write protection at page-level granularity, and control of processor instructions related to indirect jumps at runtime. For write protection, the VMM prohibits untrusted OS kernels from maliciously modifying read-only memory regions such as those containing program code. For control of processor instructions, the VMM controls indirect jump instructions, such as indirect jump and call instructions, invoked by an untrusted OS kernel at the target program context in accordance with security policies. While control of memory management operations and processor instructions is implemented by using highest privilege control, the control operations are executed and security policies are managed in a separate VM by using VM isolation.

In this usage model ShadowXeck does not induce any limitations, unlike the previous systems described above, on using kernel extensions inside VMs; rather, it registers them as legitimate kernel extensions in advance. For example, users can enable kernel extensions for use as parts of security systems that they have developed. After a kernel extension is enabled, its functionality applies to all processes in the kernel context, except for the processes to be controlled.

To control only the behavior of untrusted OS kernels running in the target program context, ShadowXeck provides guest OS kernels with different memory mapping, depending on whether the target programs are running in kernel mode. Consequently, ShadowXeck reduces the number of controlled indirect jump instructions, because the VMM does not intercept them in a kernel context associated with processes that are not to be controlled.

1.4 Contributions

The contributions of this work for each of the proposed systems are as follows.

ShadowVox: We have designed and implemented a security system for controlling system calls from outside VMs by using two basic techniques: introspection in OS-level semantics and interposition at processor-instruction granularity. The two basic techniques are also applied in the other two proposed systems: Shadowall and ShadowXeck.

Although Garfinkel et al. [43] initially proposed the idea of introspection in OS-level semantics from outside VMs, which they called *virtual machine introspection (VM introspection, or VMI)*, and a wide variety of VMI-based systems have been proposed, we have clarified the OS information required for VMI. Specifically, this

work reveals OS information relevant to process management and system calls for VM introspection and what this information depends. In addition, we have provided a support program for automatically generating the OS information relevant to process management and system calls.
To show that the proposed "out-of VM" scheme for controlling system calls is applicable to different processor architectures, we implemented ShadowVox on both Intel and AMD processor architectures and applied it to control several types of existing application programs. In addition, we demonstrated the effectiveness of the "out-of VM" scheme and measured the overhead introduced by ShadowVox.

Shadowall: We have designed and implemented a security system for protecting memory and virtual disk data by using memory multiplexing and file protection schemes. A memory protection scheme without cryptographic techniques has been integrated with a file protection scheme.
To evaluate the viability of the system, we demonstrated that Shadowall could protect memory and disk data related to application programs from being leaked and tampered with. the performance overhead was also measured.

ShadowXeck: We have designed and implemented a security system for controlling the behavior of a guest OS kernel involved with a target application program, without any restrictions on kernel extensions. ShadowXeck controls OS kernel behavior only when target programs run in kernel mode.
To show the viability of this system, we demonstrated the effectiveness of its protection schemes: prohibition against write operations to read-only regions, and control of indirect jump instructions in a kernel context involved with an application program. The performance impact of ShadowXeck was also measured.

1.5 Organization

The remainder of this thesis proceeds as follows. Chapter 2 describes the first proposed system, ShadowVox. We also describe the two basic techniques commonly leveraged by all of the proposed systems. Then, Chapters 3 and 4 describe the second and third proposed systems, respectively. Finally, we conclude the thesis in Chapter 5.

Chapter 2

Control of System Calls

2.1 Motivation

A VMM [20, 35, 67, 71, 77, 78, 101, 105, 116] provides strong isolation between VMs. This is called *VM isolation*. If an attacker succeeds in taking over an application program running in a VM, the damage caused by the compromised program is confined inside the VM by an isolated execution environment based on VM isolation. There have already been systems that enforce access control between VMs by using VM isolation [48, 69, 92].

Although these systems based on VM isolation provide coarse-grained control, i.e., control at VM granularity, they do not provide fine-grained control inside a VM, i.e., control at application program granularity. If an attacker succeeds in taking over an application program with administrator privileges in a VM, the attacker can insert malicious programs in the compromised VM and induce leakage of confidential data. A promising approach for overcoming this problem is to leverage security systems such as sandboxing systems [11, 45, 86, 109] and host-based intrusion detection systems [44, 49, 98, 107]. These systems can monitor and control the behavior of untrusted programs and prevent attackers from tampering with computing resources.

Schemes that combine a VMM and security systems can be classified into two types according to where the security systems run: "in-VM" and "out-of-VM" schemes. An "in-VM" scheme executes security systems on the same VM as the programs whose behavior is monitored and controlled by the security systems. The programs whose behavior is monitored and controlled are referred to as *target program*s. This scheme has two advantages. First, it requires little or no modification to the code of a VMM or a guest OS kernel. Second, OS-level semantic views, such as those for processes and files, are easily available to programs running on a guest OS kernel. Unfortunately, this scheme has a disadvantage. If an attack evades detection by security systems and successfully obtains administrator privileges of the guest OS kernel, it can subvert and elude the security

systems.

In contrast, an "out-of-VM" scheme executes security systems outside the VM on which the target programs run. This VM is referred to as a *target VM*. This scheme addresses the problem of attacks that are aware of security systems. Even if an attacker obtains the administrator privilege of a guest OS kernel, the attacker cannot stop the VMM or security system running outside the target VM. On the other hand, this scheme induces two limitations. One limitation is that security systems obtain only hardware-level semantic views exposed by a VMM, with information such as register values and memory data. It is not straightforward for the security systems to identify and control the inner states of target VMs in terms of OS-level semantics. The other limitation is that security systems interpose only events that a VMM mediates, such as privileged instructions, exceptions and interrupts.

For this work, we apply the "out-of-VM" scheme to enhance the security inside target VMs. As security systems such as sandboxing systems become widely used, attackers also attempt, with the existence of such systems in mind, to hijack server and client programs and to cause leakage and tamper with confidential data. Such attackers attempt to subvert or elude security systems by using privileged programs that have already been hijacked or by exploiting vulnerabilities of the security systems themselves. Several security systems run with administrator privileges. Therefore, it is important to protect not only server and client programs but also security systems.

In this chapter, we present *ShadowVox*, a security system for controlling system call execution invoked by target programs from outside target VMs. To overcome the above two limitations of the "out-of-VM" scheme, we apply two basic techniques: *virtual machine introspection(VM introspection)* [43], and *dynamic binary instrumentation*. VM introspection is a technique to fill the semantic gap [24] between hardware-level and OS-level semantic views by using OS information relevant to process management and system calls. Dynamic binary instrumentation is a technique to intercept events that a VMM cannot intercept without modifying the source code of the OS kernel.

The rest of this chapter is structured as follows. First, we describes the threat model. Next, in Section 2.3, we explain the above two basic techniques, which are first introduced for ShadowVox, but are commonly leveraged in all of the proposed systems. Section 2.4 presents the system design, while Section 2.5 describes the implementation details. Then, Section 2.6 presents an evaluation of the system. Finally, Sections 2.7 and 2.8 discuss related work and summarize the chapter, respectively.

2.2 Threat Model

ShadowVox is a security system for enhancing the security of ShadowVox itself. It runs outside target VMs inside a VMM and a separate VM. ShadowVox prevents compro-

mised programs with administrator privileges from corrupting ShadowVox itself. Even if an attacker takes over a program with administrator privileges, the attacker cannot force ShadowVox to terminate. Moreover, the attacker cannot undermine the files containing ShadowVox's security policies.

ShadowVox's security concern with target programs (e.g., a Web server and anti-virus tools) is to control their behavior in user mode. Specifically, like sandboxing systems running on an OS or application layer [11, 45, 86], ShadowVox controls system calls executed by target programs, according to security policies.

The VMM and the separate VM are part of the trusted computing base (TCB). Furthermore, an OS kernel running on a target VM, referred to as a *target OS kernel*, is also part of the TCB. If an attacker hijacks a target OS kernel, the attacker can subvert or evade ShadowVox's control. For example, the compromised target OS kernel could cause ShadowVox to misidentify data relevant to process management and system calls, as described in Section 2.3.1. In another example, the compromised target OS kernel could maliciously modify system call procedures in the kernel space, such as system call routines and tables. These attacks in the kernel space can be alleviated by using ShadowXeck, another of the proposed security systems, which controls the OS kernel behavior. Chapter 4 covers the details of ShadowXeck.

2.3 Basic Techniques for Controlling Application Programs from Outside VMs

In this section, we describe the two basic techniques for controlling VMs at application program granularity. These techniques are applied in all of the proposed systems- ShadowVox, Shadowall, and ShadowXeck. First, we describe a technique for identifying the inner states of a VM from outside the VM. The term *virtual machine introspection (VM introspection, or VMI)* [43] is used for the process of identifying these inner states from outside VMs. Second, we describe a technique for intercepting a processor instruction issued by a VM, called *dynamic binary instrumentation*.

2.3.1 Virtual Machine Introspection

The VM introspection technique here is implemented using OS information, i.e., prior knowledge about process management and system calls in OS kernels and data relevant to such process management and system calls. In this thesis, we clarify the OS information required for VM introspection. Specifically, we describe how to identify inner states in terms of OS-level semantics from outside of a VM and what the OS information required for VM introspection depends on. Furthermore, we present auxiliary programs

for automatically generating the data relevant to process management and system calls in OS kernels.

Process Identification

VM introspection approaches are primarily classified into two categories according to process identification methodologies. The first category is for detecting or implicitly controlling process events, such as process creation, from the context switching of the process address space. For example, Antfarm [56] identifies context switches of processes and inferred creation and termination of processes, by tracking write operations to the CR3 control register for the x86 and x86-64 processor architectures. The CR3 register includes the physical address of the top-level page table for the currently running process. A VMM can detect and control write operations to CR3 because they are privileged operations. This approach only assumes that an OS kernel provides different address spaces for each process, and most commodity OSs have satisfied this assumption. Therefore, this approach is also applicable to OS kernels, whose design is not open like Windows. For several reasons, however, this approach based on address space switching is not sufficient to achieve the current goals.

First, we need to identify which processes are running; that is, we need the names of processes in order to control only specific applications. In contrast, the above approach can only identify switching of the process context. In addition, this approach would require additional prior knowledge of OS kernels for identifying creation and termination of processes and program execution. For example, Sawazaki [94] has proposed a system based on address space switching, which identifies differences between process creation and termination and program execution. This system depends on how entries in the top-level page table are updated. Second, we also need to identify the inner states of VMs from other events occurring inside those VMs, such as system calls. The above approach, however, limits process identification. Finally, this approach cannot recognize differences between lightweight processes, namely *threads*, since those having the same thread group ID share their address spaces.

The second category of VM introspection approaches is for controlling the inner states of a VM by using prior knowledge of OS kernels running on the VM. This category is further classified into two types according to which kind of prior knowledge is leveraged. In this section, we focus on the Linux OS kernel, whose design is open, as a guest OS kernel. The first type, such as IntroVirt [58], leverages OS kernel functionalities, i.e., existing functions in the kernel code region. For example, this means invoking the `sys_getpid` function from outside a VM to identify the ID of a process currently running on the Linux OS kernel. Leveraging existing functions in the kernel code region requires two kinds of prior knowledge. One is the calling conventions of the x86 and x86-64 processor architectures, i.e., how functions pass arguments and how they receive a return value. The

other is information on functions provided by a guest OS kernel, including the virtual addresses of their entry points and the types of their arguments and return value. However, leveraging existing functions in the kernel code region has two drawbacks. First, extra context switches between a VMM and VM are incurred by invoking the existing functions. Second, this approach depends on the existing functions. If a VM uses a vulnerable function, it can be taken over. In fact, vulnerabilities have been reported in the Linux OS kernel [96, 97].

For these reasons, we adopt another approach for controlling the inner states of a VM by using prior knowledge. This approach, exemplified by Livewire [43] and VMwatcher [55], leverages prior knowledge of process management on guest OS kernels. VMwatcher obtains the data from a symbol exported by the guest OS kernel (`init_task_union`). Unlike with VMwatcher, we provide a mechanism for obtaining process-related data at the VMM layer according to how a guest OS kernel obtains hardware-state data, such as the values of registers and data in RAM. The mechanism depends on the versions of OS kernels and the architectures on which they run. In the rest of this section, we concentrate on version 2.6 of the Linux OS kernel and the IA-32 and AMD64 architectures. For example, the proposed systems provide a mechanism corresponding to the `current` macro to obtain instances of the `task_struct` object including the execution context of each process. In the IA-32 processor architecture, kernel versions from 2.6.0 to 2.6.19 obtain `task_struct` instances as pointers to a kernel mode stack stored in the task state segment (TSS), whereas kernel versions 2.6.20 and later obtain them from the FS or GS segment register. In the AMD64 processor architecture, version 2.6 uses the GS segment register.

With these mechanisms, the proposed systems leverage information on the data structures of a guest OS kernel. The information includes the offsets of member variables in kernel objects and the sizes of the member variables. Table 2.1 lists the information required in order to obtain process-related data. For example, the `pcurrent` variable of the `x8664_pda` object and the `pid` variable of the `task_struct` object are leveraged to obtain the process ID of a currently running process in Linux 2.6.16 on the AMD64 architecture. First, the `task_struct` instance is obtained using the FS register and the information in `pcurrent`. Next,the process ID is obtained using the `task_struct` instance and the information in `pid`. The proposed systems also leverage `THREAD_SIZE`, which indicates the size of the region including the `task_struct` instance and the kernel mode stack in the IA-32 architecture. All of the leveraged information depends on the processor architecture, the OS kernel version, and the configuration at the time of building the OS kernel image. Therefore, we could assume that each VM administrator will provide this information to the security systems.

However, this is a cumbersome task for VM administrators that have little knowledge of OS kernels. To overcome this problem, we provide a program (*script*) to automatically generate this information in the same way as generating `asm-offsets.h` on the Linux

Table 2.1: Information on data structures related to process management

kernel object	member variables	purpose
task_struct	pid, tgid	obtaining process and thread group IDs
	uid, gid, euid, egid	identifying user and group IDs
	comm	identifying process command name
	tasks	obtaining process list
	active_mm	obtaining pointer to mm_struct object
	real_parent	obtaining pointer to parent process
	children	obtaining lists of child processes
	sibling	obtaining lists of sibling processes
	ptrace	identifying whether process is monitored by `ptrace` system call
mm_struct	pgd	identifying top-level page table
	start_code, end_code	identifying range of code region
	start_data, end_data	identifying range of data region
	start_brk, brk	identifying range of heap region
	start_stack	identifying location of user mode stack
	arg_start, arg_end	identifying range of region allocated for command-line arguments
	env_start, env_end	identifying range of region allocated for environment variables
vm_area_struct	vm_start, vm_end vm_next, vm_flags	identifying information on memory-mapped regions
list_head	prev, next	tracking double linked list
x8664_pda	pcurrent	obtaining pointer to task_struct object (for kernel version 2.6.29 or older)
thread_info	task	obtaining pointer to task_struct object (for IA-32 and kernel version 2.6.19 or older)

OS kernel. The script generates information on process management and writes it to a file during the building of an untrusted OS kernel image.

System Call Identification

The proposed systems identify when application programs start and terminate, and they spawn a new process according to the execution of system calls. Shadowall also identifies system calls involved in manipulation of memory and files, to protect memory and virtual disk data related to application programs. In addition, ShadowXeck needs to identify information on all system calls to control their execution. Therefore, as with the process identification mechanism described above, the proposed systems provide mechanisms to manipulate data related to system calls by leveraging three types of prior knowledge about system calls.

The first type of prior knowledge is the calling conventions of system calls, such as usage of registers and the kernel mode stack. Specifically, we need to understand which registers are used to pass a system call number and arguments and to store a return value, and where the system call number, arguments, and the return value are on the kernel mode stack. This mechanism depends on the architecture on which OS kernels run.

The second type of prior knowledge is information on the number of a system call and the relation between the number and name. Since this information depends on the architecture where OS kernels run and the version of the OS kernel, we assume that VM administrators provide this information. As with automatic generation of information on process management, this information is acquired while building guest OS kernel images.

The third type of prior knowledge consists of understanding which arguments are pointers to user address spaces and the sizes of the user objects to which they point. This mechanism depends on the architecture where OS kernels run and the version of the OS kernel. Therefore, this information is automatically generated while building guest OS kernel images, in the same way as for information on process management. If an argument points to a character string, the data are manipulated until the terminating null character is found.

Limitations of This Approach

Although the proposed approach based on OS information can overcome the problems of approaches based on switching the CR3 value, such as Antfarm, it has two major limitations. The first limitation is greater dependence on OS kernels than with the CR3-based approach. In other words, OS kernel source code is required. The second limitation is

```
...
c0104c46:  89 f6          mov %esi, %esi

c0104c48:  <system_call>:
c0104c48:  50             push %eax
c0104c49:  fc             cld
c0104c4a:  06             push
...
```

(a) Before patching

```
...
c0104c46:  89 f6          mov %esi, %esi

c0104c48:  <system_call>:
c0104c48:  f4             hlt
c0104c49:  fc             cld
c0104c4a:  06             push
...
```

(b) After patching

Figure 2.1: Overwriting the first byte of a system call routine

that the trustworthiness of identifying the inner states depends on data relevant to process management and system calls managed by an OS kernel running inside the VM. If an attacker can artfully tamper with this data while maintaining consistency with the process management and system call procedures inside the VM, the attacker can evade the control and protection mechanisms of the proposed systems. Note that, unlike IntroVirt, the proposed approach does not depend on OS kernel functionalities.

2.3.2 Dynamic Binary Instrumentation

Although an original VMM can intercept privileged, sensitive instructions [83, 90], the proposed systems also need to intercept other instructions related to system calls and function pointers in the kernel space. Here, we present dynamic binary instrumentation, a mechanism for intercepting the execution of any instruction for the x86 and x86-64 architectures, whose instructions vary in length.

Dynamic binary instrumentation consists of two phases: patching and emulation. In the patching phase, dynamic binary instrumentation overwrites the instruction to be intercepted, replacing it with the HLT instruction. Figure 2.1 shows an example of the patching phase for the entry point of a system call routine in the kernel code for the IA-32 architecture. Next, the systems observe the system call invocation inside the VM, because HLT is a privileged instruction. In addition, they decode the instruction at the entry point of the system call routine and store its binary data sequence (`50` at `0xc0104c48` in Figure 2.1) in the VMM address space.

In the emulation phase, a general protection fault is induced by the HLT instruction when the system call is invoked inside the VM. After executing control for system call invocation, the proposed systems emulate the corresponding original instruction stored in the patching phase. In addition, they set the instruction pointer to the subsequent instruction and make the VM restart execution.

This technique enables the proposed systems to intercept any instruction without any negative effect on the subsequent instruction, because the length of the HLT instruction is one. They preserve the control flow to any instruction whose previous instruction has been overwritten. Furthermore, if there is an attempt to overwrite a previous or subsequent instruction after an instruction has been overwritten, a second patching is performed without regard to the first patching. The overhead incurred by instruction emulation, which is generally considerate significant, is reduced because this technique emulates only one original instruction.

There are two other approaches for intercepting events occurring inside a VM. The first approach is to intercept such events at page granularity. Before a VM starts running, writable pages including function pointers are marked as read-only. Then, write operations to these pages can be intercepted because these operations cause page fault exceptions. Unfortunately, this approach has two drawbacks. First, the execution of system calls cannot be intercepted, since they are contained in memory regions that are originally marked as read-only. Second, redundant page fault exceptions are generated when untargeted data, such as run queues, exist in the same writable pages. This causes substantial performance degradation because run queues are frequently updated.

The second approach is to use a CPU emulator. Although a CPU emulator can intercept all executed instructions, such interception adds substantial overhead due to the emulation of the executed instructions.

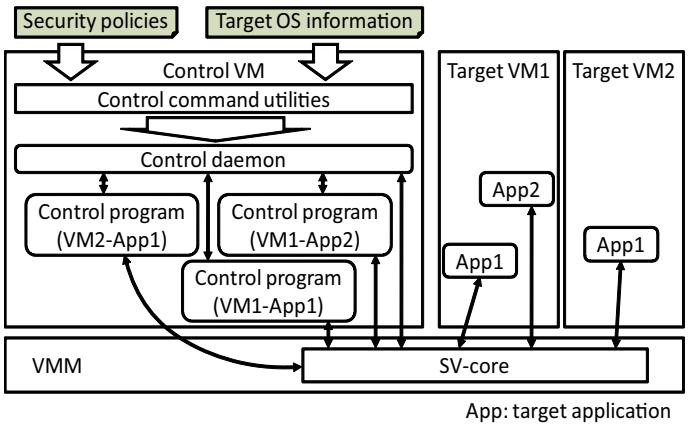

Figure 2.2: ShadowVox: system for controlling system calls invoked by target application programs

2.4 Design

2.4.1 Overview

As describe above, this chapter presents *ShadowVox*, a system for controlling the behavior of processes in a target VM, according to the information on low-level events observed by a VMM.

Figure 2.2 illustrates the structure of ShadowVox. The VMM and a special VM in the system cooperate to control the execution of system calls invoked by a specific program, according to security policies and the given system information for a target OS kernel. The special VM is referred to as a *control VM*, and the specific program whose behavior is controlled is a *target program*. The system information about the target OS kernel is referred to as *target OS information*. ShadowVox consists of an in-VMM component (*SV-core*), two types of programs (a *control daemon* and *control program*s), and *control command utilities* in the control VM. SV-core controls the system calls executed by the target programs. The main processing includes system call interception, identification of target programs, simple policy decisions, and execution of response actions for the intercepted system calls. On the other hand, the control VM manages security policies, decisions on how control programs are managed using the control command utilities, and complex policy decisions that SV-core does not make. The VMM and the control VM can simultaneously control multiple target VMs running on the same VMM.

In-VMM component: SV-core

SV-core receives the target OS information and security polices through the control command utilities, and it intercepts the entry and exit of system call procedures in the target OS kernel space. SV-core must also intercept the exit of a system call, because the execution must be identified and controlled for certain system calls, such as `fork` and `accept`. When a target program starts, terminates, and generates a new process, SV-core notifies the control VM of these events. When SV-core cannot also determine how an intercepted system call must be controlled, it forwards the policy decision to control programs. Later, Section 2.4.3 explains the proposed security policy.

Control VM

The control daemon manages the target OS information and security policies and notifies SV-core of this information through the control command utilities. Furthermore, the control daemon launches a control program when a target program starts. The control program in cooperation with SV-core controls the system calls invoked by the target program. A control program is launched for each instance of a target program.

ShadowVox provides the following control command utilities to the administrator of the control VM. All commands must be executed with administrator privileges.

- `sv_vps`: This command retrieves the ID of a target VM. It shows information on all processes residing in the target VM. The information includes process IDs, user IDs, and command names. Intuitively, this command is regarded as a remote version of the `ps` command in UNIX systems.

- `sv_vconf`: This command sends target OS information. ShadowVox holds system information by using the binary image file of the OS kernel. It distinguishes the binary image file by a hash value generated from the file content. The `sv_vconf` command receives the ID of a target VM and three files. The first file stores the memory layout of critical data structures of the target OS kernel. The second file stores system call information. The third file stores information about the memory locations where SV-core should intercept the entry and exit of system calls.

- `sv_vcntl`: This command enables SV-core to identify information about processes and system calls inside a target VM. It receives the ID of the target VM and the binary image file of the target OS kernel. First, the command links the target OS information given by `vconf` with the target OS kernel instance. Then, it overwrites the instructions at the entry and exit of system calls with privileged instructions.

- `sv_vstart`: This command starts execution control of a target program. It receives the ID of the target VM, the target program name, and a file specifying a security

```
[cvm] # sv_vconf targetOS.img \
          targetOS_info.txt syscall_info.txt syscall_hook.txt
[cvm] # sv_vcntl 1 targetOS.img
[cvm] # sv_vstart 1 apache2 policy.txt log.txt
```

Figure 2.3: Usage example in ShadowVox

policy. It provides a command-line option for user to specify the name of a log file for execution.

Usage Example

Figure 2.3 shows a sample session of the system. In this session, a user attempts to control processes in a target VM whose ID is 1. In the first line, the user gives ShadowVox the target OS information for a binary image file of a target OS kernel, `targetOS.img`. The user also specifies three files containing information on data structures related to process management in the target OS kernel (`targetOS_info.txt`), system calls of the target OS kernel (`syscall_info.txt`), and instructions intercepted by SV-core (`syscall_hook.txt`). Next, the user executes the `sv_vcntl` command to enable SV-core to control processes and system calls inside the target VM. In this session, the information registered in the first line is used as the target OS information of the target VM. In the last line, the user starts controlling a target program, `apache2`, with a security policy in a file `policy.txt` and a log file `log.txt`.

ShadowVox can also control currently running processes. First, a user obtains information on all processes inside a target VM with the `sv_vps` command. The user then chooses a list of processes to place under control and stores the information for specifying the processes in a file, `proclist-cntl.txt`. The target processes are specified with process IDs or program paths. Finally, the user starts controlling the processes specified in `proclist-cntl.txt` with the `sv_vstart` command.

2.4.2 Leveraging Knowledge of OS Kernels

ShadowVox controls system calls invoked by target programs inside VMs by using the two techniques described in Section 2.3: VM introspection and dynamic binary instrumentation. Whereas VM introspection enables ShadowVox to identify inner states at the OS level (e.g., processes and files), dynamic binary instrumentation enables ShadowVox to interpose execution of an arbitrary processor instruction. The mechanism of VM introspection restores execution states at the OS level from those at the hardware level, such as register values and data on memory, according to two types of prior knowledge about

an OS kernel. The first type is knowledge about process management, i.e., how an OS kernel identifies kernel objects related to process management from hardware states. The second type is knowledge about system calls, such as calling conventions.

The above two techniques require system information about an OS kernel, which the target VM administrator needs to present. This includes the following information:

- **Data structures related to process management**: This is information on the memory layouts of kernel objects and the maximum size of the kernel stack. Memory layout information consists of a pair of elements: the offset of a member variable in a kernel object and its size (e.g., on Linux, the process ID variable, `pid`, in the process management object, `task_struct`). The VM introspection mechanism leverages this information to identify and control information on processes. Since the VM introspection mechanism provides a support program to automatically generate this information at build time, users do not need to be familiar with the data structures of an OS kernel.

- **System calls**: This includes the number of system calls, and the relationship between the number and name of each system call. This category also includes information, for each system call, on which arguments are pointers to user address space and the sizes of the user objects to which these arguments point. The VM introspection mechanism uses the information to identify and control information on system calls. As with data structures related to process management, the VM introspection mechanism provides a support program to automatically generate this information.

- **Memory locations of the entry and exit of system calls**: These are the virtual addresses of the entry and exit of system calls in the kernel address space. The dynamic binary instrumentation mechanism uses these addresses during patching phase to overwrite the virtual addresses with the privileged HLT instruction. This information depends on having a binary image of an OS kernel. ShadowVox acquires this information from a symbol table file, `System.map`, generated at build time for an OS kernel.

Though ShadowVox currently supports only Linux as the target OS, it can support other UNIX-like OSs if the above-mentioned information is available.

2.4.3 Security Policy

Syntax

Figure 2.4 shows an extracted version of the security policy syntax used by the proposed system. Appendix A lists all of the syntax. As with traditional sandboxing systems based

PolicyFile	→	`default:` DefAction ModuleSpec*
DefAction	→	Action | `skip`
Action	→	`allow` | `deny(`*errno*`)` | `killProc(`*signame*`)`
		| `policyChange(`*policyfile*`[,`*logfile*`])` | `ask`
ModuleSpec	→	ModuleName `default:` DefAction SysCallSpec*
ModuleName	→	`processMod` | `fileMod`
SysCallSpec	→	*syscallName* `default:` DefAction ControlExpr*
ControlExpr	→	Cond* Action
Cond	→	FileCond | Cond `or` Cond
FileCond	→	`fileEq(` *argnum*,*path*`)` | `filePrefixEq(`*argnum*,*pathprefix*`)`

Figure 2.4: A part of security policy syntax in ShadowVox

on system call interception (e.g., Systrace [86]), the security policies used here specify controlled system calls through pattern matching of system call arguments. The policy syntax emphasizes support for flexible, fine-grained policy description more than support for policy description at a high level of abstraction (e.g., the application level) in this thesis. Eventually, the security policy syntax should also support user-friendly policy description at a higher level of abstraction.

The `default:` field at the top level indicates "Action," an action taken for system calls that do not match any pattern. Part of "ModuleSpec" specifies the rule for each system call. System calls are classified into eleven groups, called *module*s in ShadowVox. Examples of these modules are a module for process operations (`processMod`), a module for file operations (`fileMod`). All eleven modules are described in Appendix A. Each system call necessarily belongs to only one modules. Examples of system calls are `execve`, `fork` and `clone` for `processMod`; `open`, `chown`, and `poll` for `fileMod`. The `default:` field in the specification for each module indicates the default response action for the module.

Part of "SysCallSpec" indicates the rules for an individual system call. The head of "SysCallSpec," *syscallName*, gives the system call name. The `default:` field for each system call indicates a default response action for the system call. Part of "CotrolExpr" specifies pairs consisting of system call argument patterns and the corresponding response actions. Examples of such argument patterns are the file name (`fileEq`).

The elements in "Action" indicate the actions taken when a system call matches a pattern. `allow` continues execution of the system call, `deny(`*errno*`)` causes execution of the system call to fail with the error code *errno*, and `killProc` sends the signal named *signame* to the target process. `policyChange(`*policyfile*`)` dynamically switches the current security policy to the one specified by the file *policyfile*. `ask` asks the user to provide the

```
default:  deny(EPERM)
fileMod default:  allow
  open default:  allow
    fileEq(1,"/etc/passwd") or filePrefixEq(1,"/etc/cron.d")
      deny(EACCES)
processMod default:  allow
  execve default:  killProc(SIGKILL)
    fileEq(1,"/usr/bin/wserver") policyChange("wserver.pol")
```

Figure 2.5: Sample security policy in ShadowVox

response action for the intercepted system call when it was executed. The user can select any response action except for `ask`. As with `allow`, `skip` continues the execution of the system call. Whereas `allow` notifies the control program of the system call interception, however, `skip` continues execution without notifying the control program. Therefore, performance degradation due to control of system calls should be reduced by specifying `skip` as the response action for system calls that are not security-sensitive, such as `wait`, `poll`, and `gettimeofday`.

Figure 2.5 shows an example of a security policy. When ShadowVox is given this sample policy, system calls that do not belong to the `processMod` module or the `fileMod` module will fail with the error code `EPERM`. The execution of system calls that belong to `processMod` or `fileMod` but do not match any pattern will be continued. Opening the file `/etc/passwd` or any file under directory `/etc/cron.d` will fail with the error code `EACCES`. Any other file can be opened. When a target process invokes `execve` with the argument `/usr/bin/wserver`, ShadowVox switches to the security policy stored in the file `wserver.pol`. The execution of `execve` with other arguments will fail and `SIGKILL` will force the target process to terminate.

Description

Although a ShadowVox user requires information on the system calls executed by a target program to describe an appropriate security policy, it is not straightforward to acquire such information. The user can describe the security policy by using a log file containing information on the system calls executed by the target program. To generate such a log file, the user runs the target program on ShadowVox with a security policy whose `default:` field at the top level contains `allow`. The `allow` specification signifies that the control program will be notified of the system calls executed by the target program.

However, it is difficult to collect all system call patterns that target program can potentially execute during log-file generation. The problem in policy generation from a log

file is the specifications for system call patterns that are not recorded in the log file but that the target program can execute. To address this problem, ShadowVox provides the `ask` response action, which asks the user to determine the response action at runtime.

Unlike ShadowVox, another approach uses static binary analysis [36, 87] to acquire information on system calls executed by a target program from its binary code. Although this approach can collect all the system calls executed by a target program, it has two drawbacks. First, it requires the target program's code and the shared libraries that it uses. In contrast ShadowVox does not assume that target VM administrators provide the binary code of target programs. Second, this approach cannot acquire information on system call arguments whose values are not determined until runtime. For these reasons, the proposed system adopts the log file approach.

2.4.4 Advantages

ShadowVox has the following advantages over other security systems.

Difficulty in attacking the security system: If an attacker takes over a process in a target VM, the attacker cannot take control of the control VM and the control programs because of the isolation between VMs.

Process-granularity execution control: A response action is applied to anomalous target processes only. Other processes in the same target VM are not affected. In contrast, if execution control was applied at VM granularity, possible response actions would be coarser, such as a VM restart, which has the serious drawback of also terminating benign processes.

Uniform control of multiple VMs: If the same program is running in multiple target OSs on a VMM, the system can simultaneously control multiple process instances of the program with the same security policy. For example, if the same Web server program is running in ten guest OSs on the same VMM, one control VM can control ten servers with the same security policy.

Security enhancement of unprotected OSs: The proposed system is also useful for virtual hosting in which each target VM and target OS is managed by a different administrator. Some target OSs might be managed by a novice administrator and might not be sufficiently protected against attacks. ShadowVox provides a safety net for such target VMs and target OSs.

2.5 Implementation

ShadowVox has been developed using the para-virtualization version of Xen [20] 3.0.3

as the VMM, with para-virtualized Linux OS kernel 2.6.16 as the target OS. ShadowVox supports the IA-32 and AMD64 architectures, and virtual multiple processors.

2.5.1 Obtaining Process Information

As mentioned in Section 2.3.1, ShadowVox identifies inner states related to process management by using the VM introspection technique. The VM introspection technique is implemented by using the data structures of a target OS kernel, such as the offsets and sizes of member variables in kernel objects (e.g., the `pid` variable in the `task_struct` object). ShadowVox provides a support program for automatically generating these data structures in the same way as generating the `asm-offsets.h` file on Linux. More concretely, the offsets and sizes of member variables in kernel objects are generated from, respectively, the `offsetof` and `sizeof` Linux macros at the time of building a target OS kernel image.

The memory address space for the para-virtualization version of the Xen VMM is shared among all VMs. The virtual address space is not changed during system call execution. Hence, SV-core (the in-VMM component) can directly access instances of `task_struct`, which is a kernel object for managing the execution context of a process, when it intercepts system calls. The process management data on Linux, including `task_struct`, are resident in guest memory.

`sv_vps` is a command for obtaining a list of processes in the target VM specified by a given VM ID. The VMM obtains this information as follows. First, if the target VM is running, SV-core stops it. The VMM switches its address space to that of the target VM by using the execution states kept by the virtual CPU of the target VM. Next, SV-core obtains a list of processes by using register values and memory data of the target VM and tracking the `tasks` member variables of `task_struct` instances. Finally, SV-core switches the address space back to the original one and sends the obtained information to the control VM.

2.5.2 Managing Target Processes

When the `sv_vstart` command is executed, SV-core adds the specified program file paths or process IDs to a hash table for managing target processes. When the `execve` system call is invoked in the target VM, the VMM checks whether the given program path is in the hash table. If it is, SV-core adds the ID of the invoking process to the list; otherwise, it does nothing. Currently, ShadowVox does not support paths that are symbolic links. When the `fork`, `clone`, or `vfork` system call is invoked in a target VM, the control VM checks the given security policy. If the policy specifies the `controlChild : detachProc` option (described in Appendix A), the new process is not controlled. Otherwise, SV-core adds the new process ID to the hash table. Lastly, when a target process invokes the

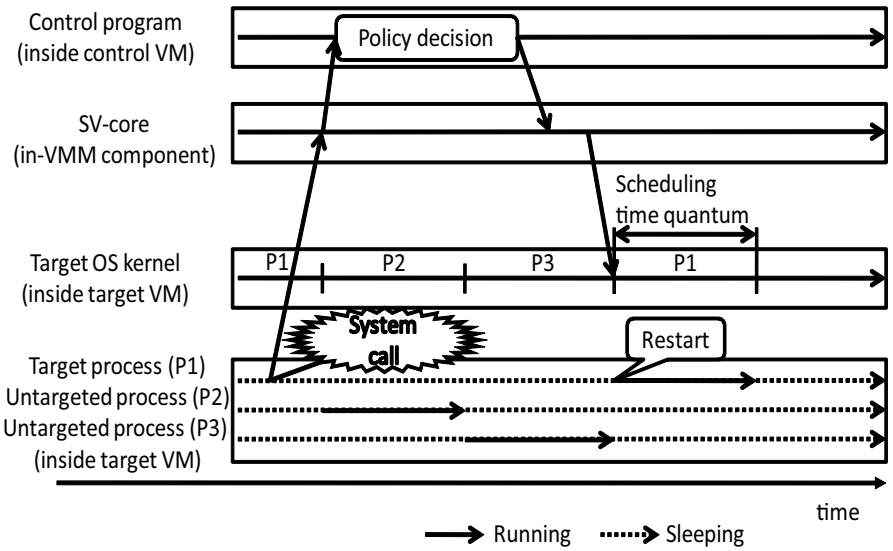

Figure 2.6: Control flow on ShadowVox for system call invocation by a target process

`exit_group` or `exit` system call in a target VM, SV-core deletes the hash table entry for the target process ID.

2.5.3 Controlling System Calls

When a system call is invoked in a target VM, SV-core first checks whether the invoking process is one of the target processes. If it is not, SV-core does nothing special and lets the system call continue. Next, it checks whether the response action for the invoked system call is `skip`. If it is, SV-core lets the system call continue. Otherwise, it reads the execution states of the target VM and obtains the number and arguments of the system call. It then notifies the control program that a system call has been invoked. The control program determines the response action and notifies SV-core what that is. SV-core then executes the response action and continues the target process (if it is still alive). As for system call exits, SV-core checks only those system calls involved in managing target processes and policy decisions (e.g., `fork` and `accept`) and lets the execution of other system calls continue.

To communicate between the VMM and the control VM, ShadowVox uses two functionalities provided by Xen-the event channel and shared memory-which are created and deleted for each control program.

Figure 2.6 shows an example of the control flow until a response action for a target process (P1) is taken. In the rectangle of processes inside the target VM, solid lines signify

a process that is running and dotted lines represent a process that is sleeping. To minimize negative effects of execution control on untargeted processes, when a system call is checked, only the corresponding target process is suspended; the enclosing target VM is *not* suspended. In Figure 2.6, untargeted programs (P2, P3) are *not* suspended while the system call invoked by P1 is checked. To achieve this, ShadowVox makes the target process repeatedly execute the intercepted instruction until the policy is enforced.

Currently, ShadowVox does not have a mechanism to prevent race condition attacks in which another thread modifies system call arguments [113]. Many techniques have been proposed to prevent such attacks [40, 42], and several could be applied in ShadowVox. However, preventing race condition attacks involves a tradeoff between security and performance. It would be an interesting to evaluate the effectiveness of these techniques in the context of security setting proposed here.

2.5.4 Obtaining System Call Information

SV-core identifies the number and arguments of an executed system call from register values and data on the kernel stack of the target VM, using the VM introspection technique described in Section 2.3.1. The relationships between the number and name of a system call, the type and size of each argument, and the calling conventions depend on the OS kernel version and the architecture on which the OS kernel runs. In particular, there are four major architectural differences.

The first difference is the number of system calls. For example, on Linux 2.6.16, used for developing the prototype system, the number of system calls for the IA-32 architecture is 311, whereas the number for the AMD64 architecture is 273. The second difference is the relationship between the number and the name. For example, system call number 2 is `fork` on IA-32 but `open` on AMD64. Furthermore, there are system calls supported by only one of the two architectures. For example, whereas `readdir` is only supported on IA-32, `arch_prctl` is only supported on AMD64. The third difference is in the system call interfaces for network operations and interprocessor communication (IPC) operations, i.e., semaphores, message queues, and shared memory. On IA-32, `socketcall` is the system call for network operations, and the first argument determines which socket function (e.g., socket, bind, or connect) is executed. Similarly to the network operations, `ipc` is the system call for IPC operations, and the first argument determines which IPC function (e.g., semop, msgsnd, or shmat) is executed. In comparison, on AMD64, a system call is defined for each socket or IPC function. The fourth difference is in how arguments are passed on `mmap`. They are passed using the user stack on IA-32 but using registers on AMD64. (On IA-32, `mmap2` passes the arguments by using registers.)

The VM introspection technique relies on the above knowledge. As with information on process management, ShadowVox provides a support program for automatically generating data relevant to system calls. More precisely, the relationships between the

number and name of a system call, the size of the user object pointed to by an argument, and the memory layouts on the kernel stack are generated, respectively, from the header files, such as *unistd.h* for both the IA-32 and AMD64 architectures on Linux kernel 2.6.16, the `sizeof` macro, and the `offsetof` macro, at the time of building a target OS kernel image.

2.5.5 Obtaining System Call Arguments

The control VM uses system call arguments to control the execution of system calls. When a system call is intercepted, SV-core stores the value of each system call argument. If the argument type is an on-memory object, the control VM reads the address space of the user process in the target VM. Examples of on-memory objects are path names given to an `open` system call and network addresses given to a `connect` system call.

When the control VM accesses an address in the user space of a target VM, a page fault may occur. ShadowVox avoids such page faults as follows. If a system call argument is a virtual address in the user space, SV-core reads the page table of the target process and checks whether a page enclosing the address is present. If not, it forces the target OS kernel to handle page faults. When the target OS kernel completes this handling, SV-core then notifies the control VM of the system call invocation.

2.5.6 Applying Response Actions

SV-core applies a response action to a target VM when an invoked system call matches a rule in a given security policy. The `killThread`, `detachProc`, and `createNewMonitor` actions are explained in Appendix A.

As the response action for a system call failure (`deny(`*errno*`)`) at the entry of a system call, SV-core changes the invoked system call to `getpid`. It replaces the return value of `getpid` with a number corresponding to the error code name *errno*. For the failure response action at the exit of a system call, such as `accept`, `recvfrom`, or `recvmsg`, SV-core first forces the target process to execute the `close` system call. It then replaces the return value with the error code number.

As the response action for killing a target process or a target thread (`killThread`), SV-core forces the target process to send the signal *signame* with the `kill` or `tgkill` system call, respectively.

As the response action for switching a security policy (`policyChange`), SV-core changes to the security policy in the file *policyfile*. *policyfile* is read not when SV-core takes the response action but when the target program starts.

As the response action for asking the control VM administrator which response action SV-core should take (`ask`), a control program first outputs information on the intercepted

system call. The control VM administrator decides the response action according to this information, and the control VM notifies SV-core of the response action.

As the `detachProc` response action when a target process executes the `ptrace` system call, SV-core deletes the target process monitored by `ptrace` from the hash table for managing target processes. As another response action when a target program executes `ptrace`, for the case of `createNewMonitor`, SV-core notifies the control daemon of launching a new control program for the target process monitored by `ptrace`, in addition to deleting the target process from the hash table.

2.5.7 Intercepting System Calls

System Call Handling in Linux

Linux OS kernel 2.6 running on the IA-32 architecture (Linux-IA32) supports two schemes for invoking system calls. One uses a software interrupt, the INT 0x80 instruction. The other uses the SYSENTER instruction. Either scheme can be chosen at boot time according to the processor version.

SYSENTER is an instruction provided by IA-32 architectures more recent than the Pentium II. The behavior of SYSENTER is configured by the value in the model-specific register (MSR). Updating MSR values requires the highest privilege level. When SYSENTER is executed, MSR values are loaded into several registers, including the instruction pointer and the stack pointer. In addition, SYSENTER sets the privilege level to zero (the highest).

Linux-IA32 supports two schemes for switching kernel mode to user mode at the exit of system calls. One uses interrupt handling through the IRET instruction, while the other uses the SYSEXIT instruction.

Linux OS kernel 2.6 running on the AMD64 architecture (Linux-AMD64) uses the SYSCALL instruction to invoke system calls. Linux-AMD64 supports two schemes: interrupt handling through the IRET instruction, and the SYSRET instruction.

A VMM can intercept the software interrupt, SYSENTER/SYSEXIT, and SYSCALL/SYSRET, since their handling requires the highest privilege level. However, it cannot intercept interrupt handling through IRET, because switching to a lower privilege level does not require the highest privilege level.

System Call Handling in Xen

In the para-virtualization version of Xen running on the IA-32 architecture (Xen-IA32), a system call is handled without passing through the VMM by using the software interrupt. When a system call is invoked, control moves directly to a system call handler in a guest OS. The source code of a guest OS for Xen is modified to modify the config-

uration of an interrupt descriptor table. Furthermore, Xen-IA32 does not support SYSENTER/SYSEXIT. On the other hand, in the para-virtualization version of Xen running on the AMD64 architecture (Xen-AMD64), a system call is invoked with SYSCALL. At the exit of a system call, both Xen-IA32 and Xen-AMD64 use interrupt handling through IRET.

System Call Interception in ShadowVox

ShadowVox uses the dynamic binary instrumentation described in Section 2.3.2 for intercepting a system call entry on Xen-IA32 and a system call exit on Xen-IA32 and Xen-AMD64. The source code of a target OS kernel does not need to be modified to introduce dynamic binary instrumentation. In other words, the original target OS code for Xen can be used with ShadowVox.

To intercept system call invocations using the software interrupt, ShadowVox patches the binary code of the target OS kernel. When the `sv_vcntl` command is executed, SV-core overwrites the first byte in the code of the system call interrupt handler with the HLT instruction. The code address is included in the target OS information. HLT is a privileged instruction, so if it is executed at the user level, an interrupt is raised and control returns to SV-core. To continue the intercepted process, SV-core emulates the overwritten instruction and passes control back to the target OS kernel.

ShadowVox forbids modifying the physical page enclosing the instruction overwritten with HLT. Hence, even if an attacker hijacks a target VM, the attacker cannot tamper with the instruction overwritten with HLT and thus cannot elude interception by SV-core.

Finally, in implementing ShadowVox, Xen-IA32 was extended to support system call handling using SYSENTER/SYSEXIT.

2.6 Evaluation

We evaluated three aspects of ShadowVox: applications to existing application programs, its effectiveness, and its impact on performance.

2.6.1 Applications

To confirm that ShadowVox controls system calls executed with several types of application programs, we tested three cases in which ShadowVox controlled server programs and security systems on both the IA-32 and AMD64 architectures. First, target programs were run on ShadowVox, and the system calls that they executed were recorded from start up to termination. Next, the target programs' security policies were generated from log files, and ShadowVox was used to control the target programs according to the generated security policies. The three test cases were the following:

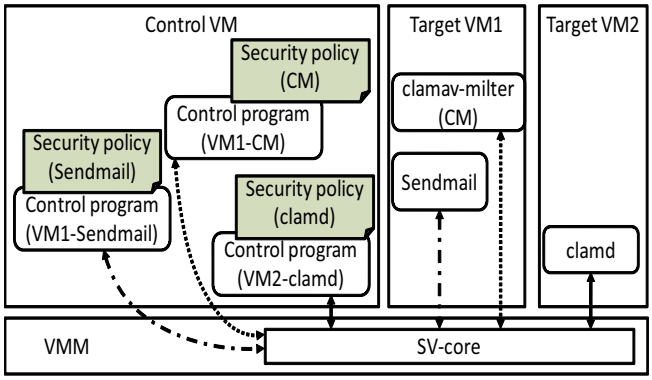

Figure 2.7: Application example for *Sendmail* and *ClamAV* (*clamav-milter* and *clamd*)

Controlling server programs: The target programs were two Web servers, *Apache* and *thttpd*, and a mail server, *Sendmail*. For the Web servers, requests to fetch static and dynamic content were sent. For the mail server, requests to send and receive e-mails were sent. The capability of ShadowVox to control multiple Apache instances running on different target VMs using a common security policy was also tested.

Controlling security programs: Five programs were selected as follows: an anti-virus tool suite, *ClamAV* [2]; host-based and network-based intrusion detection systems, *Tripwire* [60] and *Snort* [8], respectively; a command-line program for tracing system calls *strace*; and a sandboxing system based on system call interposition, *Systrace* [86]. For ClamAV, files under a home directory were scanned using two types of virus-scanning programs, *clamscan* and *clamdscan*/*clamdscan*. In addition, the virus databases used by these two programs were initialized and updated using the *freshclam* command-line program. Tripwire checked the integrity of the target VM's file system, and initialized and updated the database used for integrity checking. Snort logged access to the port number used by Apache. As programs for handling by strace and Systrace, three command-line programs, *ps*, *ls*, *cp* were executed. Requests to fetch static content were sent to thttpd, which was monitored by strace and Systrace. The values for strace and Systrace listed in Table 2.2 are values for system calls executed by these programs and do not include the system calls executed by their target programs.

Controlling a server program in collaboration with a security program: For this case, as shown in Figure 2.7, Sendmail was the server program and ClamAV (*clamav-milter*, clamd) was the security system. When an e-mail containing an attachment file was sent, Sendmail sent a request to scan the attachment file to an instance of clamav-milter running on the same target VM. Then, clamav-milter forwarded the virus

Table 2.2: Numbers of executed system call types for each target program

application name		IA-32	AMD64
thttpd		41	47
Apache		57	68
Sendmail		67	74
ClamAV	freshclam	34	39
	clamscan	33	32
	clamdscan	21	23
	clamd	45	49
	clamd-milter	61	58
Snort		32	41
Tripwire		48	55
strace		22	26
Systrace		40	N/A

scan request to an instance of clamd running on another target VM

Table 2.2 summarizes the numbers of executed system call types. This tabulation shows that a target program should have a security policy specified for each architecture on which it runs.

2.6.2 Effectiveness Against Attacks on Security Systems

ShadowVox enhances the trustworthiness of security systems based on system call interposition. Even if a program running with administrator privileges is hijacked inside a target VM, the compromised program cannot disable ShadowVox's control mechanisms for the target programs. This was demonstrated in two scenarios using the Apache Web server as the program protected by a security system and the ProFTPD [85] ftp server as the compromised program with the administrator privilege.

First, we assumed a scenario in which the security system was *not* protected by ShadowVox. Systrace [86] was selected as the security system running in the target VM. Systrace's security policy specified that Apache must not read files, except for specific ones. The specific files included configuration files under the `/etc/apache2` directory and 10 files, under the `/var/www` directory, containing static content. Although Apache permitted access to files under `/var/www` from the Internet, Systrace forbade Apache from disclosing files newly added under the directory. At this point, ProFTPD was hijacked from another physical machine by exploiting the buffer overflow vulnerability (CAN-2003-0831). The *sh* shell program was launched with the administrator privilege and

used to copy the `/etc/password` password file to `/var/www`, which also required the administrator privilege. After that, Systrace's security policy was tampered with so that all files under `/var/www` could be accessed from the Internet. Consequently, although Systrace still controlled Apache, the `/etc/password` content could be obtained though a Web browser.

Second, we assumed a scenario in which a security system, i.e., a control program for Apache, was protected by ShadowVox. The control program monitored Apache by using a file containing the same policy rules as Systrace. As in the Systrace scenario, ProFTPD was hijacked, *sh* was launched, and `/etc/password` was copied to `/var/www`. Thereafter, we attempted to tamper with the policy file used by the control program. However, this failed since the policy file was managed from outside the target VM, i.e., inside the control VM. As a result, ShadowVox's control mechanism was not subverted, and `/etc/password` content could *not* be obtained.

2.6.3 Performance Impact

Setup

ShadowVox was tested on the IA-32 and AMD64 architectures. The IA-32 architecture had an Intel Pentium 4 3.0-GHz processor with hyper-threading enabled, 1 GB of RAM, and a 1-Gbps NIC. The AMD64 architecture had dual-core AMD Opteron 2.8-GHz processors, with 8 GB of RAM and a 1 Gbps NIC. In the experiments, the control VM and one target VM were run on top of the Xen VMM. Para-virtualized Linux OS kernels ran inside both VMs. When a request to make a policy decision is sent to a control program, the control VM is not always running on the VMM since the VMM scheduler determines which processors (physical CPUs) are assigned to which VMs. In the experiments to force the control VM to always run on the VMM, the control VM and target VM were configured to use different processors. On the IA-32 architecture, the control VM and target VM were both configured with one processor and one virtual CPU, whereas the control VM and target VM were both configured with two processors and two virtual CPUs on the AMD64 architecture. On the IA-32 architecture, the memory sizes of the control VM and target VM were 512 and 256 MB, respectively. On the AMD64 architecture, the memory sizes of the VMs were both 1 GB.

Microbenchmarks

To investigate how the system call interception and communication between the VMM and the control VM contribute to the entire overhead, microbenchmark programs were run on ShadowVox. Execution times were compared by using the following settings.

ShadowVox (SYSENTER): Para-virtualized Linux running on IA-32 (system calls are invoked with SYSENTER instruction)

ShadowVox (INT0x80): The same setting as the above except that system calls are invoked with the software interrupt

ShadowVox (SYSCALL): Para-virtualized Linux running on AMD64 (system calls are invoked with SYSCALL instruction)

Xen: Para-virtualized Linux running on IA-32 or AMD64

Xen+ptrace: Para-virtualized Linux running on IA-32 or AMD64 with an extension for intercepting system calls using `ptrace`

Linux: Native Linux running on IA-32 or AMD64

Linux+ptrace: Native Linux running on IA-32 or AMD64 with an extension for intercepting system calls using `ptrace`

The ShadowVox settings (SYSENER, INT0x80, and SYSCALL) were further divided into two cases based on the response action in the security policy: `allow` and `skip`. Specifically, the `allow` case means that the SV-core notified the control program of interception, while the `skip` case means that it did *not* notify the control program.

The microbenchmarks were four programs described below. Each program invoked 10,000 sets of system calls.

getpid: This program repeated invocation of the `getpid` system call.

open: This program opened a file in a home directory and immediately closed it, with the `open` and `close` system calls.

socket: This program created a TCP/IP socket the and immediately closed it, with the `socket` and `close` system calls. The system call handling for network operations differs between the IA-32 and AMD64 architectures, as described in Section 2.5.4.

fork: This program forked a process, and the parent process waited for the child process. The child process immediately completed its execution with the exit_group system call. A given security policy specified that the child process also had to be controlled. The parent process repeatedly invoked the `fork` and `waitpid` (IA-32) or `wait4` (AMD64) system calls.

The experimental results on IA-32 are indicated in Figures 2.8, 2.9, 2.10, and 2.11. The experimental results on IA-64 are shown in Figures 2.12, 2.13, 2.14, and 2.15. The

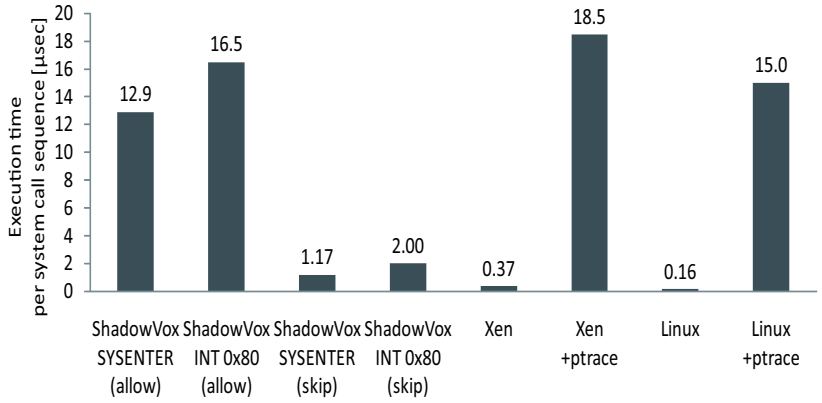

Figure 2.8: `getpid` microbenchmark results on ShadowVox, Xen, and Linux (IA-32)

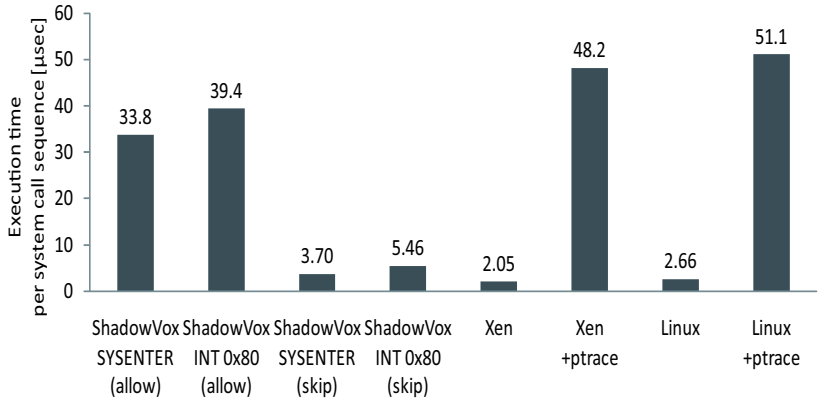

Figure 2.9: `open` microbenchmark results on ShadowVox, Xen, and Linux (IA-32)

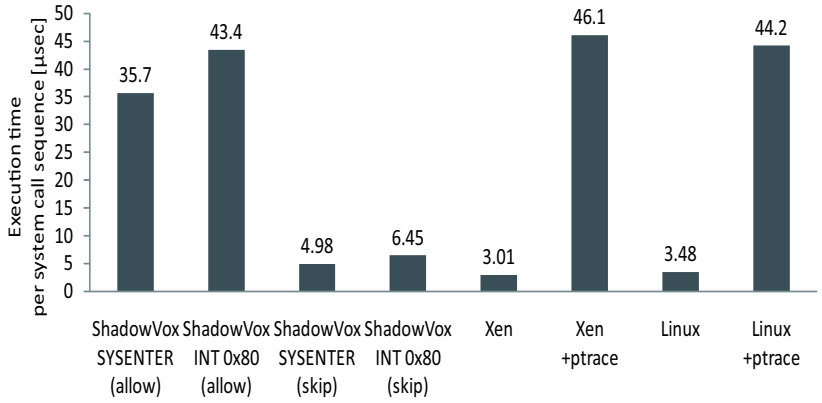

Figure 2.10: `socket` microbenchmark results on ShadowVox, Xen, and Linux (IA-32)

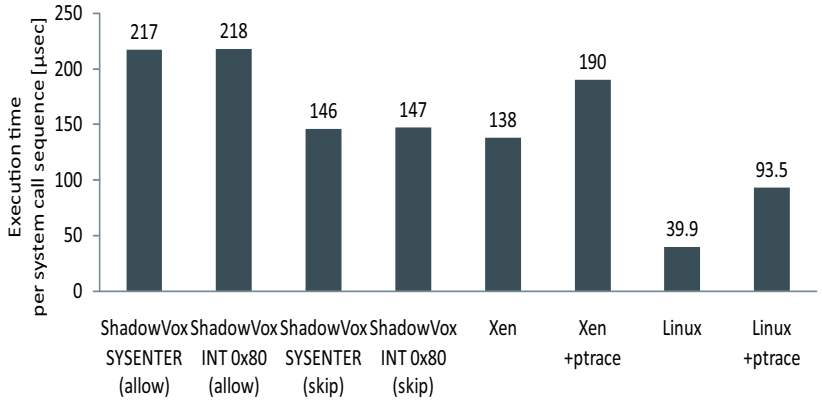

Figure 2.11: `fork` microbenchmark results on ShadowVox, Xen, and Linux (IA-32)

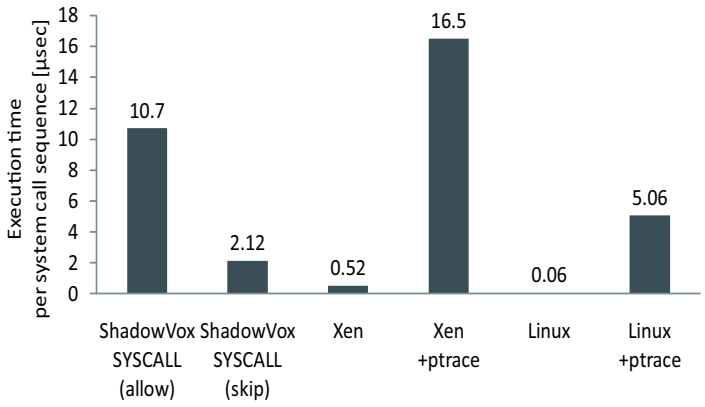

Figure 2.12: `getpid` microbenchmark results on ShadowVox, Xen, and Linux (AMD64)

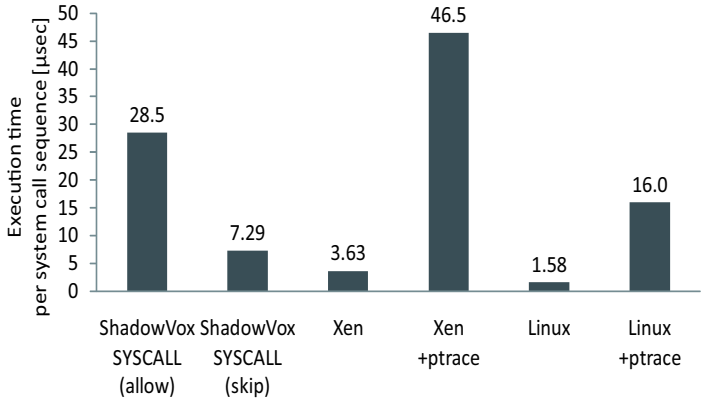

Figure 2.13: `open` microbenchmark results on ShadowVox, Xen, and Linux (AMD64)

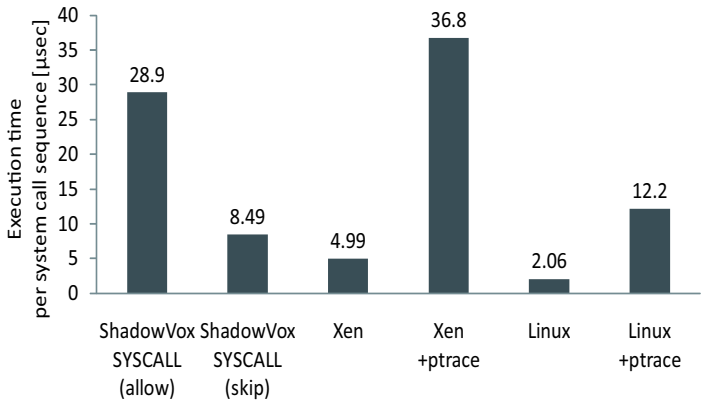

Figure 2.14: `socket` microbenchmark results on ShadowVox, Xen, and Linux (AMD64)

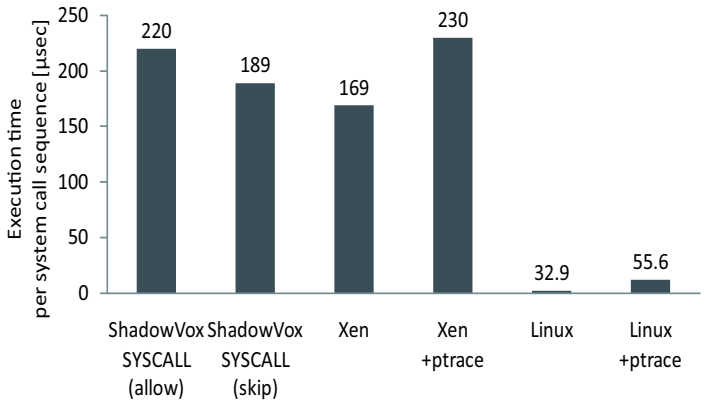

Figure 2.15: `fork` microbenchmark results on ShadowVox, Xen, and Linux (AMD64)

figures indicate the average execution time per system call sequence, so that lower values are better. The results show that the overhead incurred by ShadowVox, for both the `allow` and `skip` cases was lower than that incurred by Xen+ptrace except for fork in the `allow` case. This indicates that ShadowVox controlled executed system calls with lower overhead as compared with an approach that monitors processes from inside a target OS kernel, with a process tracing facility provided by the target OS kernel. The main reason for the lower overhead is that the number of notifications to the control program is less than that of the `ptrace` monitor program. For example, for a single `getpid` execution, SV-core either notified the control program of interception once, i.e., at the system call entry (`allow` case), or it did not notify the control program at all (`skip` case). In contrast, the `ptrace` monitor program performed notification twice, i.e., at the system call entry and exit. The same consideration explains why fork in the `allow` case had overhead close to that incurred by Xen+ptrace, since notification was also performed for system call interception at the exit. The difference in overhead between Xen and the `skip` case was much smaller than the difference between Xen and the `allow` case. This indicates that the overhead due to system call interception was much smaller than that due to execution control of system calls including communication between different VMs via the VMM. A user can reduce the overhead incurred by ShadowVox by specifying the `skip` action for system calls that usually do not require control, such as `poll` and `gettimeofday`.

Application Benchmarks

We expected that the overhead incurred by controlling application programs would not be as high as that incurred by controlling microbenchmark programs. To confirm this, we measured the overhead imposed on the performance of a Web server program and security systems. The application programs were run on ShadowVox to record their executed system calls, and their security policies were described. To obtain practical security policies, the policies were generated for each application program by using Systrace, and the generated policies were used to describe the security policies for ShadowVox. `allow` and `skip` were specified as the response actions for rules generated by Systrace that had policy conditions for system call arguments and for those with no conditions, respectively. Furthermore, the policy rules generated by Systrace were modified to allow execution of the `access`, `stat`, and `lstat` system calls to continue without notifying the target program. Specifically, `skip` was specified as the response action for these three system calls.

The application programs included the following:

Web server *Apache*: Requests were sent to fetch two types of static content (1- and 100-KB files) and dynamic content (CGI) to Apache by using the *ApacheBench* benchmark program. The CGI program obtained information on the platforms on which

Apache and ApacheBench were running and sent that information to Apachebench. Apache ran on the target VM while ApacheBench ran on a different machine in the same LAN. The platform running ApacheBench consisted of a Pentium 4 3.2-GHz processor with hyper-threading enabled, 2 GB of RAM, and a 100-Mbps NIC. The number of requests was a power-of-two number from 1 to 1024. Appendices B.1 and B.2 describe the security policies for Apache running on the IA-32 and AMD64 architectures, respectively.

Anti-virus tool suite *ClamAV* and file system integrity checker *Tripwire*: For ClamAV, we scanned files for viruses by using two types of programs: *clamscan* and *clamd*. clamscan is a command-line virus scanner, which reads the virus databases every time it is executed. clamd is a virus scanner daemon, which performs virus scans according to requests received from the *clamdscan* client. clamd reads the virus databases at start-up or when update requests for the databases are received from users. ClamAV's six test files, including five infected files, were scanned. For clamd, the system calls executed by clamdscan were also controlled. For Tripwire, the file system of the target VM were scanned for changes.

Web server *thttpd* in combination with network-based intrusion detection system *Snort* or sandboxing system *Systrace*:
thttpd was executed on the target VM, while ApacheBench was executed on a different machine, which was the same physical machine used in the above Apache case, in the same LAN. ApacheBench issued requests to fetch a 1-KB file. The number of requests was a power-of-two number from 1 to 1024. Snort monitored access to the port used by thttpd, while Systrace monitored the system calls invoked by thttpd. Snort was controlled in two cases based on which programs were targeted: Snort, or both Snort and thttpd. Systrace was controlled according a security policy specifying that thttpd did not need to be controlled. (The policy specified `detachProc` in the `execByPtracingProc:` field. `detachProc` and the `execByPtracingProc:` field are explained in Appendix A.)

To compare with an existing security system based on system call interposition, the throughput was also measured for an instance of Apache whose behavior was controlled by Systrace running in the target VM on the IA-32 architecture. In this experiment, Systrace controlled Apache by using the `ptrace` system call. Systrace 1.6e does not support the AMD64 architecture.

Experimental results for the Web server on IA-32 are shown in Figures 2.16, 2.17, and 2.18, and experimental results for the Web server AMD64 are shown in Figures 2.19, 2.20, and 2.21. The values indicate the processing time per request, so that smaller values are better. The results show that the performance degradation due to ShadowVox was lower than that due to Systrace. This indicates that system call interception by para-

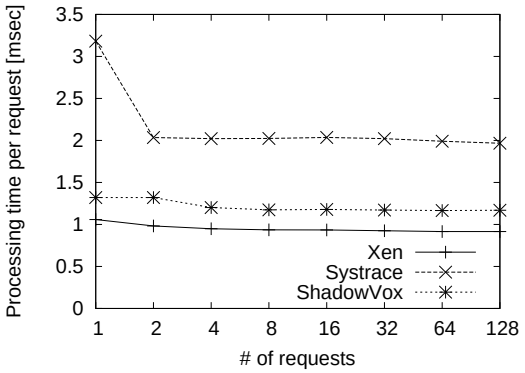

Figure 2.16: Web service throughput on ShadowVox, Xen, and Systrace for IA-32 (1-KB file)

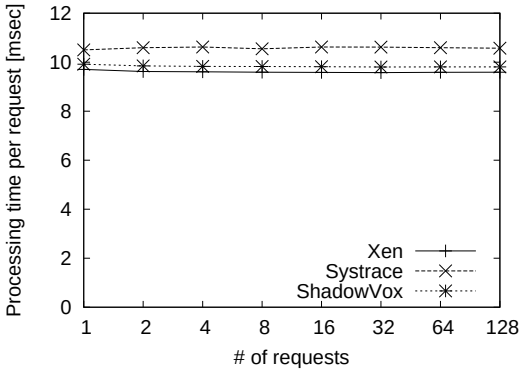

Figure 2.17: Web service throughput on ShadowVox, Xen, and Systrace for IA-32 (100-KB file)

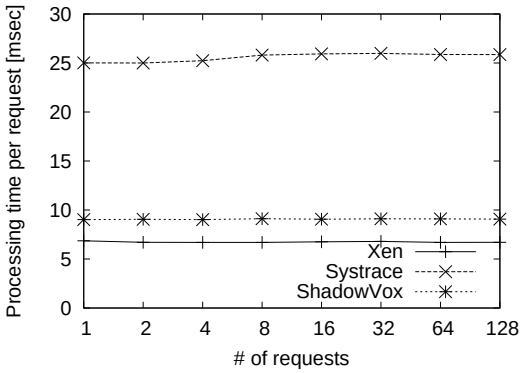

Figure 2.18: Web service throughput on ShadowVox, Xen, and Systrace for IA-32 (CGI)

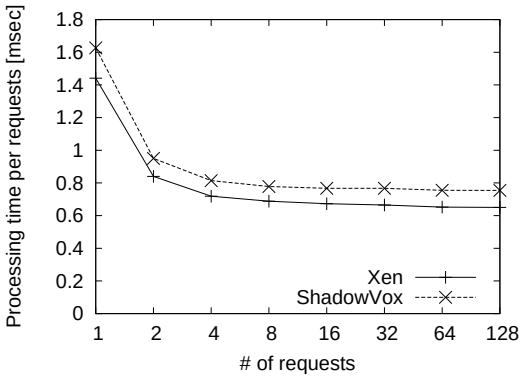

Figure 2.19: Web service throughput on ShadowVox and Xen for AMD64 (1-KB file)

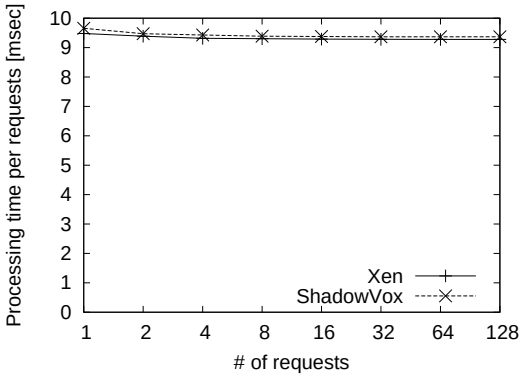

Figure 2.20: Web service throughput on ShadowVox and Xen for AMD64 (100-KB file)

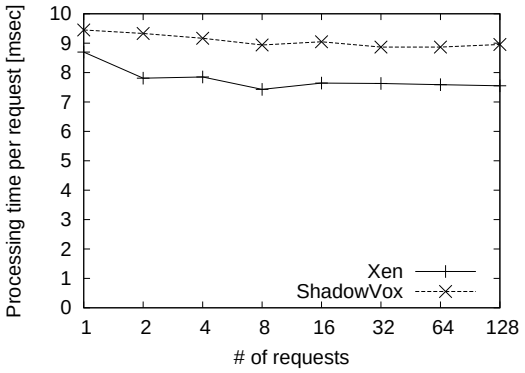

Figure 2.21: Web service throughput on ShadowVox and Xen for AMD64 (CGI)

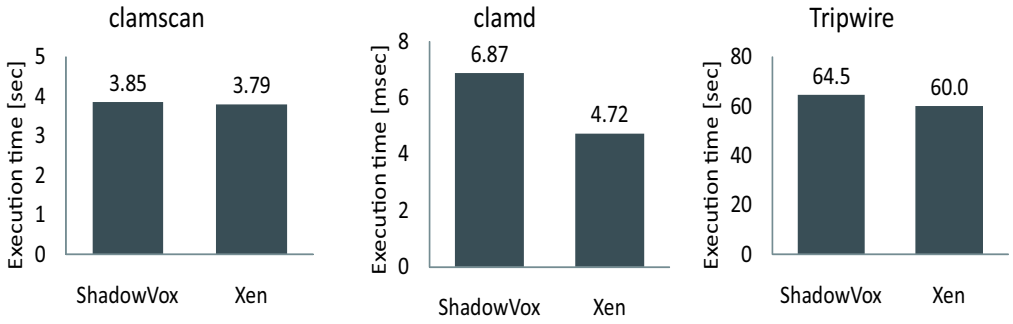

Figure 2.22: File Checking on ShadowVox and Xen (IA-32)

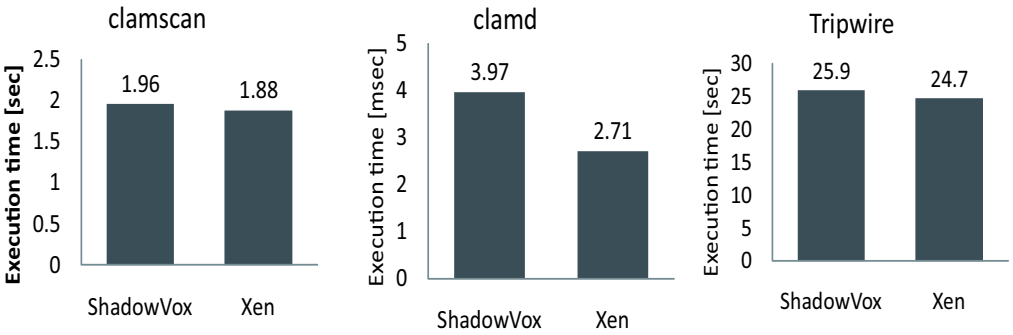

Figure 2.23: File Checking on ShadowVox and Xen (AMD64)

virtualization of the VMM has less impact on performance than that of the `ptrace` process tracing facility. Systrace also provides a kernel patch for reducing the overhead due to system call interception. However, this kernel patch was not applicable to the target OS kernel since it was a para-virtualized OS kernel with the source code modified from that of original OS kernel. In contrast, ShadowVox controlled the target program with lower overhead than that incurred by `ptrace`, without modification of the target OS kernel source code.

Figures 2.22 and 2.23 summarize the experimental results for the security systems on IA-32 and AMD64, respectively, while Figures 2.24, 2.25 show those for the Web server in cooperation with Snort on IA-32 and AMD64, respectively. Lastly, Experimental results for the Web server in cooperation with Systrace on IA-32 are shown in Figure 2.26. As above, smaller values are better for the data in all of these figures. As expected, the results for application programs show that the performance degradation for these programs was lower than that for the microbenchmark programs. The main reason for the lower overhead is that system call execution accounts for a smaller potion of the entire execution of the application programs.

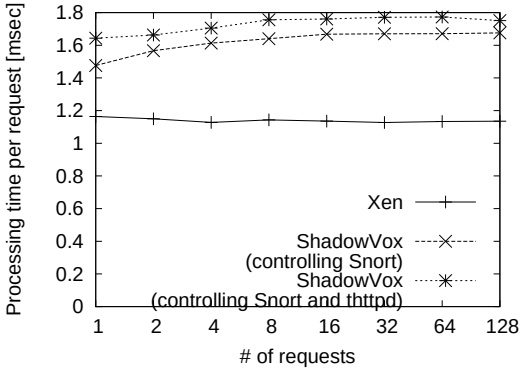

Figure 2.24: Web service throughput on ShadowVox and Xen for IA-32 (Snort)

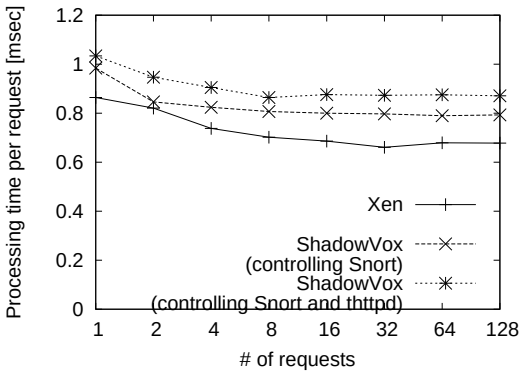

Figure 2.25: Web service throughput on ShadowVox and Xen for AMD64 (Snort)

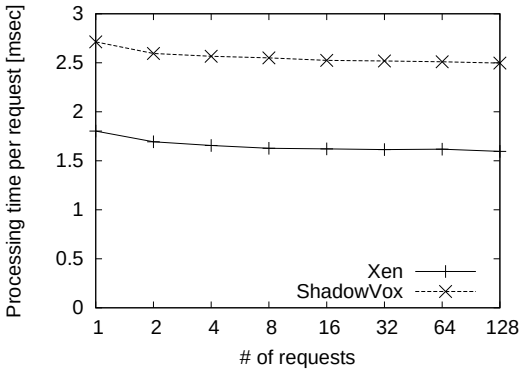

Figure 2.26: Web service throughput on ShadowVox and Xen for IA-32 (Systrace)

Table 2.3: Number of policy rules

application name		IA-32	AMD64
Apache		134	47
Apache	(CGI)	143	65
ClamAV	clamscan	54	64
	clamd	116	102
	clamdscan	41	40
Tripwire		81	76
Snort		93	94
Systrace		78	N/A

In addition, Table 2.3 summarizes the number of policy rules for each target program. The applied security policies were regarded as practical security policies.

Throughout this evaluation, ShadowVox was found applicable to several application programs and enhanced the security of a system based on system call interposition. We expect that the performance degradation due to ShadowVox would be acceptable to users.

2.7 Related Work

2.7.1 Enhancing Security from Outside VMs

Livewire [43] is an intrusion detection system that controls the behavior of a VM from outside. As in this work, Livewire uses information on the guest OS to understand the behavior in a monitored VM. However, there is a difference in which events in a monitored VM are intercepted by the VMM. Livewire intercepts write accesses to non-writable memory regions and accesses to a network device. On the other hand, ShadowVox intercepts system calls invoked by target processes. Therefore, Livewire cannot intercept instructions at a system call exit, since they are not privileged instructions and the their execution does not cause illegal access faults. In contrast, ShadowVox intercepts instructions at a system call exit by using dynamic binary instrumentation. Furthermore, there is a difference in the granularity of response actions for intercepted events. While Livewire takes response actions at VM granularity, ShadowVox takes response actions at process granularity.

IntroVirt [58] is a system that detects and responds to intrusions from outside a VM. It intercepts the execution of vulnerable code parts and executes vulnerability-specific predicates in the target system. While security checks in IntroVirt are performed accord-

ing to a vulnerability specification and a corresponding predicate, ShadowVox works according to a specification of how to check and control system calls. Since IntroVirt's protection mechanism depends on known vulnerabilities, it cannot prevent the behavior of target programs that include unknown, vulnerable code. In addition, there is a difference in the mechanism for obtaining execution states at OS-level abstraction outside a VM. Whereas IntroVirt obtains these states by using the guest OS kernel code, ShadowVox restores execution states from register values and data on the guest memory, as described in Section 2.3.1.

VMwatcher [55] is a system that detects malicious software (malware) by comparing the memory and virtual disk states reconstructed from outside with the memory and virtual disk states obtained from inside it. The states reconstructed from outside a VM are also used as input data to existing anti-malware systems. Whereas VMwatcher performs the malware detection when a user issues a request, ShadowVox controls the behavior of a target process when the target process executes a system call. Furthermore, ShadowVox controls the target process according to a security policy.

Asrigo et al. [17] proposed a honeypot system that collects information on file and network socket operations in a VM. Whereas that system collects the behavior of a guest OS kernel, ShadowVox controls the behavior of target processes.

XenAccess [81] is a library for monitoring raw memory access and disk I/O and restoring their inner states at OS-level abstraction from outside a VM. Unlike XenAccess, ShadowVox controls a monitored VM according to the behavior of target processes.

Lycosid [57] is a system that detects covertly existing processes in a VM. It uses the Antfarm [56] technique, which infers the existence and behavior of processes in a VM from the switching of process address spaces. Although Lycosid does not require information on a guest OS kernel, it cannot precisely identify process termination. Furthermore, it cannot control process behavior, file operations, and network operations.

2.7.2 Access Control at VMM and OS Layers

sHype [92] and NetTop [48] provide an infrastructure for controlling information flows and resource sharing between VMs. VMware ACE [104] provides a secure virtual desktop environment by managing packages consisting of a pair of a VM instance and a security policy. Terra [41] deploys security-sensitive programs in a separate trusted VM to protect these programs from other, untrusted programs. While the granularity of execution control in these systems is at the VM level, the proposed system controls execution at the process granularity.

There are also security-enhanced OSs based on mandatory access control (MAC) [10], such as SELinux [52], LIDS [4], TOMOYO Linux [46], and AppArmor [76]. Unlike approaches that enhance security from outside a VM, security-enhanced OSs can provide fine-grained control at OS-level abstraction. For example, such OSs can limit privileges

for root accounts. However, in practice, it is difficult to describe MAC-based security policies in consideration of the relationships between all the processes and files on a system [18, 53]. Consequently, users of security-enhanced OSs suffer from a tradeoff between the fine-grained control and the resulting complexity of the security policies. In contrast, ShadowVox simply makes attacks on security systems harder through isolation between VMs. There is also a difference in application for multiple execution environments, i.e., multiple target VMs. Whereas one security-enhanced OS must run for each target VM, one VM (i.e., the control VM) on ShadowVox can simultaneously manage and control multiple target VMs.

2.7.3 System Call Interposition

A number of security systems based on system call interception have been proposed. Most of these systems, including those in [37, 38, 45, 86, 98], run a security program on the same OS as the protected programs and potential malware. This makes it easier for malware to attack these security systems. On the other hand, with ShadowVox, the security system runs outside the guest OS. It is difficult for an attacker to stop the security system because the attacker cannot even observe it.

2.8 Summary

We have proposed ShadowVox, a system that enhances the security of VMs at application-program granularity by controlling the execution of system calls from outside the VMs. This "out-of-VM" scheme makes it harder to attack ShadowVox, since it runs outside the target VMs. ShadowVox controls processes according to a given security policy by using two basic techniques: VM introspection and dynamic binary instrumentation. VM introspection is a technique for identifying execution states and events in OS-level semantics from outside VMs by leveraging OS information on process management and system calls. We have clarified the OS information required for VM introspection. More concretely, we have revealed what prior knowledge on the OS information and what data relevant to this prior knowledge are required, how OS-level semantic views are identified using the information, and what the information depends on. Dynamic binary instrumentation is a technique for intercepting events such as system calls without modifying the source code of OS kernels. A security policy given by the user specifies which system calls are controlled and how they are controlled.

We implemented ShadowVox on the IA-32 and AMD64 processor architectures and confirmed that it controlled a diverse range of application programs, including server programs and security systems. In addition, we demonstrated that ShadowVox controlled the behavior of an Apache server instance under the condition that a ProFTPD

server instance with administrator privileges had been hijacked. The experimental results of the throughput for the Apache Web server. showed that system call interposition with ShadowVox has less impact on performance than that by a security system using the `ptrace` process tracing facility.

Future work on the proposed system will include the following considerations. First, the security policy approach should be refined. The current policy description is based on system call arguments and depends on processor architectures and OS kernel versions. To relieve users of the burdensome task of description, ShadowVox should support automated policy generation based on the execution logs of target programs. Another way to reduce the required description effort would be to raise the abstraction level of a security policy. A useful side effect of raising the abstraction level would be in cutting erroneous descriptions. As described in Section 2.5.3, we also plan to explore and implement an effective prevention mechanism against attacks using race conditions, such as symbolic link and time-of-check-to-time-to-check attacks.

Chapter 3

Protection for Application Data on Memory and Virtual Disk

3.1 Motivation

The effectiveness of security systems running on the OS and application layers is ensured as long as that no attacker can compromise the target OS kernel and privileged programs running in the same execution space. However, the target OS kernel and privileged programs are vulnerable [1, 5, 6, 7]. In addition, the libraries used by such programs can also be vulnerable [3, 5, 7]. Therefore, if an attacker exploits these vulnerabilities, the attacker can subvert or disable the protection mechanism designed at the OS and application layers. Since Janus [45] and Systrace [86] control the behavior only of programs specified by the user, an attacker can hijack programs whose behavior is not controlled by these security systems. Using such hijacked programs, the attacker can illegally operate on target program data on memory and disk after tampering with the security policies of the security system. For example, the attacker can induce leakage of confidential information on a program controlled by the security system. The attacker can also tamper with the code regions of target programs controlled by the security system to modify their control flow.

A VMM-based approach is useful for protecting target program data on memory and disk from untrusted programs, including target OS kernels, running in the same execution space as the target programs. Security systems based on a VMM can isolate target program data in one VM from other untrusted programs running in another VM. For example, Terra [41] runs a target program outside the target VM. However, Terra's approach increases the amount of computing resources consumed and the number of inter-VM context switches, since users must assign one VM per target program to protect the target program data. If there is more than one target program, the impact of resource consumption and inter-VM context switches are greater because the user must assign

one VM to each target program to isolate all of the target programs.

Other examples are Nizza [100] and Proxos [103]. These systems isolate the security-sensitive components of a target program, such as a key authentication mechanism, from the rest of the target program components and other untrusted programs, in order to run the target program with a smaller trusted computing base (TCB). However, these security systems lose their compatibility with the existing target program binaries in addition to requiring extra inter-VM context switches. Hence, the user must modify the source code of existing target programs to use them with these systems.

In this chapter, we present Shadowall to achieve two goals *simultaneously*. The first goal is to solve the problem of security systems running in the OS and application layers, i.e., to prevent compromised programs in the same execution space as target programs from inducing leakage and tampering with target program data on memory and disk. The second goal is to address the problems with existing VMM-based security systems, i.e., to overcome the problems with resource consumption, inter-VM context switches, and target program binary compatibility. To achieve these two goals, a VMM on Shadowall controls memory management operations and interposes system call procedures in cooperation with a different VM from the target VMs. Shadowall can protect the confidentiality and integrity of target program data when target programs run inside target VMs.

We assume that a user applies Shadowall to existing server and client programs as target programs without modifying their source code. In addition, the target programs include proprietary programs that governments and corporations plan to develop. Such proprietary programs require preservation of not only their integrity but also their confidentiality. Application of Shadowall to proprietary programs prevents attackers from analyzing their content.

The rest of this chapter is organized as follows. Section 3.2 describes the threat model. Section 3.3 presents the design of the proposed system, followed by a description of a prototype implementation in Section 3.4. Section 3.5 discusses the evaluation of the system. Finally, Sections 3.6 and Section 3.7 discuss related work and summarize the chapter, respectively.

3.2 Threat Model

Shadowall prevents attackers from inducing data leakage and tampering with target program data. The target program data include memory data in the user space and virtual disk data such as executables, configuration files, and database files. Attackers are assumed to maliciously manipulate memory data through APIs for controlling processes, such as the ptrace system call on Linux, loadable kernel modules, and special devices such as /dev/mem and /dev/kmem.

A VMM and a trusted VM isolated from the target VMs are the trusted computing base (TCB) of Shadowall. Meanwhile, target VMs are not part of the TCB; that is, Shadowall does not trust any programs running inside target VMs.

If an attacker exploits vulnerabilities of OS kernels and privileged programs, the attacker can maliciously manipulate target program data regardless of whether the target program is vulnerable. For example, an attacker can tamper with target program data on memory, such as control data and non-control data [25], to alter the behavior of a target program. Control data include return addresses saved on the stack and function pointers. Non-control data include variables used as effective user IDs and authenticated flags.

An attacker can also tamper with such data to circumvent or subvert the protection provided by security systems. For example, the attacker can tamper with files related to a security system controlling a target program, such as configuration, policy, and database files. Then, the attacker can leak target program data that had originally been confidential because the protection of the security system has already been disabled. Furthermore, to make an attack more difficult to detect, the attacker can transiently modify target program data on memory and a virtual disk. With mimicry attacks [79, 108], an attacker can evade sophisticated detection by a security system by tampering with target program data.

To protect target program data, we need to protect both memory data and the virtual disk data. For example, an attacker can modify the control flow of memory data if its data are not protected. An attacker can similarly leak or tamper with the content of target program files on a virtual disk if its data are not protected.

Shadowall identifies target program contexts by using the VM introspection technique described in Section 2.3.1. If a compromised OS kernel falsifies target program data managed in the kernel (e.g., process and memory management data), the target program might run incorrectly. Even so, Shadowall can prevent the compromised OS kernel from leaking and tampering with the target program data on memory and a virtual disk.

On the other hand, certain threats are outside the scope of Shadowall. The first threat consists of to hijacking a target program by exploiting its own vulnerability to buffer overflows. To prevent this kind of attack, we assume that Shadowall is used with a security system that controls the behavior of a target program from outside a target VM, such as a sandboxing and intrusion prevention systems (e.g., ShadowVox, a VMM-based sandboxing system that controls the system calls invoked by target programs, as described in Chapter 2). A second threat outside Shadowall's scope is modification of kernel space data related to a target program in order to change its behavior. This kind of attack is often accomplished using kernel-level rootkits. One example is gathering network data related to a Web server, where an attacker captures the network data at the kernel layer and forwards it to the attacker's malicious program. Another example is attacks such as those on denial of services (DoSs), where an attacker removes target processes from a scheduling list and remains at DoSs pending system call events.

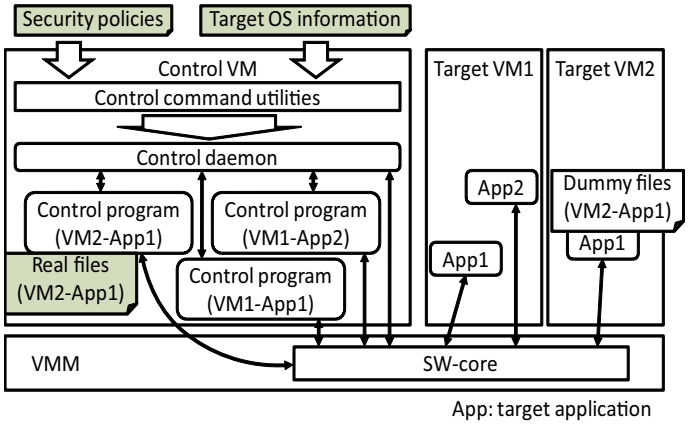

Figure 3.1: Shadowall: system for protecting memory and virtual disk data involved with target application programs

3.3 Design

3.3.1 Overview

Figure 3.1 outlines the structure of the proposed system, Shadowall. The proposed data protection mechanisms are incorporated into a VMM and a VM controlling the target VMs, called a *control VM*.

The mechanism for protecting memory data includes an in-VMM component called *SW-core*. The mechanism for protecting virtual disk data, on the other hand, includes both SW-core and a *control program* residing in the control VM. A control program is generated for each target program instance by a *control daemon*. Furthermore, a *command utilities* are provided for controlling target program data from outside a target VM in the control VM.

A security policy and information on a target OS kernel, called *target OS information*, are used to control a target program from outside the target VM. Since the control VM manages the security policy and target OS information, if an attacker takes over the target VM, the attacker cannot tamper with them.

Files that include target program data used at runtime are also managed inside the control VM. Such a file is referred to as a *real file*. However, a file associated with a real file is deployed inside a target VM. The associated file is referred to as a *dummy file*. The dummy file is deployed in a target VM to maintain consistency with the file operations performed by a target OS kernel.

The target program data are controlled at process granularity by using the target OS information, as with the sandboxing system, ShadowVox, described in Chapter 2. The

target OS information includes information on kernel objects for process management and on system calls. Shadowall controls the execution state of a target program from outside the target VM by using the target OS information. A support program is also provided to automatically generate target OS information that depends on the OS kernel image. Hence, a target OS kernel requires that its design, in terms of processes and system calls, be available. For example, the target OS can be a UNIX-like OS.

In deploying Shadowall, the target VM user or the control VM administrator determines which target programs to place under control, and which target files protect, in which parts of the protected memory area. These decisions are registered in the proposed system by using the control command utilities and the security policy.

3.3.2 Usage

To control the target program data from outside a target VM, the following command-line programs are provided.

- `mkdummy`: This is a command to create dummy files for a target program. The argument is a file describing the associated real files. Whenever a target file is updated, the user creates the associated dummy file. The dummy files created by this command are deployed inside a target VM.

- `sw_vconf`: This is a command to register target OS information. The arguments include a target OS kernel image and three files necessary to control the target program data. One file stores information on kernel objects related to process management in a target OS kernel. The second file stores information on system calls. The third file stores information on pairs consisting of a symbol and a virtual address for intercepting the entry and exit of system calls. Shadowall manages one set of target OS information for each target OS kernel image. A hash value generated from the target OS kernel image is used as an identifier for the corresponding target OS information.

- `sw_vcntl`: This is a command to associate a target OS kernel with target OS information. The arguments are a target OS kernel image and an identifier for the target VM. After this command is executed, Shadowall can execute control at process granularity from outside the target VM.

- `sw_vstart`: This is a command to start control of target program data. The arguments are a target VM ID, the name of a target program, and a security policy for the target program.

Figure 3.2 shows an example of procedures related to `target_prog` residing in a target VM whose ID is 1; these procedures are followed until data control starts. First,

```
[cvm] $ mkdummy target_files.txt
[cvm] # sw_vconf targetOS.img \
      targetOS_info.txt syscall_info.txt syscall_hook.txt
[cvm] # sw_vcntl 1 targetOS.img
[cvm] # sw_vstart 1 target_prog policy.txt
```

Figure 3.2: Usage example in Shadowall

the user executes `mkdummy` to create dummy files. The target files, such as executable and configuration files, are specified in the `target_files.txt` file. Next, the user notifies the control daemon of the target OS information for the target OS kernel image by using `sw_vconf`. The user provides arguments consisting of the target OS kernel image file, `targetOS.img`, and the three files for controlling the target data from outside the target VM, as described above. `targetOS_info.txt` contains the information on kernel objects for process management, `syscall_info.txt` contains the information on system calls, and `syscall_hook.txt` contains the information on symbol / virtual address pairs for intercepting the entry and exit of system calls. Next, by executing `sw_vcntl` with arguments consisting of the target VM ID `1`, and the target OS kernel image `targetOS.img`, the user associates the target VM instance with the target OS information registered through `sw_vconf`. Finally, the user executes `sw_vstart` to send a request to start controlling the target data related to `target_prog`. The user provides `sw_vstart` with arguments consisting of the target VM ID `1`, the name of the target program `target_prog`, and the security policy file `policy.txt`. After this, whenever `target_prog` starts, its memory and virtual disk data are protected according to the security policy.

3.3.3 VMM-based Protection Mechanisms

Memory Data Protection

Paging is adopted as a memory management scheme in many commodity OSs such as Windows and Linux. In paging, the OS kernel makes one process share page tables accessed in different execution modes, namely, the kernel and user modes. Thus, for a virtual address, a process running in user mode accesses the same physical page as when it is running in kernel mode.

In the VMM architecture, unlike an OS kernel running directly on hardware, the VMM manipulates address translation into the physical addresses of VMs. To protect target data in the user space, the proposed system extends the memory management scheme to the VMM layer, so that the VMM can interpose in memory management and

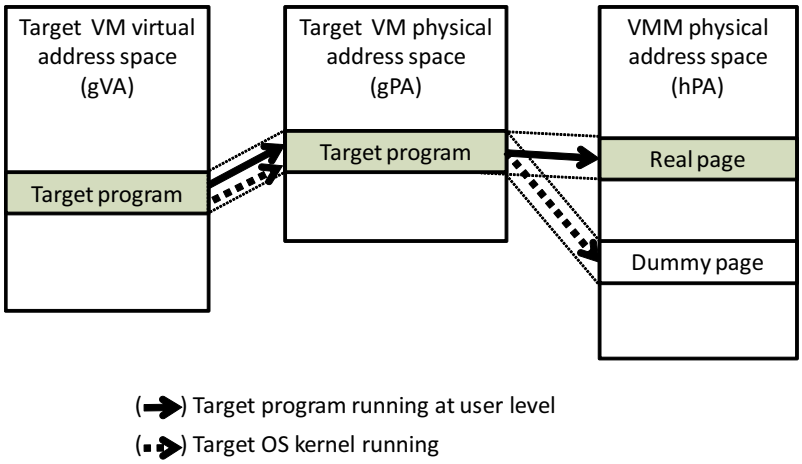

Figure 3.3: Physical page multiplexing according to execution modes

multiplex physical pages. The multiplexing of physical pages depends on the execution mode of a target program. As indicated in Figure 3.3, the VMM presents physically different memory mapping from a guest virtual address (gVA), to a host physical page (gPA) according to the execution mode. The term *real page* represents the physical page accessed when the target program runs in user mode. In contrast, the physical page accessed by the target OS kernel is a *dummy page*. After memory multiplexing, the physical addresses accessed by the target OS kernel remain the same as before. Switching between the user and kernel modes occurs at the entry and exit of exceptions and the entry and exit of interrupts. Switching also occurs at the entry and exit of system calls.

Since a VMM running at a higher privilege level than an OS kernel controls the final address translation of a target VM, malicious address translation can be disabled at the VMM layer. Even if a compromised OS kernel attempts to leak and tamper with target program data, Shadowall forces it to access the corresponding dummy pages.

A target OS kernel accesses user space data, such as file path names and objects for network connections, in certain system call procedures. Therefore, Shadowall needs to adjust the user space data so that the target OS kernel can appropriately cope with the user space pointer of a system call argument. To achieve this, the VMM manipulates target program data so that only the memory region used by a system call can be temporarily shared between the content of a dummy page and the content of the associated real page. The sharing duration is from a system call entry to a system call exit.

This data sharing is a limitation on the proposed memory protection scheme. In other words, it gives an attacker the chance to gather and modify user space data to which system call arguments point. Note that the attacker cannot leak and corrupt user space

data involved with the files whose content is protected by Shadowall, i.e., the*target file*s. The details of this are described later in this section.

Unlike existing systems [26, 66, 119] that use cryptographic protection to multiplex protected pages logically, Shadowall multiplexes protected pages physically. Whereas this approach requires extra physical pages for multiplexing, the other approach introduces additional overhead due to encryption and decryption. In a target VM configured with multiple virtual CPUs, target processes cannot run in parallel to exclusively manipulate decrypted pages.

Virtual Disk Data Protection

The content of target files is protected through VM isolation and control from outside the target VM. The content of a real file is managed outside the target VM. Control of the content of a target file depends on which process manipulates it. target files are specified by the security policy. When a control program manipulates a target file, the in-VMM component, SW-core, and a control program in the control VM emulate the procedures of the system call involved in a file operation to update the content of the real file. Since system call emulation is not processed through the target OS kernel, the user space data need not be shared between the real page and the dummy page, as described in Section 3.3.3. The proposed file protection scheme protects the contents of a target file on memory in combination with the memory protection scheme described in Section 3.3.3.

System calls related to operations on file content, such as `open` and `read`, are emulated, while those related to operations on file attributes, such as `stat` and `fchown`, are not emulated. This approach has two advantages. First, it reduces the number of kinds of emulated system calls. Second, it does not require emulating changes in user IDs or group IDs related to a target file in the control VM. User and group IDs are assumed to be controlled by a combination of Shadowall and a sandboxing system to control them, described in Chapter 2.

Unlike the proposed scheme, there is a VMM-based approach for protecting a target file in combination with NFS. However, this combination approach with NFS has two main problems. First, an NFS client transfers the content of a target file to a target OS kernel through file manipulation. A compromised target OS kernel can leak and tamper with the content during the NFS data transfer. In contrast, the proposed scheme can protect the target file content because the file transfer is processed outside the target VM. Second, NFS access control is enforced through user and group IDs, whereas Shadowall controls file operations according to which processes execute.

Shared Data and Network Data Protection

With regard to protecting shared data related to a target program, the user is assumed to specify that all programs accessing the shared data are target programs. Shadowall does not hide shared library data, such as that in libc libraries, on the memory and virtual disks because this shared data are also used by other programs that Shadowall does not control. If Shadowall hid the shared data, the uncontrolled programs would crash. Instead, Shadowall prevents attackers from tampering with the shared data by requiring target programs to access the shared data stored as a target file in the control VM instead of the shared data in a target VM.

Since network data are accessed through procedures within a target OS kernel, attackers can maliciously access them. To protect private network data related to a target program, the target program is assumed to use a secure communication mechanism in the application layer, such as SSL. Whereas private network data are protected by SSL, Shadowall protects data related to the SSL certificate and the SSL private key. Meanwhile, public network data such as Web content is not assumed to be protected because users can confirm that content through their Web browsers.

3.3.4 Security Policy

Figure 3.4 lists the syntax of the proposed security policy. It specifies which parts of memory regions are protected, which files are target files, and how the target data are placed under control.

The `executable:` field is followed by *pathOutsideVM*, the executable path name of a target program in the control VM. The target memory and virtual disk data are specified in the parts following the `shadowMemoryFile:` field.

For memory data protection, the user specifies whether all target program data are multiplexed (`all`) or parts of the data are multiplexed (`partially`). For partial multiplexing, the user specifies the parts. The multiplexing granularity includes an OS segment ("KernSeg"), the memory-mapped region (`mmapRegion`), and the section (`ELFSec`) and segment (`ELFSeg`) of the ELF executable. The specifications of the OS segment include the code region (`text`), the static and dynamic regions (`data` and `brk`, respectively), the stack region (`stack`), and the region for environment variables and command-line arguments (`env` and `arg`, respectively). In partial multiplexing, the user can also specify which parts are read-only regions ("ReadOnlyRgn"). Such read-only regions are not multiplexed at the VMM layer because they are protected by prohibiting write operations to a read-only region.

The "ShadowFileSpec" part indicates target files, with *outsideVM* and *insideVM* corresponding to real and dummy files, respectively. The "Permission" part indicates the permission for a target file.

PolicyFile	→	`executable :` *pathOutsideVM* `shadowMemoryFile`:ShadowMemFile CalleeSpec*
ShadowMemFile	→	`all` ShadowFileSpec* | `partially` PartSpec+ ShadowFileSpec*
PartSpec	→	`kernelSeg :` KernSeg+ | `mmapRegion` | ReadOnlyRgn | `ELFSec :` *secName*+ | `ELFSeg :` *segNum* +
KernSeg	→	`text` | `data` | `brk` | `stack(DOWN` | `UP,` *size*`)` | `env` | `arg`
ReadOnlyRgn	→	`readOnlyRegions :` RORegion+
RORegion	→	`text` | `mmapRegion` | `ELFSec :` *secName*+ | `ELFSeg :` *segNum*+
ShadowFileSpec	→	`shadowFile :(` `pathName(`*insideVM*`,` *outsideVM*`)` <Permission+>)+
Permission	→	`read` | `write` | `create` | `append`
CalleeSpec	→	`calleeExecve :` (*pathInsideVM policyFile*) +

Figure 3.4: Security policy syntax in Shadowall

```
executable : ./shadow/master_prog
 shadowMemoryFile : partially
    kernelSeg : data,brk,stack(DOWN,1GB),env,arg,mmapRegion
    readOnlyRegions : text
 shadowFile :
    pathName(/home/A/control.cfg,./target.cfg) <read>
    pathName(/home/A/control.log,./target.log) <read|append>
```

Figure 3.5: Sample security policy in Shadowall

Shadowall also provides a specification to control the `execve` system call invoked by a process that is currently controlled. The `calleeExecve:` field is followed by *pathInsideVM*, the executable path name of a target program inside a target VM, and *policyFile*, the security policy for controlling the callee program.

Figure 3.5 shows a sample security policy for the target program `master_prog`. While the code region of the memory data are protected by write-protection, the other regions are protected by the multiplexing scheme. While `master_prog` has permission to read the configuration file `control.cfg`, it also has permission to read and append the log file `control.log` in the virtual disk data.

The following use cases are assumed for partial multiplexing. The first case is protecting existing programs published on the Internet. No existing program data necessarily requires multiplexing. We need only prevent tampering with the write-protected

regions of existing programs, such as the code region. Here, the security policy specifies that write-protected regions are read-only, whereas multiplexed regions are writable. Another case is protecting particular regions containing confidential data. Specific areas include memory-mapped regions containing the contents of password files and private key files. The presence of the ELF sections or segments in self-developed programs is also assumed as the particular protected regions. The security policy explicitly specifies that regions containing confidential data be multiplexed. However, an attacker can modify the behavior of a target program by tampering with unprotected memory regions. Therefore, Shadowall should be used in combination with the sandboxing system described in Chapter 2 to mitigate against this kind of attack.

3.3.5 Features

Shadowall has the following main features.

Data protection scheme isolated from target VMs: Since the proposed memory and virtual disk protection schemes are based on control at a higher privilege level than that of the target VM, together with VM isolation, compromised programs residing in the target VM cannot subvert these protection schemes. Furthermore, since a target program runs inside a target VM, Shadowall does not require allocating extra VMs for each target program, which would increase the costs of resource consumption and inter-VM context switching.

Retaining compatibility with existing binaries: Unlike previous systems that partition a target program into trusted and untrusted parts [100, 103], Shadowall does not require modifying the source code of a target program or target OS kernel.

Protection control from outside target VMs: Previous systems [26, 91, 119] have protected target data by using an auxiliary utility inside a target VM. However, if an attacker takes over a program residing in a target VM, the attacker can delete the auxiliary utility or abuse it to hide a malicious program. In contrast, Shadowall prevents this attack because it does not require any auxiliary utilities. Control from outside target VMs also has an advantage in that target VM administrators can delegate the security maintenance of a target program to a control VM administrator.

Fine-grained, process-granular control: Execution control is only applied to target programs specified by the user of a control VM or a target VM. Unlike previous systems that perform control at VM granularity [48, 93], other programs in the same target VM are not affected.

3.4 Implementation

We have implemented Shadowall by using a para-virtualization version of Xen [20] 3.0.3 and Linux OS kernel 2.6.16 as the target OS kernel.

3.4.1 Multiplexing User Address Space

Management of Multiplexed Memory Regions

SW-core manages two types of information on the user space data of a target program. The first type is information on memory-mapped regions, including the range of virtual addresses currently assigned to the target program. Intuitively, this information corresponds to the `vm_area_struct` object of a Linux OS kernel. The second type is information on physical memory regions. This information includes, by the page, pairs of real physical and dummy addresses. It also includes the reference count for each physical address after control is started. In addition, when the target physical page is temporarily saved on the virtual disk, i.e., there is a transition to the *page-out* state, SW-core adds the entry of the bottom-level page table to this information. Whereas the information on memory-mapped regions is managed for each target process, the information on physical memory regions is managed for each target VM.

Target Page Table Update

To multiplex target physical pages, Shadowall provides a target program with separate page tables for each execution mode. On Xen for the AMD64 architecture, called *Xen-AMD64*, the VMM provides a process running in a VM with a separate top-level page table for each execution mode. The proposed multiplexing mechanism was implemented by extending Xen-AMD64.

When a page fault occurs in a multiplexed region, SW-core multiplexes the associated physical page. The page fault handling by SW-core needs to maintain consistency with the page fault handling by target OS kernels so that the page fault handling by SW-core cannot cause a target program or a target OS kernel to crash.

When SW-core intercepts a page fault at the exception entry, the page table entry has not yet been updated by a target OS kernel. Therefore, SW-core forces the target OS kernel to carry out page fault handling before its own page fault handling. Although the page fault handling by a target OS kernel applies to a dummy page, page fault handling by SW-core applies to a real page. Thus, a compromised target OS kernel cannot maliciously manipulate the content of a real page.

To force a target OS kernel to carry out page fault handling, SW-core first saves the value of the CR3 register and the virtual address of the instruction pointer when intercepting the entry of the page fault exception. The CR3 register value is stored at the vir-

Table 3.1: Page fault handling for each transition of page table entry

PPBA: physical page base address; #PF: page fault exception;
P: present bit; R/W: read/write bit; −: don't care condition

#PF entry			#PF exit			SW-core
PPBA	P	R/W	PPBA	P	R/W	handling
0	0	-	$\neq 0$	1	-	page allocation
$\neq 0$	1	0	$\neq 0$	1	1	copy-on-write
$\neq 0$	0	-	$\neq 0$	1	-	page-in

Table 3.2: Update of page table entry for each page fault exception

PPBA: physical page base address; #PF: page fault exception;
P: present bit; R/W: read/write bit; −: don't care condition

#PF entry			#PF exit			SW-core
PPBA	P	R/W	PPBA	P	R/W	handling
$\neq 0$	1	0/1	$\neq 0$	1	1/0	R/W update
$\neq 0$	1	-	$\neq 0$	0	-	page-out
$\neq 0$	0	-	0	0	0	page release

tual address of the top-level page table for the currently executing process. Next, SW-core continues to run the target VM to make the target OS kernel handle the page fault exception. Finally, when intercepting the exit of a page fault exception, SW-core determines whether the faulty page needs to be multiplexed by the values saved at the exception entry. If necessary, it multiplexes the faulty page.

Table 3.1 summarizes how SW-core handles each page table entry transition before and after page fault handling by a target OS kernel. In allocating a new page, SW-core allocates a real page and initializes it. The initialized data are not copied from the page allocated by the target OS kernel, i.e., the dummy page, but from a target file or the existing real page. In copy-on-write, the manipulation of multiplexing depends on the transition of the base address of the physical page in the associated page table entry. If the base address at the exit of the page fault exception is the same as that at the entry, SW-core updates the attributes of the page table entry. On the other hand, if the base address at the exit is different from the base address at the entry, SW-core allocates a new real page. The initialized data on the new real page are copied from the original real page. The attributes of the page table entry are also updated as they become necessary. In page-in handling, SW-core restores the real page from the information on physical

memory regions. The restored real page is determined according to the page table entry value at the entry of the page fault exception.

The para-virtualization version of Xen enforces read-only control on the bottom-level page tables. These page tables are validated and updated through page fault handling by the VMM or function calls to pass control from a VM to the VMM (i.e., *hypercalls* on Xen). Table 3.2 indicates how SW-core handles each page table entry update. In an attribute update, it updates the real page table entry associated with a dummy page table entry. In page-out, it updates the information on the physical memory regions in order to manage the real physical page. In both page-in and page-out, a target OS kernel manipulates dummy physical pages, while SW-core manipulates their corresponding real physical pages. Therefore, Shadowall can prevent a compromised target OS kernel from maliciously manipulating real physical pages during page-in or page-out. In page release, the reference count for the target physical page is decremented by 1. If the reference count becomes 0, SW-core releases information on the target physical regions corresponding to the target physical page.

Interception of Execution Mode Switching

To switch the user address space of a target program between the user and kernel modes, SW-core requires interception of the entry and exit of exception and interrupt handling, and of the entry and exit of system call procedures. For exception and interrupt handling, in software virtualization, the VMM can intercept their entries to control exceptions and interrupts occurring on the hardware. On Xen-AMD64, the VMM can also intercept their exits to control them through hypercalls. For a system call procedure on Xen-AMD64, the VMM can also intercept its entry to handle the SYSCALL instruction invoking the system call. On the other hand, as described in Section 2.3.2, a binary patching mechanism is introduced to intercept the exit of a system call procedure that is in place. When `sw_vconf` is executed, SW-core overwrites the first byte when the system call procedure exits with the HLT instruction.

3.4.2 Target File Protection

Target Dummy Files

A real target file is managed inside the control VM. When a real target file is created or updated, the user creates the associated dummy target file by using the `mkdummy` command. An ELF executable and or a script such as a shell script, Perl, or Python script is assumed as the target executable. For an ELF executable, a Linux OS kernel requires an ELF header, an ELF program header, and a `.interp` ELF section in order to load the executable in memory. Therefore, the `mkdummy` command clears the contents of the

dummy executable file except for the three pieces of ELF-related data. For scripts, the script file has to begin with a line of the `#!` form so that a Linux OS kernel can load the corresponding script executable in memory. Therefore, the `mkdummy` clears the contents of the dummy script file except for the line of the `#!` form. In addition, the user needs to deploy the real script executable in the control VM. The content of a target file shared with an uncontrolled program, such as a libc library file, is not cleared, while the content of all other target files is cleared.

File Manipulation Emulation

To control an operation on a target file, SW-core interposes at the entry and exit of system calls related to a file operation. When a control program has manipulated a target file, SW-core intercepts the system call. Then, it and the control program emulate the intercepted system call. The control program manipulates the real file content corresponding to invoked system calls. Meanwhile, SW-core communicates with the control program and transfers the real content between the real file and the memory regions including its content. Communication between the SW-core and the control program uses mechanisms provided by Xen: the event channel facility, and shared memory between a VMM and a VM. Although a system call is also executed inside the target VM, a target OS kernel does not manipulate a real file but a dummy file.

3.4.3 System Call Handling

System Call Arguments

System call arguments include pointers to user space. In Shadowall, a target program running in user mode manipulates real memory regions, whereas a target program running in kernel mode manipulates dummy memory regions. To maintain the consistency of a system call procedure between different execution modes, a target program running in different execution modes must manipulate the same memory data only during the system call procedure. To achieve this, SW-core makes the target program share memory regions manipulated in the system call procedure between a real page and a dummy page only during the procedure.

To share limited memory regions, SW-core uses information on the user space pointer of a system call argument. This information includes the number of system calls, and the numbers and sizes of pointer arguments. Through `sw_vconf`, this information is registered as part of the target OS information. The proposed system provides a support tool to automatically generate this information at the time a target OS kernel is generated, because the information depends on the OS kernel.

When a user space pointer is used as an input argument, SW-core copies the target

data from a real page to a dummy page at the entry of a system call. In contrast, when a user space pointer is used as an output argument, it copies the target data from a dummy page to a real page at the exit of a system call. The target data on the dummy page are cleared at the exit.

If the user space regions to which system call arguments point are not assigned when data sharing manipulation starts, SW-core restarts execution of the target VM to make the target OS kernel handle the page fault for the user space regions in advance. After page fault handling by the target OS kernel, SW-core intercepts the system call again and starts data sharing manipulation.

Program Execution Start and End

When a target program starts running through the `execve` system call, SW-core initializes the information on the memory-mapped regions for the target program. The initialization uses the memory-mapped information of a process such as the `start_code` and `start_data` members of an `mm_struct` kernel object. In cooperation with a control program, SW-core loads the content of the target program executable into memory. For an ELF executable, the content is loaded in conformity with ELF-related headers. For scripts, after reading the script file, SW-core loads the corresponding script executable. The memory-mapped information also includes the executable content by the page. If SW-core has already managed the content of the real page, it copies the real content to the initialized page. In addition, the loaded memory regions are multiplexed in conformity with the security policy.

When a process that has already been been controlled invokes `execve`, SW-core releases the process' associated the memory-mapped information. According to the security policy, SW-core determines whether the executed program must be controlled. If necessary, it initializes the memory-mapped information and multiplexes the loaded memory regions in the same way as described above.

When a target program invokes the `exit_group` and `exit` system calls to terminate execution, SW-core releases information on the memory-mapped regions and physical memory regions.

Heap and Memory-mapped Regions

When a target program invokes the `mmap` and `brk` system calls, SW-core updates the information on memory-mapped regions. It handles page faults and updates page table entries on the basis of this information. When the `mmap` and `mprotect` system calls are invoked, SW-core also updates the read and write permission in the information on memory-mapped regions. For `mmap`, SW-core also adds information on whether the file

referenced by the file descriptor argument is a target file. If it is a target file, in combination with a target program, SW-core fills in the content of the memory-mapped region with the real file content. In addition, if memory-mapped regions associated with a target file have the `PROT_WRITE` and `MAP_SHARED` flags, the content of a memory-mapped region is written to a real file when a target program invokes the `munmap` or `close` system call.

Signal Handling

On Linux, a signal handler is executed while switching between the user and kernel modes. An OS kernel saves the current execution context including the argument of a signal handler, on a user mode stack when receiving a signal. As with system call arguments, when a user mode stack region is multiplexed, Shadowall needs to intercept the signal handler invocation and transfer shared data between the user and kernel modes. To intercept a signal handler invocation, which occurs when a target program invokes the system calls to set up a signal handler, such as `sigaction` and `signal`, SW-core overwrites the first byte of the signal handler in the same way as for interception of a system call's exit. When intercepting the signal handler invocation, SW-core copies data related to the signal handler to the real user mode stack. It manages the signal handler information for each process.

File Manipulation

To identify a target file by a file descriptor, Shadowall manages information on the file descriptor associated with a target file. When a target program invokes the `open` system call, Shadowall adds the file descriptor information. When a target program invokes the `close` system call, Shadowall deletes the file descriptor information. In cooperation with a control program, SW-core emulates system calls for manipulating file content, such as `read`, `write`, and `lseek`.

Process Creation

When a target program executes a system call for creating a new process, such as `fork`, SW-core duplicates the information on the memory-mapped regions of the calling process at the exit of the system call. In addition, the information on the file descriptor and signal handler are also duplicated. When `fork` is intercepted at the system call exit, the entries in the page tables for the new process are mapped to dummy pages. Therefore, SW-core updates the page tables for the new process so as to make a target program running in user mode manipulate real pages.

When a target program executes the `clone` system call to create a new light weight process (i.e., a thread), the new process shares the necessary data with the calling process. If the `CLONE_VM` flag is set, the memory-mapping information is shared with the calling process. Similarly, if the `CLONE_FILES` flag is set, the file descriptor information is shared. If the `CLONE_SIGHAND` flag is set, the signal handler information is shared.

Interprocess Communication

Pipes and message queues are used for interprocess communication. The transferred data are kept in kernel space. Shadowall protects the data by using a cryptographic technique. In the pipe case, when a target program invokes the `pipe` system call, SW-core adds pipe file descriptors to the file descriptor information. Then, when the target program transfers data by using the pipe file descriptors, the data are encrypted before transfer and decrypted afterward. When the `msgsnd` and `msgrcv` system calls are invoked in message queues, the data are again respectively encrypted and decrypted. The current prototype system encrypts and decrypts variable length data transferred in interprocess communication through XOR operations by the byte in cipher-block chaining mode.

3.5 Evaluation

We evaluated Shadowall by using two metrics: its effectiveness against malicious operations on memory and virtual disks, and its performance overhead. In the experiments, Shadowall ran a hardware platform with two dual-core AMD Opteron 2.8 GHz processors, 8 GB of RAM, and a 1 Gbps NIC. The control VM and target VM were both configured with 4 CPUs and 1 GB of memory running a para-virtualized Linux OS kernel.

3.5.1 Effectiveness Against Malicious Operations

thttpd Web Server

In the experimental application to the *thttpd* Web server, its chroot jail function was protected. First, we ran a thttpd server instance configured with the chroot jail function enabled. The chroot jail was configured to limit the file system that thttpd could manipulate to files under the `/var/www` directory. The thttpd server in this configuration could thus not execute a CGI program using a `/usr/bin/perl` executable.

To simulate an attack against thtttpd, its configuration data on memory and a virtual disk was tampered with so that it could execute the CGI program. To tamper with the configuration data, we restarted thttpd and forced it to clear the chroot jail variable, `do_chroot`, to disable the chroot jail function. To modify the `do_root` variable, the GDB debugger was used to make thttpd stop temporarily just before its chroot jail function

was checked and the do_chroot variable was cleared. As a result of this tampering, thttpd could then access the /usr/bin/perl executable and execute the CGI program, even though it started running with the chroot jail function enabled according to its configuration file. Furthermore, to prevent the attack footprint from being detected, records indicating that thttpd was restarted and run it with its chroot jail function disabled were deleted from the log file /var/log/syslog.

Next, to demonstrate that Shadowall could prevent the attack described above, thttpd was controlled with Shadowall. The security policy specified that all the memory regions had to be multiplexed and that target files had to include the log file /var/log/syslog. The policy also specified the memory and virtual disk data related to the system logging utilities, *syslogd*, because its logs are in /var/log/syslog. Although we attempted to disable the chroot jail function in the same way as previously described, it could not be disabled for thttpd running under Shadowall control. Since the code regions on the dummy page had been cleared, a breakpoint could not be set for when the do_chroot variable was checked. Furthermore, since the real log file was controlled outside the target VM, its content could not be manipulated.

ClamAV Anti-virus Tool Suite

The virus scanning procedure was protected in an experimental application to the *ClamAV* anti-virus tool suite [2]. An infected file, mw.exe, was scanned using the command-line anti-virus scanner, *clamscan*. First, a virus database file, mw.hdb, including the virus signature for mw.exe, was created. Second, clamscan was executed to scan the mw.exe file according to the mw.hdb database. The scanner wrote a program message that mw.exe was infected to the standard output. To simulate an attack against clamscan, it was forced, in two ways, to print a false message that mw.exe was not infected. The first way was by tampering with the memory data related to clamscan. The return value of the cl_scandesc function, representing the scanning results with the GDB debugger, was modified. The second way was by tampering with the virtual disk data related to clamscan. The virus signature for the mw.exe file in the mw.hdb database was modified.

Next, we demonstrated that Shadowall could prevent this attack by controlling clamscan. The security policy specified that all the memory regions had to be multiplexed and that target files had to include the mw.hdb database file. Although we attempted to force clamscan, running under Shadowall control, to print a false message, it instead printed a true message that mw.exe was infected. As in the case of thttpd above, the return value of the cl_scandesc function on the real page could not be tampered with, not could the content of the real mw.hdb database file deployed outside the target VM.

3.5.2 Impact on Performance

Microbenchmarks

To ascertain which components contributed to the cost of the overhead introduced by Shadowall, we used the microbenchmark programs described below. A system call sequence was repeated 10,000 times for each microbenchmark.

getpid: This program repeatedly invoked the `getpid` system call.

mmap (alloc, write): mmap(alloc) allocated a 4 KB anonymous memory-mapped region and then immediately freed the allocated memory-mapped region with the `mmap` and `munmap` system calls. For mmap(alloc), Shadowall updated the information on memory-mapped regions. For mmap(write), which wrote data to the allocated region. Shadowall also updated the information on physical pages, including the handling of memory multiplexing due to page fault exceptions.

read (w/ emu, w/o emu): After read(w/ emu, w/o emu) opened a file under the home directory, it read 1 KB of content from a target file deployed both inside and outside the target VMs. For read(w/ emu), SW-core and a control program copied the content of the real file to the real memory page. For read(w/o emu), SW-core copied the content of the dummy memory page to the real memory page.

write (w/ emu, w/o emu): write(w/ emu, w/o emu) wrote 1 KB of data to a target file deployed both inside and outside the target VMs. For write(w/ emu), SW-core and a control program copied the content of the real memory page to the real file. For write(w/o emu), SW-core copied the content of the real memory page to the dummy memory page.

fork: A parent process created a new child process with the `fork` system call. The parent process waited for the child process to terminate with the `wait4` system call. The child process immediately terminated with the `exit_group` system call. Shadowall updated the information for controlling the new child, as described in Section 3.4.3.

pipe: This program created pipes for interprocess communication with the `pipe` system call. After a new child process was created with `fork`, this program transferred 1 KB of data between the parent and child processes. In addition to updating the information for controlling the new child, Shadowall encrypted and decrypted the data transferred between the parent and child processes.

The execution time was also measured for each micorobenchmark running on a Xen VMM (Xen) and a Linux OS kernel (Linux), and the measured times waere compared

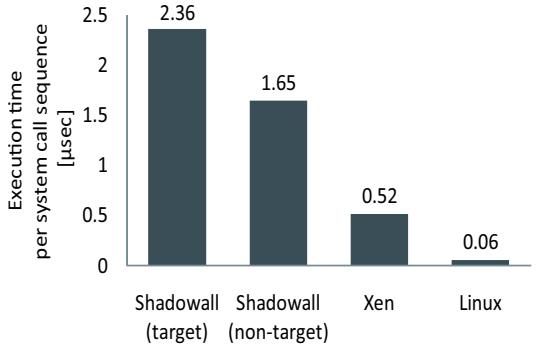

Figure 3.6: `getpid` microbenchmark results on Shadowall, Xen, and Linux

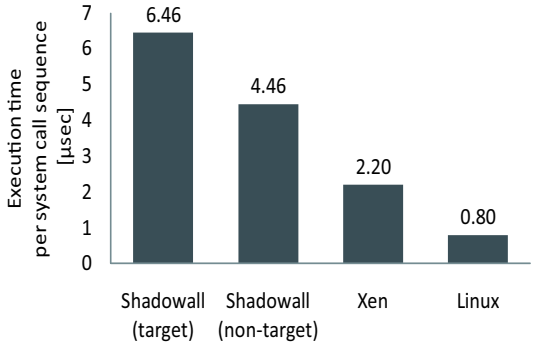

Figure 3.7: `mmap (alloc)` microbenchmark results on Shadowall, Xen, and Linux

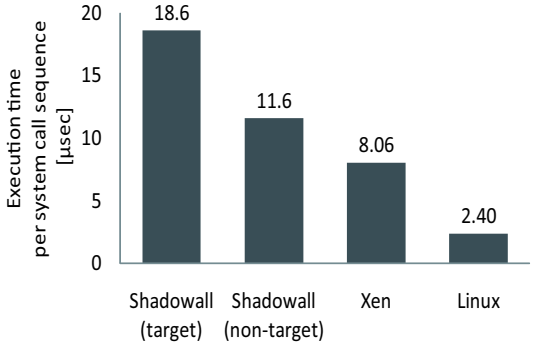

Figure 3.8: `mmap (write)` microbenchmark results on Shadowall, Xen, and Linux

with those for Shadowall. The Linux platform was configured with 1 GB of memory. For Shadowall, the execution time was also measured for each microbenchmark running without memory or virtual disk protection. For all the processes running on the target VM, Shadowall interposed the handling of page fault exceptions, page table updates, and system call procedures.

The experimental results for the microbenchmarks are shown in Figures 3.6, 3.7, 3.8, 3.9, 3.10, 3.11, and 3.12. The values in the figures are the execution times for each system call sequence. The "target" and "non-target" rows for Shadowall in the figure correspond to the respective results for each microbenchmark running with and without memory and virtual disk protection. The "read(w/ emu)" and "write(w/ emu)" columns only indicate the results for the "target" of Shadowall because the real file was manipulated in that

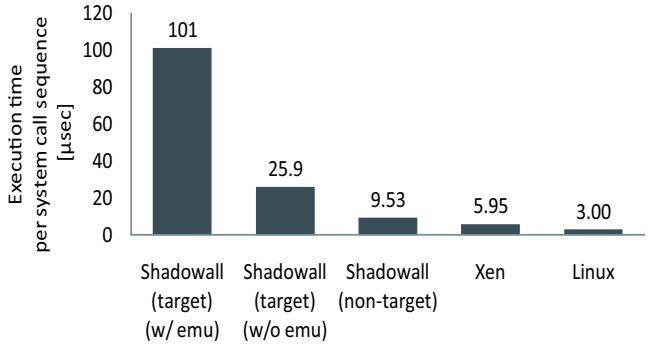

Figure 3.9: `read` microbenchmark results on Shadowall, Xen, and Linux

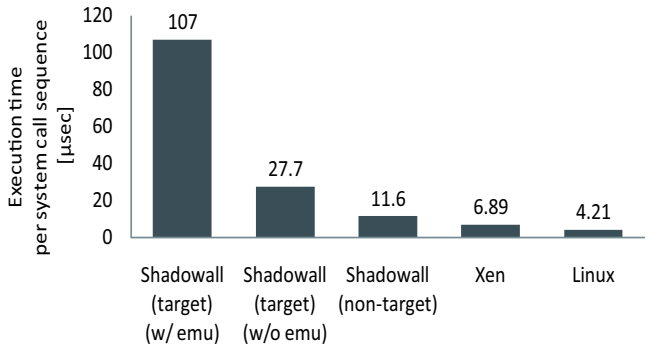

Figure 3.10: `write` microbenchmark results on Shadowall, Xen, and Linux

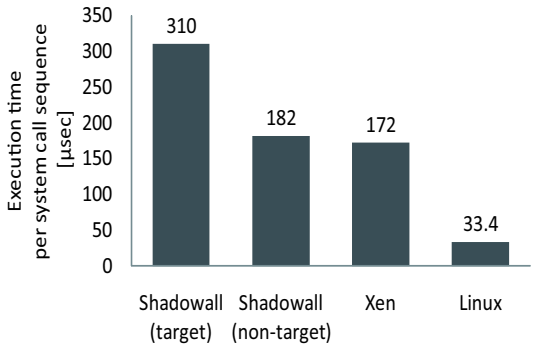

Figure 3.11: `fork` microbenchmark results on Shadowall, Xen, and Linux

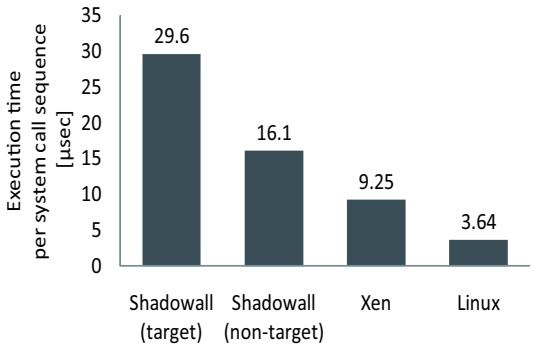

Figure 3.12: `pipe` microbenchmark results on Shadowall, Xen, and Linux

case.

For "target" programs, the overhead incurred by emulation of target file operations ("read(w/ emu)" and "write(w/ emu)") was higher than that incurred by memory manipulation ("mmap") and encryption/decryption operations ("pipe"). The factors inducing this higher overhead were considered to be the context switching between VMs and the data transfer between VMs. One factor in the higher overhead incurred by "getpid" was considered to be the interposition of system calls, which had more execution time impact on "getpid" than on the other microbenchmarks because system call invocations account for a large portion of its execution.

Application Benchmarks

Shadowall was expected to control target applications with less performance impact than on the microbenchmark programs. To confirm this, we used the *thttpd* and *Apache* Web servers and the *ClamAV* anti-virus tool suite as target programs. The security policy for each application specified that all memory regions had to be multiplexed and the target files had to include the applications' configuration files. As with the microbenchmarks, the performance of Shadowall was compared with that of a Xen VMM (Xen) and a Linux OS kernel (Linux), where the latter was configured with 1 GB of memory.

The throughput for Web services was measured using the *ApacheBench* benchmark tool for the thttpd and Apache servers. ApacheBench was launched on a separate physical machine that had an Intel Pentium 4 3.0 GHz processor with hyper-threading enabled, 1 GB of RAM, and a 1 Gbps NIC. This machine was connected to the physical machine running Shadowall via a gigabit network, and the two physical machines were deployed within the same LAN. ApacheBench issued requests to fetch two kinds of static content (1 KB and 100 KB files) and dynamic content (CGI). The CGI program, using a *Perl* script, showed the computing environment of the physical machine on which Apache ran. The security policy for CGI specified that the target files had to include the script file and the Perl executable. The number of requests was configured as 128. Whereas one process handled all the requests in thttpd, multiple processes handled the requests in Apache.

The *clamscan* and *clamd* scanners for ClamAV were also used. The clamd program is an anti-virus daemon for handling scanning requests sent from the clamd client, *clamdscan*. While clamscan read the virus database for every request, clamd only read the virus database when it starts running or receives an update request from a user. The anti-virus tools were used to scan 15 files, including five infected files. The clamscan, clamd, and clamdscan programs were specified as target programs as was the virus database file. First, the virus database file was created from the five infected files. Then, the virus database was deployed inside the control VM and used to scan for viruses. File operations on the database file inside the control VM were emulated.

Figures 3.13, 3.14, and 3.15 indicate the experimental results for the Web servers and

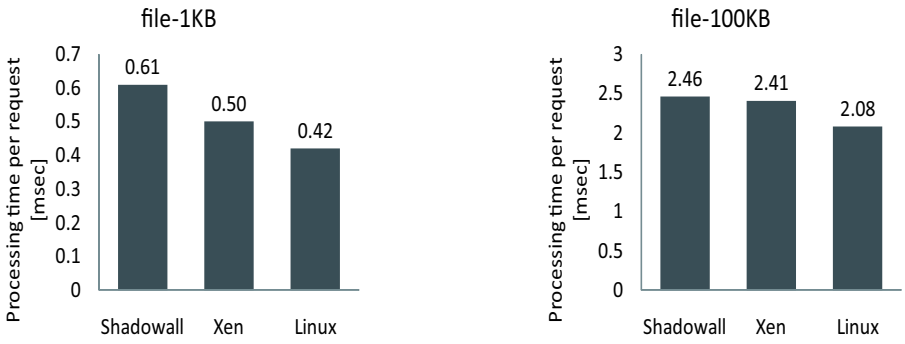

Figure 3.13: *thttpd* Web server throughput on Shadowall, Xen, and Linux

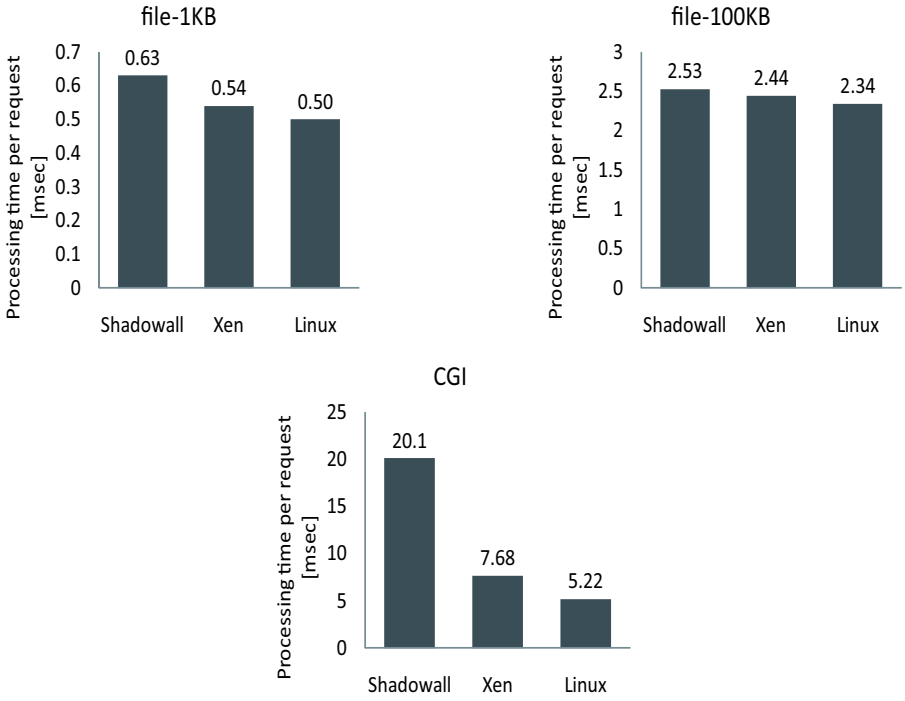

Figure 3.14: *Apache* Web server throughput on Shadowall, Xen, and Linux

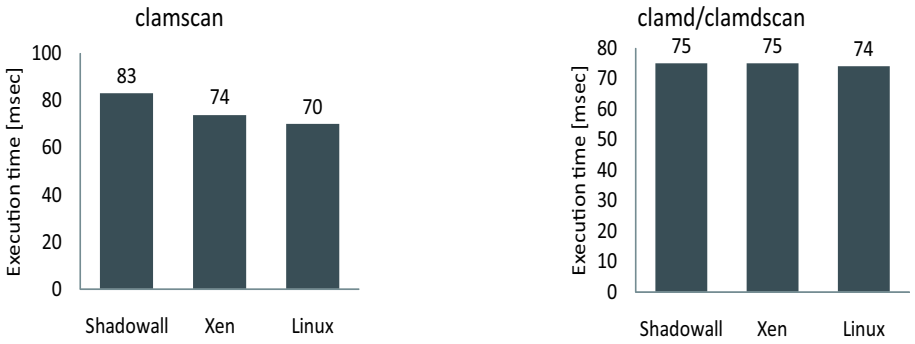

Figure 3.15: Virus scanning times on Shadowall, Xen, and Linux

anti-virus tool suite. As expected, Shadowall could control the applications with lower overhead than that incurred by controlling the microbenchmarks. The results for the application benchmarks also indicate that Shadowall had a reduced impact on system performance, relative to that of the original Xen. For the static content processed by the Web servers and clamd, the overhead introduced by Shadowall was smaller than that for the CGI and clamscan cases. The main factor in the larger overhead incurred by CGI services was that, after the script file was read, the Perl executable was loaded into memory for every request. Meanwhile, the main factor in the larger overhead incurred by clamscan was that the target files, such as the configuration files and the virus database file, were read at the time of initialization.

Although the proposed memory multiplexing scheme doubled the physical memory use of the applications, the extent of the multiplexed memory region should be reducible by applying demand paging and shared memory between threads. To explore this possibility, the impact on the maximum amount of multiplexed memory and the number of current processes after the time of initialization was also measured for the server programs: thttpd, Apache, and clamd. The number of multiplexed 4 KB pages was about 240 for thttpd and clamd, and about 1,600 for Apache. Whereas the number of current processes was only one for thttpd and clamd, Apache consisted of 56 threads in four processes. We believe that users can accept these numbers of multiplexed pages.

In summary, although the performance of the target programs was degraded more than when they were run on a Linux OS kernel in this evaluation, Shadowall could protect the memory and virtual disk data related to these programs.

3.6 Related Work

3.6.1 Data Protection at VMM and Hardware Layers

There have been VMM-based systems [26, 91, 119] that use memory multiplexing schemes to protect the memory data in user space. These previous systems require dedicated programs or commands for controlling a target program deployed inside a target VM. In contrast, Shadowall does not require adding any dedicated programs because it executes control from outside the target VM. Overshadow [26] and SP^3 [119] use cryptographic techniques to provide logically different views of the target program page to different execution modes, whereas Shadowall provides physically different views of the target program page to different execution modes. Although this increases the physical memory use, the proposed memory multiplexing scheme does not impose the extra overhead incurred by cryptographic schemes, which add the costs of encryption and decryption operations. Furthermore, for a target VM with multiple virtual CPUs, a decrypted page must be manipulated exclusively in order to prevent its content from being leaked and tampered with. As a result, cryptographic schemes also introduce overhead due to their exclusive operations for decrypted pages. Unlike the previous systems, Shadowall can also specify partially multiplexed memory regions. Rosenblum et al. [91] provided different views that physically depend on instruction and data fetches, whereas the memory multiplexing scheme here depends on the execution mode. As with its scheme for memory data protection, Overshadow encrypts its content for virtual disk protection, whereas Shadowall manages and controls its content outside target VMs. Neither SP^3 or Rosenblum et al. [91] provided a scheme for virtual disk protection.

XOM [66] is a trusted processor architecture that protects the code and data regions of a target program by using dedicated instructions and cryptographic operations. To protect a target program running on top of XOM, a user must use a XOMOS [65], i.e., an OS dedicated to running on top of XOM.

Several existing systems [41, 55, 100, 103, 115] make a target program run outside a target VM by using a VM isolation scheme. Unlike these existing systems, Shadowall runs a target program inside a target VM but its memory and virtual disk data are controlled from outside the target VM. Proxos [103] executes trusted system calls in a trusted VM while executing untrusted system calls in an untrusted VM. It requires modifying the source code of not only an OS kernel but also a target program. In contrast, Shadowall supports a target program without requiring any changes to its source code. Proxos' security policy specifies which system calls are trusted, whereas Shadowall's security policy specifies which parts of memory regions and which files are protected. Nizza [100] is a microkernel-based trusted architecture that isolates security-critical software components such as a key authentication mechanism from an untrusted OS. Like Proxos, Nizza requires modifying the source code of a target program and of programs that cooper-

ate with the target program. VPFS [115] is a private file system based on a microkernel, which protects sensitive files stored in an untrusted OS. A sensitive application running in a trusted VM manipulates the sensitive files through a VFS server. Unlike Shadowall, VPFS requires modifying the source code of the sensitive application to use the VPFS client API. To detect stealthy malware outside an untrusted VM, VMwatcher [55] periodically performs integrity checks on memory and virtual disk data , whereas Shadowall controls data manipulation when operations occur.

To protect sensitive files such as shared libraries inside an untrusted VM, SVFS [123] controls access to sensitive files from outside an untrusted VM. It provides access control mechanisms at the file system layer, whereas Shadowall protects the content of a target file by controlling system calls. Since SVFS transfers sensitive file data by using an NFS client, a compromised OS kernel in an untrusted VM can manipulate such content maliciously. Shadowall, on the other hand, can prevent malicious manipulation by a compromised OS kernel because SW-core and the control program running in a control VM transfer the content of a target file between memory and a virtual disk.

3.6.2 Data Protection at OS and Application Layers

Various existing security systems run at the OS and application layers to enhance the security of applications at process granularity. PeaPod [84], using OS-level virtualization, controls the behavior of an untrusted program by providing isolation between the protection domains allocated by the process and access control between protection domains. Janus [45] and Systrace [86] are sandboxing systems that control system calls invoked by untrusted programs according to their security policies. Solitude [54] is an isolation and recovery system based on a file system at the application layer. According to a security policy, Solitude provides an execution environment bound to a name space based on a file system for untrusted programs. None of these various security systems can prevent attacks from untrusted programs that are not under the systems' control. Tripwire [60] periodically checks file-system integrity, but unlike Tripwire, when operations for memory and virtual disk data occur, Shadowall controls these operations. SELinux [52] and LIDS [4] are security-enhanced OSs that enforce mandatory access control to protect their execution environments. Since security systems running at the OS and application layers are managed and controlled by an OS kernel, the compromised OS kernel and privileged programs can evade and subvert data protection mechanisms provided by the security systems.

3.7 Summary

We have proposed Shadowall, a security system that protects memory and virtual disk data relevant to target programs specified by the user, from outside the target VMs. Even if an OS kernel or a privileged program is compromised in a target VM, Shadowall prevents the compromised program from leaking and tampering with the memory and virtual disk data related to the target programs. Shadowall hides this data from other programs, including an untrusted OS kernel running inside the same target VM. To protect memory data in user space, the VMM multiplexes physical pages to provide different views to different execution modes. To protect virtual disk data, a separate trusted VM manages files related to an untrusted program. When a target program invokes a system call involved in file operations, the VMM emulates the invoked system call in cooperation with the trusted VM. For each target program, the user applies a security policy to specify which parts of memory regions and which files are protected. Experimental evaluation demonstrated that Shadowall successfully disabled synthetic attacks that attempt to modify the memory and virtual disk data of existing application programs.

The following are current avenues of exploration for future work. First, the implementation of Shadowall's memory protection mechanism, through which target programs execute new programs, should be improved, since the experimental results for CGI Web throughput showed that the overhead was significant larger in this case than in other cases. In particular, program loading should be optimized. In the current naive implementation, whenever a program is launched by a target program, Shadowall reads the binary data from a separate VM, the control VM. A caching mechanism should thus be introduced for this binary data. Another topic work for future work is to maintain compatibility with legitimate programs. The current prototype system does not permit even legitimate programs (e.g., security systems) to manipulate "real" data involved in untrusted programs. To overcome this limitation, the security policy could be extended so that only particular programs specified by the user can manipulate "real" data related to untrusted programs but protected by Shadowall. Finally, the protection of register values when switching the execution mode is under investigation in order to enhance the proposed data protection scheme.

Chapter 4

Application-aware Control of Kernel Behavior

4.1 Motivation

Kernel-level malware using rootkits, referred to as *malicious kernel-level rootkits*, poses serious threats in two ways. First, such malware attacks impact system-wide behavior and sensitive or critical data. For example, malicious kernel-level rootkits attempt to tamper with OS kernel code and data to install backdoors. Second, many malware programs can subvert or circumvent anti-malware software in order to hide their activities and presence. Various systems have been proposed for detecting [59, 73, 74] and preventing [43, 62, 68, 88, 99, 114] these attacks.

However, these existing systems have three drawbacks: strict limitation on kernel extensions, evasion of control and protection mechanisms, and performance degradation. Regarding strict limitation on kernel extensions, Patagonix [68] and NICKLE [88] permit only authenticated kernel code to be executed in untrusted VMs. Loadable kernel modules are often used to extend the functions of a Linux OS kernel without forcing it to restart. To get such extensions recognized as legitimate by Patagonix and NICKLE, however, users must register their loadable kernel modules in advance. Kruegel et al. [62] verified the validity of loadable kernel modules at load time. Certain existing security systems, however, such as sandboxing and anti-virus systems, have utilized loadable kernel modules that monitor and control processes and files [34, 42] in manners similar to those used by malicious kernel-level rootkits. Therefore, it can be difficult to distinguish between malicious kernel modules used by kernel-level rootkits and legitimate ones used by security systems. Consequently, Kruegel et al.'s protection scheme could mistakenly regard a legitimate loadable kernel module as illegitimate.

As for evasion of control and protection mechanisms, prior systems [55, 68, 73] per-

form runtime checks periodically or on demand. The difficulty in subverting or evading systems based on periodic runtime or on-demand checks [55, 68, 73] depends on the intervals of these checks. A longer interval induces lower overhead but also increases the chances of timing attacks.

Lastly, there are three problems that degrade performance. First, unlike self-contained malicious programs at the user level, malicious kernel-level rootkits are not used alone but in cooperation with other malicious programs. Therefore, most systems [88, 111, 121] use a CPU emulator to control and analyze behavior and data flow through system-wide tainting and slicing techniques and control at the granularity of a processor instruction. Second, another group of prior systems [43, 99] based on hardware protection mechanisms uses non-executable (NX) bits or write-protect bits, on the Intel and AMD architectures, for each page. This coarse-grained control at page granularity frequently causes unnecessary page faults when controlled and uncontrolled data are mixed in the same page. Finally, runtime checking at the kernel level generally imposes much higher overhead than that at the user level, since the kernel space is shared among all processes. Consequently, runtime checking does not occur in the context of processes to be controlled but in the context of all currently running processes.

In this chapter, we present ShadowXeck, a system for controlling only the behavior of target OS kernels associated with improving existing systems in the two ways described above. The proposed system does not confine the functions of kernel extensions inside target VMs. These functions instead apply to all currently running processes except for those to be controlled inside the target VMs. Furthermore, this system reduces performance degradation because it performs control only at the kernel context involving the target programs, in accordance with a security policy. In other words, ShadowXeck does not interpose by controlling kernel behavior involved in programs that are not to be controlled.

One usage scenario is to control programs that are vulnerable to attack by kernel-level rootkits (e.g., system utility programs such as *ps* and *ls*). Another usage scenario is to complement security systems that provide security for application programs at the user level. For example, although ShadowVox, presented in Chapter 2, can control the entry and exit of system calls, it cannot control system call procedures in kernel mode. In cooperation with ShadowXeck, ShadowVox can reduce the trusted computing base (TCB), because ShadowXeck ensures the execution of legitimate kernel code and the use of a valid system call table. In contrast, ShadowXeck is inadequate for controlling OS kernel behavior for all processes. Examples include controlling a keyboard interrupt handler and a process scheduler. Control of the behavior of all processes does not take advantage of ShadowXeck's two key features, i.e., control with no restriction on kernel extensions, and lower performance overhead.

The rest of this chapter is organized as follows. In Section 4.2, we describe the threat model. Section 4.3 presents the design of the proposed system, followed by a detailed

description of a prototype implementation in Section 4.4. Sections 4.5 and 4.6 discuss an evaluation of the system and related work, respectively. Finally, we summarize the chapter in Section 4.7.

4.2 Threat Model

We assume that attackers have the ability to modify the control flow of target OS kernels by tampering with their code and data through kernel-level malware. A large amount of kernel-level malware attempts to modify an OS kernel's control flow. Among 26 examples of real-world, kernel-level malware on Linux, including *PhalanX* and 25 others shown by Petroni et al. [74], 25 (96%) have functionality for modifying the control flow of OS kernels. In the threat model for ShadowXeck, the main goals of attackers are to hide their presence and activities from system utility programs, such as `ps` and `ls` commands, to induce leakage of system and personal confidential data manipulated by privileged programs, and to disable the protection mechanisms of security systems, such as sandboxing and intrusion detection systems, on the victimized VM.

Here, kernel-level malware is assumed to be embedded using loadable kernel modules and special devices such as `/dev/kmem` and `/dev/mem`. Attackers can accomplish this in the following two ways.

- An attacker tampers with read-only data in arbitrary memory locations by granting malicious write permission for this data. For example, such data includes code regions and the system call table on Linux. The attacker accomplishes this by setting the R/W bits of the page table entry for read-only regions or by creating new memory mappings to these regions.

- An attacker tampers with values in registers and writable data on memory in order to modify the control flow determined by these values at runtime. The values are used by indirect jumps, such as indirect CALL and JMP instructions, at the processor-instruction level. From the perspective of a higher level, e.g., programming-language level, this kind of attack includes tampering with function pointers related to a virtual file system (VFS) or a `/proc` file system. This kind of attack belongs to the category of kernel object hooking (KOH) [39].

ShadowXeck is a security system for ensuring that the behavior of target OS kernels in the target program context is legitimate, even if kernel-level malware resides in the target VM. In other words, ShadowXeck is not a security system for detecting the existence of kernel-level malware, getting rid of it, and ensuring the safety of OS kernel behavior. Instead, it allows target VM users to extend OS kernels under the condition that they do not modify write-protected regions in the kernel space. Hence, malicious kernel-level

rootkits could potentially be inserted in target VMs. ShadowXeck cannot prevent a malicious kernel-level rootkit from subverting target application programs using in-OS kernel functions.

The VMM and a special VM cooperating with the VMM are the trusted computing base (TCB). Moreover, ShadowXeck depends on process management data for the target OS kernels, because it identifies the launch, termination, and process creation of a target application program by using the VM introspection technique described in Section 2.3.1. For example, an attacker can modify process management data involved with target application programs in order to bypass identification by ShadowXeck. However, execution of these target application program without ShadowXeck's control can be thwarted through combination with Shadowall, because Shadowall does not load target application program data into memory.

Here, we do not consider detection and prevention of tampering with the data structures managed by OS kernels to conceal malicious presence and activities [59, 114]. This kind of attack belongs to the category of direct kernel object manipulation (DKOM) rootkits [50]. ShadowXeck does not focus on corruption of lists of current processes and loadable kernels. This kind of attack also includes tampering with lists of running process, such as the run queue in Linux and callback functions for handling an interrupt. Furthermore, ShadowXeck does not focus on tampering with return addresses on kernel stacks, such as in return-to-libc attacks. Rather, ShadowXeck can be used in combination with existing techniques [32, 64] to mitigate this kind of threat.

In addition, improving the security of target programs at the user level is outside the scope of ShadowXeck. However, the behavior of target programs can be controlled using ShadowVox presented in Chapter 2. Furthermore, user space memory and virtual disk data can be protected using Shadowall presented in Chapter 3.

4.3 Design

4.3.1 Overview

Figure 4.1 shows the structure of ShadowXeck. A user operates ShadowXeck through control command utilities, while the security policies for each target program are managed in the control VM. ShadowXeck has three types of components:

- *In-VMM component* (*SX-core*): This component prevents adversaries from corrupting read-only regions. In addition, it controls the behavior of target OS kernels for each target program by using target OS information.

- *Control command utilities*: These includes commands for providing SX-core with target OS information and starting control of the behavior of the target OS kernels.

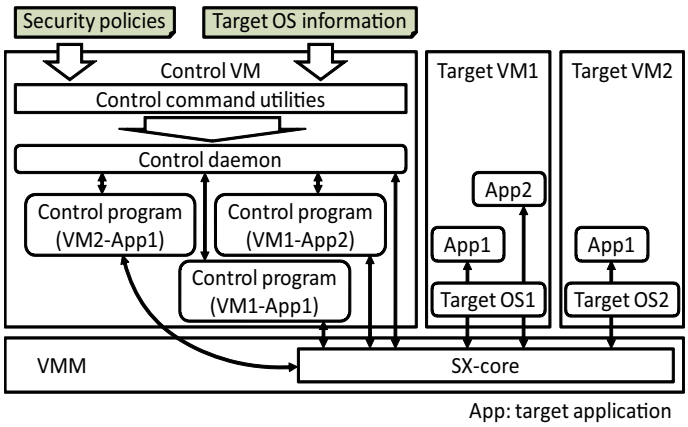

Figure 4.1: ShadowXeck: system for controlling the behavior of OS kernels at the contexts of target application programs

The utilities also include the commands for generating the target OS information on process management structures and system calls, as described in Section 2.3.1, and template security policies, as described in Section 4.3.4.

- *Control daemon*, *control program*: The control daemon handles requests from a control command. It also launches a control program for each target program when the target program starts. The control program provides security policies to SX-core and writes execution logs received from SX-core to a file.

With regard to protecting read-only regions, ShadowXeck permits target VM users to use the kernel extensions without departing from the designs of target OS kernels. For example, the system does not permit modification of data that were originally read-only, such as data in code regions.

With regard to controlling indirect jumps, ShadowXeck does not control them when kernel-level malware tampers with a value in a register or at a memory location where a destination address is stored. Rather, ShadowXeck conforms with the security policy in place when indirect jumps are executed.

Control of indirect jumps is based on the target process context. To determine whether the current process context is involved with a target program, ShadowXeck uses information on the design of a target OS kernel. This information includes the process management data structure and system calls. Hence, a Linux OS kernel was chosen as the target OS kernel because Linux can acquire this information. UNIX-like OSs are also considered applicable in ShadowXeck.

4.3.2 Controlling OS Kernel Behavior

To control the behavior of target OS kernels, we have developed two runtime protection schemes. First, ShadowXeck prevents illegitimate manipulation of each page table inside the target VM at the VMM layer to prevent tampering with read-only data, such as code regions and the system call table. Second, ShadowXeck controls the control flow of a target OS kernel determined at runtime. This corresponds to the control of indirect jump instructions at the assembly-language level, such as indirect CALL and JMP instructions whose control is transferred according to the values of a register or values stored on writable memory regions. In other words, this control approach corresponds at the program-language level to function pointers and `switch` statements in the C language. Whereas the scheme for protecting read-only data are applied for each target OS kernel, the scheme for controlling indirect jump instructions is applied for each target program.

Approaches for controlling indirect jump instructions at the VMM layer are primarily classified into two categories according to their memory access contexts: instruction context, data context. Approaches based on instruction context control when indirect jumps are executed, while approaches based on a data context control when destinations are written and read. The proposed system belongs to the category of approaches based on instruction context. ShadowXeck intercepts indirect jump instructions issued by a target OS kernel. In the instruction context, there is another possible approach that makes a target OS kernel render different views according to the memory access context, in the same way as the system developed by Rosenblum et al. [91] and NICKLE [88]. This approach multiplexes memory regions including indirect jump destinations, and it renders different physical pages according to whether memory manipulation is performed in an instruction or data context. However, this approach also requires synchronizing data between physical pages associated with the same virtual address, since this will also include data, except for indirect jump destinations, that do not require control. In fact, the Linux OS kernel, used in the evaluation described here, allocates function pointers for operating on `/proc` file system entries as indirect jump destinations and run queues as data that are frequently updated but are not targets in the same physical page. Furthermore, indirect jump destinations also exist in data that are dynamically allocated in the kernel space, such as function pointers in the `file` and `sock` kernel objects. Dynamically allocated data must be multiplexed when they are allocated and released, in addition to the above synchronous processes.

The approaches based on data context are divided into two types. One type uses a hardware memory protection mechanism [82, 99]. This type of approach clears the R/W bits in a page table entry associated with pages including indirect jump destinations in order to prohibit write operations to those pages. However, this approach performs control at page granularity as with the approach based on memory multiplexing. Therefore,

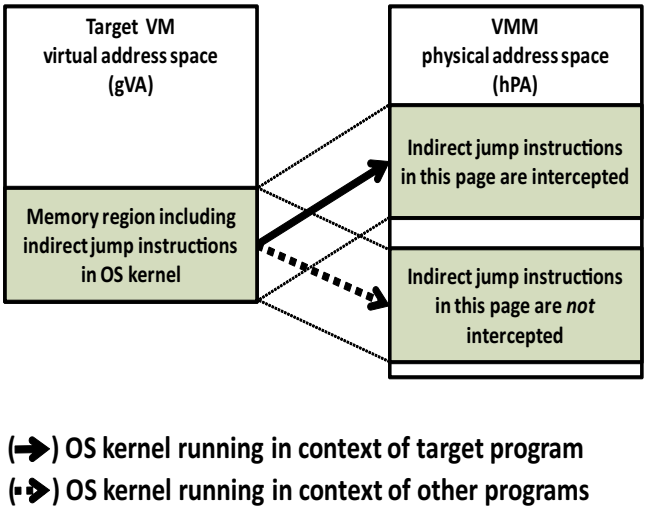

Figure 4.2: Multiplexing kernel address space including indirect jump instructions

redundant interceptions are also frequently caused by write operations to data that are co-located with indirect jump instruction destinations but not with targets. In the second type of approach, a target OS issues write and read instructions to indirect jump destinations [74, 112]. However, this approach causes redundant interceptions, i.e., it also intercepts instructions that are not indirect jump instructions. Furthermore, these approaches require the source code of a target OS kernel in order to collect indirect jump destinations through data flow analysis of the OS kernel.

4.3.3 Application-aware Control

Multiplexing Kernel Address Space

To reduce the performance degradation of processes that are not to be controlled, ShadowXeck renders different code regions, depending on the process context at the kernel level. To control indirect jump instructions issued only in the context associated with the target program, it multiplexes physical pages that contain indirect jump instructions in the kernel space, as shown in Figure 4.2. The VMM translates virtual addresses inside a target VM (gVA) into physical addresses in physical memory, since it needs to control Memory Management Unit (MMU) operations on the physical machine. This multiplexing at the VMM layer not only prevents kernel-level malware residing in a target VM from disabling this but also is transparent to a target OS kernel; that is, ShadowXeck provides different views to a target OS kernel without making the OS kernel aware of the address translation. ShadowXeck needs to intercept indirect jump instructions to control

them when they are issued by a target OS kernel. Therefore, dynamic binary instrumentation, described in Section 2.3.2, is applied to force generation of an exception when a target OS kernel issues an indirect jump instruction.

Indirect jump instructions are controlled in the context of all processes, rather than only that of the target program, in two ways: for flexibility for kernel extensions, and for system performance. First, one goal is to permit target VM administrators to use kernel extensions, such as loadable kernel modules, without any harm to the original design of a target OS kernel. Whereas ShadowXeck enables target VM administrators to utilize kernel extensions flexibly, unlike with previous approaches [68, 88], the behavior of a target OS kernel associated with a target program is controlled according to the security policy. Second, ShadowXeck reduces the performance degradation incurred by controlling indirect jump instructions, since its control is executed only in the context of a target program.

The kernel address space is partially multiplexed, in that only physical pages including indirect jump instructions in the kernel space are considered. A Linux OS kernel shares the kernel address space containing code and data regions among all current processes. The kernel address space is roughly classified into two regions: write-protected and writable regions. Multiplexing can be performed separately for each region because the regions are generally aligned with a page boundary [1]. Indirect branch instructions exist in the write-protected regions. ShadowXeck ensures consistency among the different physical pages that depend on the current process context, since data in the write-protected regions are not updated dynamically. The kernel address space also contains writable shared data, i.e., data that must be shared across the execution of all processes and are dynamically updated, such as run queues and process lists. If the whole kernel address space of a target program was multiplexed, it would require ensuring the consistency of the writable shared data, since they exist in different physical pages. Partial multiplexing enables ShadowXeck to intercept indirect jump instructions issued by a target OS kernel without an extra mechanism to maintain such consistency of writable shared data. The proposed multiplexing scheme has an additional advantage of reducing the number of multiplexed physical pages.

The Linux OS kernels for x86 and x86-64 architectures use large pages of 4 and 2 MB, respectively, to manage the kernel address space. The amount of multiplexed memory, however, can be reduce because the VMM manages the address translation to physical pages. To achieve this, ShadowXeck manages the address space of a target OS kernel with a smaller 4-KB boundary [2].

[1] Linux version 2.6.18 and older kernels are not aligned with page boundaries

[2] The original Xen 3.0.3 manages the kernel address space running on a VM with a 4-KB aligned page.

Leveraging Information on OS Kernels

To control indirect jump instructions according to the process context of a target OS kernel, ShadowXeck uses three types of information, on process management, system calls, and indirect jump instructions. This information is collectively referred to as *target OS information*. As described in Section 2.3.1, information on process management and system calls is used to identify when a target program starts and generates a new process, and to intercept the entry and exit of system calls at the VMM layer. Information on indirect jump instructions includes the instructions themselves and the memory locations where they are executed.

4.3.4 Security Policy

Control of Indirect Branch Instructions

ShadowXeck uses security policies to control indirect jump instructions issued by a target OS kernel. A security policy is described by the administrator of a target VM or of the control VM, which is a trusted, separate VM. The challenge in describing a security policy is to identify legitimate branch destinations for each instruction. It is not straightforward for users who do not know enough about the internal structure of an OS kernel to determine whether an issued instruction is legitimate. Furthermore, the virtual addresses of branch destinations depend on the binary image of an OS kernel.

To overcome this challenge, ShadowXeck provides two operation modes on ShadowXeck: *profile* and *control*. In the usage scenarios here, the user first runs a target program in profile mode to log information on indirect jump instructions. This information includes the following:

- the current instruction pointer;
- the virtual address of a branch destination (destIP); and
- information on whether the virtual address of a branch destination is stored in a register or on memory. If it is stored on memory, the information also includes the virtual address of the branch source (srcIP).

In profile mode, the user needs to repeat each operation more than once for a target program, such as by executing commands and sending requests to a server program because destIP and srcIP can differ for each operation. Next, the user describes the security policy for the target program according to the above information obtained from execution in profile mode. Finally, the user runs the target program in control mode to control the behavior of the target OS kernel. During execution in control mode, if the current security policy regards a branch instruction as illegitimate, when in fact it is legitimate, the user needs to update the security policy accordingly.

PolicyFile	→	`defAction:` Action+ IndirectSpec+
IndirectSpec	→	`currIP:` *virtAddr* CondAct+
CondAct	→	`(destIP:(`*virtAddr* | `*),` `memLoc:(`*virtAddr* | `*),` `action:` Action+`)`
Action	→	`allow` | `fix(`*virtAddr*`)` | `raiseException` | `log`

Figure 4.3: Security policy syntax in ShadowXeck

Syntax

Figure 4.3 lists the security policy syntax. A security policy specifies how each indirect jump instruction should be controlled for different branch sources and destinations.

The `defAction:` field at the top level is followed by "Action", an action taken for indirect jump instructions that do not match any pattern. The rule part of "IndirectSpec" specifies pairs consisting of a condition and a response action ("CondAct") for an individual indirect jump instruction whose virtual address is specified by the part following the `currIP:` field. The condition part describes pairs consisting of the virtual addresses of a branch source and destination (the *virtAddr* sections following `memLoc:` and `destIP:`). The `*` in the condition part represents the case of being true at any time. When both *virtAddr* sections are specified by `*`, "CondAct" represents the specification for indirect jump instructions that do not match any condition.

("Action") specifies how an issued indirect jump instruction should be controlled. `allow` permits execution of the indirect jump instruction to continue. `log` logs information on the indirect jump instruction and continues its execution. `fix` means that the indirect jump destination is fixed with a specific virtual address (*virtAddr*), whereas `raiseException` means that ShadowXeck forces a general purpose exception immediately after restarting execution of the indirect jump instruction.

Generation

A ShadowXeck user describes security policies for each target program by using logs generated in profile mode. However, it is not easy even for users who are familiar with OS kernels to describe appropriate security policies corresponding to each execution log. To reduce the time and effort in description, ShadowXeck provides the following two command utilities.

- `mkpolicy`: This is a command for generating a policy template file. The user gives three input arguments and one output argument. The input arguments are the name of the execution log file and the response actions for "defAction" and "CondAct". The output argument is the name of the policy template file. When a branch

destination is stored in a register, *virtAddr* for `memLoc` is described as $\star$ in the policy rule. Furthermore, when the branch destinations for a pair of `currIP` and `destIP` are stored in a dynamically allocated memory region, the *virtAddr* parameters for `memLoc` are likely to be different. In this case, they are replaced with $\star$, and their elements are fused together to form one policy rule. The user manually updates the rules in a generated policy template, if necessary.

- `readlog`: This is a command for displaying the data in the execution log given as an input argument. This information includes the three components of indirect jump instructions, as described in Section 4.3.4. The command shows these components in order of interception or of a statistical summary for each indirect jump instruction. To show them comprehensively, data based on the virtual addresses included in execution are translated into a log output based on symbols such as function names. The symbol-based output, which is more comprehensive than the output based on virtual addresses, should help in describing security policies.

4.3.5 Usage

Control Commands

The following commands are provided for controlling the behavior of target OS kernels.

- `mkindirect`: This is a command for generating information on indirect jump instructions, which is part of the target OS information. The generated information includes the virtual addresses of memory locations where these instructions are issued.

- `sx_vconf`: This is a command for providing the SX-core with three types of target OS information: process management structures, system calls, and indirect jump instructions. Whereas information on indirect jump instructions is generated using `mkindirect`, information on process management structures and system calls is generated at the time of creating a target OS kernel image, as described in Section 2.3.1. SX-core manages the target OS information by using a hash value generated from the binary image of the target OS kernel. Consequently, the target OS information can be shared among target VMs that leverage the same binary image of an OS kernel.

- `sx_vcntl`: This command makes SX-core correlate a target VM with target OS information that has already been provided by `sx_vconf`. This command should be executed by the target VM. The protection mechanism for write-protected regions becomes active after execution of this command.

```
[cvm] $ mkindirect targetOS.img output_indirect.txt
[cvm] # sx_vconf targetOS.img \
          process_info.txt syscall_info.txt output_indirect.txt
[cvm] # sx_vcntl 1 targetOS.img
[cvm] # sx_vstart profile 1 target_prog output_prof.log
[cvm] $ mkpolicy output_prof.log log allow policy.txt
[cvm] # sx_vstart control 1 target_prog policy.txt
```

Figure 4.4: Usage example in ShadowXeck

- `sx_vstart`: This is a command for controlling target programs in profile or control mode. The command should be executed by the target program. The control mechanism for indirect jump instructions becomes active after execution of this command.

Usage Example

Figure 4.4 shows an example of the flow that continues until control using control mode starts. The target program in this example is `target_prog` running on a target VM whose ID is 1.

The user executes three commands before starting profile mode. First, the user generates `output_indirect.txt`, the file including information on indirect jump instructions, by issuing `mkindirect` with `targetOS.img`, the file with the target OS kernel image, as the input argument. Second, the user issues `sx_vconf` to provide SX-core with the target OS information on `targetOS.img`. The target OS information consists of three files containing information on process management structures (`process_info.txt`), system calls (`process_info.txt`), and indirect jump instructions (`output_indirect.txt`). Third, `sx_vcntl` links the target VM to the target OS information that has already been provided by `sx_vconf`. The user gives sx_vcntl the target VM ID (`1`) and `targetOS.img`, corresponding to the target OS kernel, as input arguments.

Henceforth, the user starts controlling the target VM in profile and control modes. First, the user runs ShadowXeck in profile mode by using `sx_vstart` to gather information on issued indirect jump instructions. `sx_vstart` is given three input arguments and one output argument: the operation mode, `profile`; the target VM ID, `1`; the name of the target program, `target_prog`; and the name of the file for logging issued indirect jump instructions, `output_prof.log`. After finishing the execution in profile mode, the user executes `mkpolicy` to create `policy.txt`, the file containing the policy template for `target_prog`, and updates `policy.txt` appropriately. The three input arguments for `mkpolicy` are `output_prof.log`; `log`, the response action when any policy con-

dition is not met, and `allow`, the response action when one of the policy conditions is met. Finally, the user starts operation in control mode by executing `sx_vstart`, with `control`, the target VM ID, `target_prog`, and `policy.txt` as inputs. Afterward, whenever `target_prog` is executed in the target VM, ShadowXeck controls the behavior of the target OS kernel associated with it. ShadowXeck provides a mechanism for controlling indirect jump instructions with `sx_vstart` in control mode, while enabling the protection mechanism for write-protected regions at the time of executing `sx_vcntl`.

4.3.6 Advantages

The proposed system has the following advantages.

Difficulty in disabling and abusing protection mechanisms: SX-core prevents corruption write-protected data and controls indirect jump instructions issued by target OS kernels, and it runs at a higher privilege level than do the target VMs. Furthermore, the target OS information and security policies are managed, and control commands are executed, in the control VM isolated from the target VMs. Consequently, if an attacker hijacks a target VM, the attacker cannot subvert the protection mechanisms at the VMM layer or abuse control commands and tamper with security policies in the control VM.

Optimization of controlling the behavior of OS kernels: The system controls the behavior of only target OS kernels in the context associated with target programs, because of the partial multiplexing in the kernel space. Furthermore, the control is based not on the hardware memory protection mechanism but on specific instructions, i.e., indirect jump instructions. As a result, there are no redundant page faults caused by control that is not based on the hardware memory protection mechanism. The multiplexing scheme is also useful from the perspective of hiding memory regions patched to force SX-core to intercept indirect jump instructions when the target OS kernel is runs in a context other than those of the target programs.

Automatic generation of information on OS kernels: Information on write-protected data and indirect jump instructions depends on binary images of the target OS kernels, as does information on process management structures and system calls. The system provides commands to dispense with the time and effort of generating the information on write-protected data and indirect jump instructions. Users need only issue these commands for the file of the target OS kernel image.

Maintenance of binary compatibilities: The system does not require users to modify the source code of target programs and target OS kernels.

Unified control over multiple VMs: In a usage model that distributes target VMs in which a target OS has already been installed, if the same target program runs on a target VM, the system can simultaneously control the behavior of the target OS kernels involved in the target program.

4.4 Implementation

We have designed and implemented ShadowXeck on an AMD64 processor architecture, using a para-virtualization version of Xen [20] 3.0.3 as the VMM, and a Linux OS kernel 2.6.16 as the target OS kernel.

4.4.1 Obtaining Information on OS Kernels

ShadowXeck requires two types of information on target OS kernels: the memory ranges for write-protected memory regions and information on indirect jump instructions as generated by `mkindirect`. This information is gathered using the file of the target OS kernel image, since the information depends on each instance of an OS kernel. The current prototype system supports ELF as an executable and x86 and x86-64 as the CPU architecture.

The ranges for write-protected memory regions are used to prevent tampering with code and read-only data within those regions. ShadowXeck reads the ELF program header from the file of an OS kernel binary image and extracts the virtual address ranges of ELF segments that do not have write permission, i.e., those for which `PF_W` is cleared. On the other hand, the information on indirect jump instructions is used to control those instructions when issued by target OS kernels. ShadowXeck extracts the information by using the `objdump` program with the binary image file. This includes the virtual addresses for the indirect CALL and JMP instructions.

4.4.2 Protection for Write-protected Regions

A target VM is forbidden from modifying write-protected regions in the VMM layer in order to protect code and read-only data contained within those regions. When `sx_vconf` is executed, the control daemon extracts information on the read-only memory ranges from the target OS kernel image file, which is one of `sx_vconf`'s arguments, and sends it to SX-core. Then, SX-core traverses the page tables of the currently running process and obtains physical addresses corresponding to the memory ranges. The kernel address space on Linux can be traversed from the top-level page table of the current process because the kernel address space is shared among all processes.

When `sx_vcntl` is executed, the protection mechanism is activated. On a para-virtualization version of Xen, the entries of the bottom-level page tables are updated via

the handler for a page fault exception or a hypercall, which is a software interrupt from a VM to a VMM. Thus, to protect write-protected regions, SX-core controls updates of their entries on the basis of virtual addresses obtained at the time of executing `sx_vcntl`. When an update is attempted on an entry of a bottom-level page table, SX-core checks whether granting of write permission is attempted on one of the physical pages for write-protected regions; that is, it checks whether the R/W bit is set in one of the entries. If the granting of write permission is attempted, SX-core makes the target VM generate a page fault exception.

4.4.3 Interception

Dynamic Binary Patching

To control the behavior of a target OS kernel running in the context of target programs, SX-core must capture two types of events that occur in the kernel space: indirect jump instructions and system calls. To achieve this, the mechanism of dynamic binary instrumentation, described in Section 2.3.2, is applied to the target OS kernel. SX-core overwrites the first byte of each instruction to be intercepted with the binary sequence of the HLT instruction. It needs to overwrite only that instruction, i.e., only the instruction to be intercepted, and it can intercept the instruction since HLT is a privileged instruction whose length is one. When the instruction overwritten by HLT is executed, SX-core emulates the original instruction and continues to execute the target VM.

There is another technique for intercepting any instruction. HookSafe [112] overwrites an intercepted instruction with a five-byte JMP instruction. However, if the length of the subsequent instruction is less than five, the subsequent instruction must also be overwritten. For example, if the subsequent instruction is the destination of another assembly routine, HookSafe's binary patching cannot overwrite it. If it was overwritten, it would become an invalid opcode.

Indirect Branch Instructions

Dynamic binary patching is applied to each indirect jump instruction when `sx_vcntl` is executed. The target OS information includes the virtual addresses of indirect jump instructions to be intercepted. SX-core traverses the currently running process and duplicates physical pages including indirect jump instructions. For each duplicated physical page, ShadowXeck overwrites indirect jump instructions with the HLT instruction. Furthermore, for each indirect jump instruction, SX-core saves a byte sequence consisting of the instruction and its length in order to emulate the instruction and execute the subsequent instruction. The duplicated physical pages are used when the address space of a target OS kernel is multiplexed.

In cooperation with control programs, SX-core logs all issued indirect jump instructions in profile mode or indirect jump instructions whose response action is `log` in control mode. The log data are transferred between SX-core and the control programs by using two facilities of Xen: shared memory between a VMM and the control VM, and the event channel.

Start and End of Execution and Process Creation

The proposed system needs to identify when a target program executes, spawns new processes, and terminates control of the kernel behavior at process granularity. To accomplish this, ShadowXeck performs control at the system-call level, i.e., at the entry and exit of system calls. On the para-virtualization version of Xen for the AMD64 architecture (Xen-AMD64), the VMM can intercept the entries of system calls because it must handle the SYSCALL instruction that set the current privilege level to zero (the highest). On the other hand, the facility of dynamic binary patching is applied in the same way for indirect jump instructions to intercept the exits of system calls. Binary patching is applied to the kernel address space not only with interception of indirect jump instructions but also that without interception. In this section, the kernel address space with interception of indirect jump instructions is referred to as *target kernel space*, and that without interception as the *original kernel space*. In addition, SX-core intercepts the exits of system calls on demand, i.e., only when a target program starts or spawns a new process.

When a target program starts inside a target VM with the `execve` system call, SX-core creates the corresponding target kernel space not at the entry but at the exit since a target OS kernel still does not create an original kernel space for a target program at the entry. At the entry, if a target program attempts to start, SX-core overwrites the binary code of the system call exit with the HLT instructions to intercept the `execve` exit. At the exit, SX-core multiplexes the kernel address space and overwrites the binary code of the system call exit with the original code. When a target program executes a new program, SX-core updates the target kernel space at the time of starting the target program above.

When a target program generates a new child process with system calls such as `fork` and `clone`, SX-core creates the target kernel space for the child process at the exit. As in the above `execve` case, SX-core uses dynamic binary patching to intercept system call exits on demand. When SX-core intercepts the first exit of a parent process running in the target kernel space or of a child process running in the original kernel space, it multiplexes the kernel address space for the child process and overwrites the binary code of the system call exits in the original kernel space with the original code.

When a target program terminates with the `exit_group` and `exit` system calls, SX-core releases the top-level page table for the target program.

4.4.4 Multiplexing Kernel Address Space

The kernel address space is partially multiplexed to control the behavior of target OS kernels only in the context of target programs. The multiplexing scheme is transparent to a target VM, which remains unaware of it. The context of ShadowXeck corresponds to the value of the CR3 control register containing the physical address of the current top-level page table. When SX-core intercepts a write operation to the CR3 register of a privileged instruction, it renders different views of the kernel address space according to the current value of the CR3 register. SX-core multiplexes only memory ranges that contain indirect jump instructions in the kernel space, for each the page.

ShadowXeck starts procedures for multiplexing when a target program is executed with the `execve` system call. First, SX-core duplicates the current top-level page table (ORIG_PT). Next, a newly created top-level page table (NEW_PT) is updated to intercept indirect jump instructions in the kernel space. SX-core traverses NEW_PT and replaces it with the physical address of a page containing the interception code for each entry of a bottom-level page table that contains indirect jump instructions. Moreover, SX-core manages the pair consisting of ORIG_PT and NEW_PT. Afterward, when switching of the current top-level page table to ORIG_PT is attempted, SX-core sets a physical address that does not point ORIG_PT but NEW_PT to the CR3 register. When the target program terminates, SX-core releases NEW_PT and the pair of ORIG_PT and NEW_PT. Furthermore, when the target program executes a different program, SX-core releases the current NEW_PT and the corresponding pair with ORIG_PT, and creates new ones. ShadowXeck intercepts only indirect jump instructions issued by target OS kernels running in the context using NEW_PT.

ShadowXeck can prevent attacks from threads running only in kernel mode, called *kernel threads*, which use the memory address space of a previously running process. If a malicious kernel thread running after a process belonging to a target program attempts to tamper with an indirect jump instruction, SX-core can prevent the attack since it runs in the memory address space that includes interception of indirect jump instructions. In addition, code and static data in the Linux kernel space should reside in the memory if the kernel is normally running. Therefore, unlike with the multiplexing scheme in ShadowXeck for the user space, SX-core does not need to save and restore physical pages including the interception code through page-out and page-in handling to prevent compromised target OS kernels from disabling the protection scheme.

4.5 Evaluation

We evaluated the feasibility of the proposed system from two viewpoints: its effectiveness against kernel-level malware, and the performance degradation introduced by

Table 4.1: Linux kernel-level rootkits

name	attack vector	target object
adore-ng	LKM	function pointer
Mood-NT	/dev/kmem	system call table
SucKIT2	/dev/kmem	kernel text
eNYeLKM	LKM	kernel text
Superkit	/dev/kmem	kernel text
phalanX	/dev/mem	system call table
Knark	/dev/kmem	system call table
KIS	LKM	system call table
Synapsys	LKM	system call table
override	LKM	system call table

LKM: loadable kernel module

ShadowXeck. In the experiments, ShadowXeck was run on a hardware platform with two dual-core AMD Opteron 2.8 GHz processors, 8 GB of RAM, and a 1 Gbps NIC. The control VM and the target VM were both configured with 4 CPUs and 1 GB of memory running a para-virtualized Linux OS kernel.

4.5.1 Effectiveness Against Kernel-level Malware

ShadowXeck controls the behavior of target OS kernels through two protection schemes: protecting write-protected data and controlling indirect jump instructions issued by target OSs. Whereas ShadowXeck prevents tampering with write-protected data at the VMM layer, it controls indirect jump instructions according to the security policies for target programs.

Modification of System Call Table

For write-protected data, the Linux system call table was protected. Many Linux kernel-level rootkits tamper with system calls, as indicated in Table 4.1. However, such kernel malware fails in tampering with entries of the system call table because the current Linux OS kernel prohibits their modification. For example, kernel version 2.6.0 or later prevents the installation of kernel-level rootkits that assume that the symbol of the system call table is available. Furthermore, kernel version 2.6.16 or later can also prevent installation of kernel-level rootkits that assume that the system call table is writable.

Even if a Linux OS kernel initializes the system call table as read-only data, attackers can tamper with its entries. They can create a new page table entry to translate another

virtual address into the physical address of the physical page containing the system call table. The R/W bit in the new page table entry is set, while that in the original page table entry remains to be cleared. This is a general technique for granting write permission to any read-only region. Therefore, attackers can also tamper with kernel text data by using this technique.

To demonstrate this, a malicious loadable kernel module, *MalLKM*, was implemented. MalLKM takes as an argument the virtual address of a system call table entry that it attempts to tamper with, *orig_vaddr*. First, MalLKM obtains an instance of the `page` kernel object corresponding to *orig_vaddr* with the `virt_to_page` macro. Next, it creates a writable memory region corresponding to the `page` instance by using the `vmap` function with `new_vaddr` as one of the arguments. Finally, MalLKM saves the value of the system call entry and overwrites it with pseudo-exploit code. This pseudo-exploit code informs the `/var/log/message` file that the attack has succeeded and executes the saved original code.

The experiment demonstrated that ShadowXeck prevented MalLKM from tampering with the system call table. First, MalLKM was installed on Xen, and it tampered with the `open` entry in the system call table. After this attack, MalLKM's message was output when `open` was invoked on Xen. Next, MalLKM was installed on ShadowXeck, and it attempted to tamper with the `open` entry. However, MalLKM failed to tamper with the entry, and the attempt caused a general protection fault.

Modification of Function Pointers

For indirect jump instructions, ShadowXeck protected the behavior of the target OS kernel involved in the system utility programs *ps*, *ls*, and *netstat*. We selected these programs since several types of current malware attempt to circumvent them to conceal the malware's malicious activities and existence. OS kernel behavior was modified with a kernel-level rootkit, *adore-ng*, and its user-level auxiliary program, *ava* [12]. adore-ng tampers with the function pointers of root and `/proc` file systems to hide processes, files, and network ports. In this experiment, adore-ng targeted the process of the current *bash* shell program, the *test* executable file under the home directory, and port number 2222 used by *nc* running in listen mode.

After the installation of adore-ng and hiding of three targets by ava, to simulate modification of OS kernel behavior, *ps*, *ls*, and *netstat* were executed on Xen. adore-ng tampered with the `lookup` function pointer of the `proc_root_inode_operations` variable to hide the three targets. To hide the *bash* process from *ps*, adore-ng tampered with the `readdir` function pointer of the `proc_root_operations` variable. To hide the *test* executable from *ls*, adore-ng tampered with the `readdir` function pointer of the `ext3_dir_operations` variable. To hide port number 2222 from *netstat*, adore-ng tampered with the `seq_show` function pointer of a `tcp_seq_afinfo` instance. As a result,

```
defAction : allow log
   currIP: 0xffffffff8010bd6d
     (destIP: 0xffffffff8010bdfc, memLoc: *, action: allow)
     (destIP: 0xffffffff8010c420, memLoc: *, action: allow)
     (destIP: 0xffffffff80117bb0, memLoc: *, action: allow)
   currIP: 0xffffffff80125a91
     (destIP: 0xffffffff80127a90, memLoc: *, action: allow)
   ...
   currIP: 0xffffffff8018f4b8  # do_lookup()
     (destIP: *, memLoc: 0xffffffff80366f48,
      action: fix(0xffffffff801b8280))  # proc_root_lookup()
     (destIP: *, memLoc: 0xffffffff803671a8,
      action: fix(0xffffffff801ba650))
     (destIP: *, memLoc: 0xffffffff80368588,
      action: fix(0xffffffff801bbad0))
   ...
   currIP: 0xffffffff80194bfd  # vfs_readdir()
     (destIP: *, memLoc: 0xffffffff80367030,
      action: fix(0xffffffff801b81a0))  # proc_root_readdir()
   ...
```

Figure 4.5: A part of the security policy for *ps* in ShadowXeck

the three targets could be hidden from *ps*, *ls*, and *netstat*.

Next, we demonstrated that ShadowXeck disabled the modifications by adore-ng. First, the target programs, *ps*, *ls*, and *netstat*, were run on ShadowXeck in profile mode to gather information on the indirect jump instructions issued by those programs. Second, security policies were generated for each target program by using the gathered information. Figure 4.5 shows a part of the security policy for *ps*. A part of the security policy for *netstat* is shown in Appendix C. All of the security policies specified that the `lookup` of `proc_root_inode_operations`, used in the `do_lookup` function, had to be fixed at `proc_root_lookup`, using the `fix` specification. Similarly the security policy for *ps* also specified that `readdir` of `proc_root_operations`, used in the `vfs_readdir` function, had to be fixed at `proc_root_readdir`, using the `fix` specification. The security policy for *ls* also specified that `readdir` of `ext3_dir_operations`, used in the `vfs_readdir` function, had to be fixed at `ext3_readdir`, using the `fix` specification. Finally, the security policy for *netstat* specified that if a `show` function pointer of a `seq_operations` instance, used in the `seq_read` function, was neither `unix_seq_show`, `raw_seq_show`, `tcp4_seq_show`, nor `udp4_seq_show`, then a general protection fault had to be gener-

ated using the `raiseException` specification. For `show`, we used `raiseException` because a `seq_operations` instance was dynamically allocated, and there were four possible instances of `show`. In the other cases, we used `fix` because each variable with a function pointer was statically allocated.

Finally, the targets of adore-ng were hidden in the same way as on Xen, and the target programs were executed on ShadowXeck by using the targets' security polices. *ps* and *ls* reported the existence of the *bash* process and the *test* file, respectively, while a general protection fault occurred and *netstat* failed to execute. Although function pointers were modified by adore-ng, the behavior of the target programs running on ShadowXeck was determined by the targets' security policies. In another respect, ShadowXeck permits users to leverage any kernel extension and thus maintains the flexibility of kernel extensions.

4.5.2 Impact on Performance

To clarify the overhead introduced by ShadowXeck, three kinds of application programs were used as target programs. First, the target programs were run in profile mode to collect the indirect jump instructions issued by the target OS kernel in the programs' context. The issued indirect jump instructions were collected when they launched, the execution of each benchmark program was repeated 10 times and terminated. Next, security policies were created from the collected data. Finally, the target programs were run using their security policies in control mode in order to measure their execution times.

The application programs are enumerated below with their experimental descriptions.

- System utility programs (*ps*, *ls*, *netstat*): These programs were executed without any arguments.
- Web servers (*thttpd*, *Apache*): To measure the throughput of each Web server, *ApacheBench* was run on another physical machine deployed in the same LAN as the physical machine on which ShadowXeck ran. The physical machine for ApacheBench was an Intel Pentium 4 3.2 GHz processor with hyper-threading enabled, 2 GB of RAM, and a 100 Mbps NIC. ApacheBench sent requests to obtain two kinds of static content (1- and 100-KB files) for each Web server. In addition, it sent requests to obtain dynamic content (CGI) for Apache. The CGI program (a *Perl* script) displayed information on the physical machines on which Apache and ApacheBench ran. Specifically, the *Perl* program displayed two CGI environmental variables (`HTTP_REFFER`, `HTTP_USER_AGENT`, and `REMOTE_ADDR`) and the result of the `gethostbyname` function. The number of both requests was configured as 128 times. Whereas thttpd handles all requests with one process, Apache processes them by using multiple threads.

- Anti-virus tool suite (*ClamAV* [2]): The time required for scanning 15 files (five of which were infected) was measured for two types of virus scanning programs: *clamscan* and *clamd/clamdscan*. First, a virus database file containing the virus signatures of the five infected files was created. The clamscan command-line program specified files or specified directories. The clamd daemon scanned files or directories at the request of the clamdscan client program. While clamscan read the virus database files for every request, clamd read them at start up or when receiving an update request from a user.

ShadowXeck only intercepts indirect jump instructions issued by target programs because of its memory multiplexing scheme. Therefore, it could be expected to reduce performance degradation for programs that ShadowXeck does not target. To confirm this, the execution times were also measured for benchmark programs not controlled by ShadowXeck. In addition, the benchmark programs were run using Linux OS kernels running on bare metal and on the *QEMU* CPU emulator. QEMU has been used to develop several previously proposed systems for analyzing, detecting, and preventing kernel-level malware [55, 63, 88, 89, 111, 121, 122]. The Linux on bare metal was configured with 1 GB of RAM. The Linux on QEMU was configured with four virtual processors and 1 GB of memory.

Figure 4.6 shows the experimental results for the system utility programs. Figures 4.7, 4.8, 4.9, 4.10, and 4.11 show the experimental results for Web servers. Figure 4.12 shows the experimental results for anti-virus tool suite. The binary image of the Linux OS kernel used as the target OS kernel included 1378 indirect jump instructions (1290 CALL instructions and 88 JMP instructions), and the number of 4 KB pages containing these instructions was 82.

The main factor in the higher performance degradation of the system utility programs was that the interposition of ibis accounted for a larger portion of execution than for the other benchmark programs. In addition, the number of indirect jump instructions issued was measured for a single execution of each of the system utility programs. The numbers for ps, ls, and netstat were approximately 5480, 1150, and 1480, respectively. This shows that the main reason for the higher overhead with ps than with ls and netstat was the amount of interposition of indirect jump instructions. For Web service throughput, the larger penalties for CGI on Apache were due to the additional handling for the `execve` system call each time CGI invoked `execve`. This handling includes identification of the execution name and memory multiplexing for the target programs. As expected, ShadowXeck controlled the target programs with lesser impact than on the other programs. Furthermore, it controlled the target programs with much smaller performance penalties relative to QEMU.

In summary, we believe that ShadowXeck could control the behavior of untrusted OS kernels within acceptable performance penalty ranges throughout all of the evaluations.

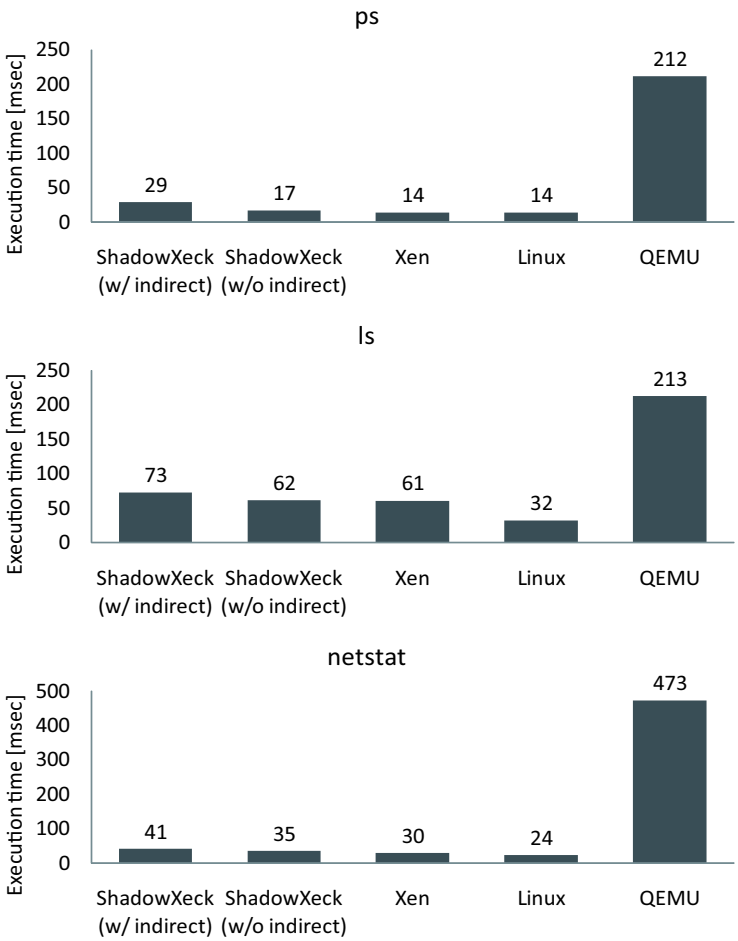

Figure 4.6: Execution time for system utility programs on Shadowall, Xen, Linux, and QEMU

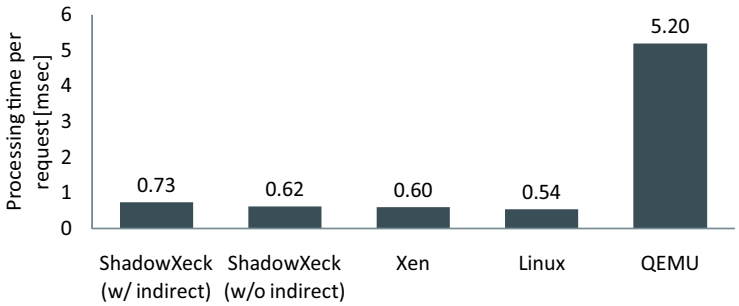

Figure 4.7: *thttpd* Web server throughput on ShadowXeck, Xen, Linux, and QEMU (file 1KB)

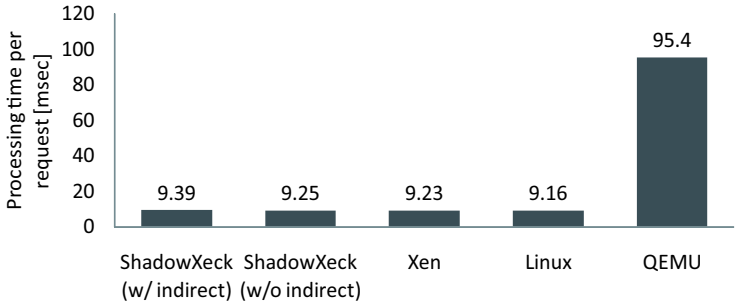

Figure 4.8: *thttpd* Web server throughput on ShadowXeck, Xen, Linux, and QEMU (file 100KB)

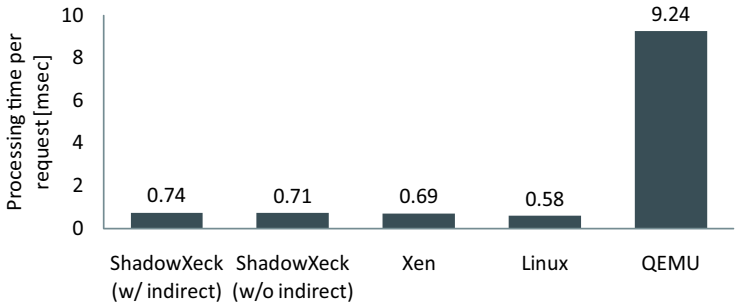

Figure 4.9: *Apache* Web server throughput on ShadowXeck, Xen, Linux, and QEMU (file 1KB)

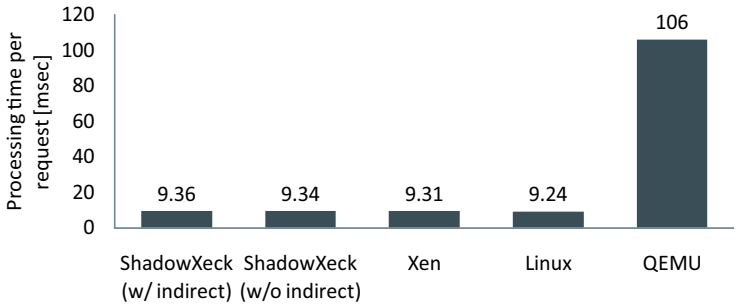

Figure 4.10: *Apache* Web server throughput on ShadowXeck, Xen, Linux, and QEMU (file 100KB)

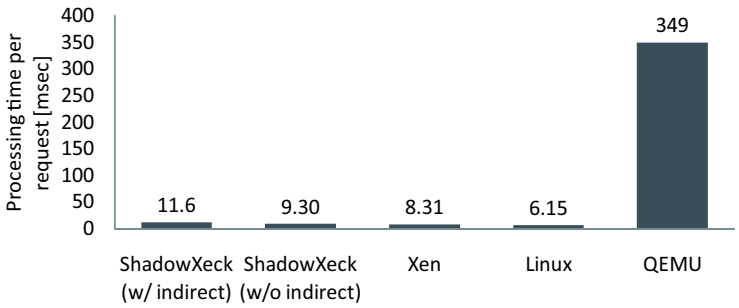

Figure 4.11: *Apache* Web server throughput on ShadowXeck, Xen, Linux, and QEMU (CGI)

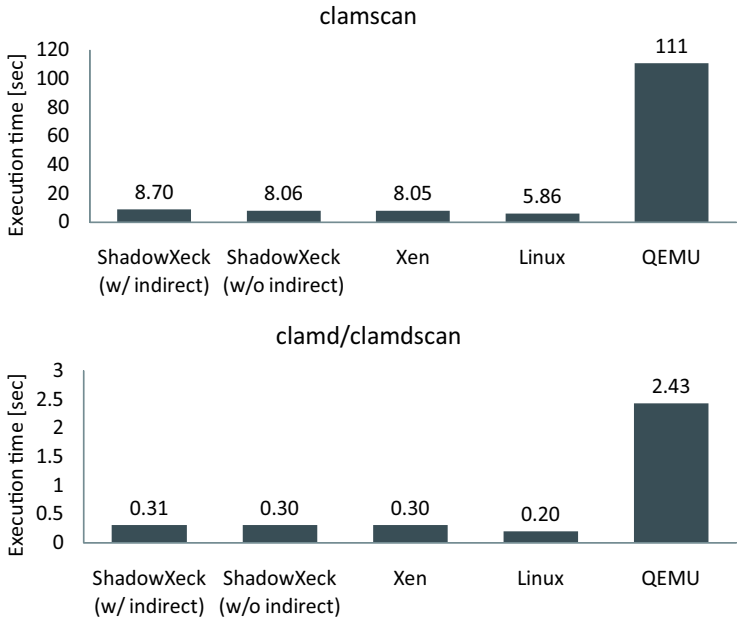

Figure 4.12: Virus scanning time on ShadowXeck, Xen, Linux, and QEMU

4.6 Related Work

There has been a variety of research on kernel-level malware, which can be divided into four categories.

4.6.1 Control of Kernel Behavior

HookSafe [112] is closely related system that controls the behavior of kernel-level malware by prohibiting modification of write-protected data in the kernel space and controlling function pointers in the kernel space. However, there are several differences in controlling function pointers between HookSafe and ShadowXeck. First, HookSafe controls them according to memory access in the data context, i.e., according to read and write operations for indirect jump destinations. On the other hand, ShadowXeck controls the pointers according to memory access in the instruction context, i.e., according to execution of indirect jump instructions. Second, HookSafe requires the source code of an OS kernel in order to collect memory locations from indirect jump destinations, whereas ShadowXeck does not require the source code because the memory locations from indirect jump instructions are collected of a binary image of an OS kernel. Third, the binary patching mechanism in HookSafe for intercepting operations on reading and writing in-

direct jump destinations can potentially overwrite subsequent instructions, since HookSafe overwrites the instruction that manipulates the indirect jump destination with a 5-byte JMP instruction. In comparison, ShadowXeck does not have this problem because an indirect jump instruction is overwritten with a 1-byte HLT instruction. Furthermore, the two systems differ in profiling. Whereas the purpose of profiling on HookSafe is to acquire instructions that manipulate the indirect jump destination, the purpose of profiling on ShadowXeck is to acquire indirect jump instructions issued by a target program.

Livewire [43] pioneered a virtual machine introspection (VM introspection) scheme that execution states at the OS level, such as processes and files, from outside an untrusted VM. Livewire provides an event-driven protection module for detecting tampering with security-sensitive memory regions, including kernel code and the system call table by maintaining these regions as read-only at the VMM layer. Lares [82] marks pages including protected data as read-only in order to monitor their modification. However, this page-granularity protection causes unnecessary page fault exceptions caused by writes to data that were not targeted in the same pages.

UCON$_{\mathcal{KI}}$ [117] provides an access control mechanism to protect kernel integrity. Meanwhile, ShadowXeck controls kernel behavior by forbidding modification of write-protected data and controlling indirect jump instructions. Whereas UCON$_{\mathcal{KI}}$ requires creation of security policies for access control from scratch, ShadowXeck can reduce the effort of generating security policies by using templates generated from execution logs. Furthermore, a UCON$_{\mathcal{KI}}$ prototype system based on QEMU adds significant overhead incurred by instruction emulation.

4.6.2 Prevention and Detection of Disapproved Kernel Code Execution

There have been several systems for preventing and detecting execution of kernel-level malware by forbidding execution of any unauthenticated kernel code [68, 88, 99]. NICKLE [88] provides different views depending on memory access contexts by multiplexing the kernel address space and permitting only authenticated code to be executed in the instruction context. In contrast, ShadowXeck multiplexes the kernel address space according to the contexts of application programs. Furthermore, unlike ShadowXeck, NICKLE does not provide a protection mechanism for read-only data. SecVisor [99] allows OS kernels to execute only authenticated code by applying hardware memory protection mechanisms. It provides a W$\oplus$X protection scheme in which authenticated kernel code is executable but not writable, while other, unauthenticated regions including kernel data, user code, and user data are made writable but not executable by controlling the R/W bits of page table entries and using their NX bits (No eXecute bits), on an AMD architecture. Patagonix [68] detects execution of unauthenticated kernel code by using the NX bits of page table entries and trusted databases of authenticated kernel code.

Unlike ShadowXeck, these systems require registration of kernel extensions, such as

loadable kernel modules, beforehand. In contrast, ShadowXeck controls indirect jump instructions in the contexts of target programs while preserving the flexibility of kernel extensions. Furthermore, these systems cannot control the behavior of authenticated code.

Kruegel et al. [62] detected kernel-level rootkits by analyzing binaries of loadable kernel modules (LKMs) at load time. The legitimacy of the LKMs is identified from whether they write to forbidden function pointers and whether they use and write to forbidden kernel symbols. The binary analysis can potentially limit the leveraging of kernel extensions of LKMs, since a legitimate LKM with the behavior close to that of kernel-level rootkits (e.g., the LKMs of anti-malware systems) can be falsely identified as an illegitimate LKM. Unlike in ShadowXeck, the binary analysis is performed in the same execution space as untrusted OSes. If attackers take over an untrusted OS with the binary analysis mechanism, they can subvert it. The same holds for an existing rootkit detection system, chkroot [27]. Furthermore, this approach cannot prevent attacks with special devices (/dev/mem, /dev/kmem).

4.6.3 Detection of Kernel-level Malware at Periodic Intervals or on Demand

Livewire also provides a polling protection module to detect tampering with the /proc file system interface used to acquire currently running processes. The module detects modification by comparing the interface acquired from outside the VM with the interface acquired inside the VM. VMwatcher [55] reconstructs the inner states of memory and virtual disks (e.g., processes and files) by using the VM introspection scheme, and it detects malware by comparing the reconstructed states outside a VM with those obtained through existing programs inside the VM. The reconstructed data are also used for execution of off-the-shelf anti-malware systems outside the VM.

Copilot [73] periodically detects malicious modification to data related to an untrusted OS kernel by using a PCI-based kernel monitor. Peroni et al. [59] proposed a system for detecting integrity violations in kernel objects (e.g., a process list) from security-relevant, user-specified constraints on the kernel objects. Protected kernel objects are checked asynchronously to determine whether their constraints hold. Whereas this system requires expert knowledge about an OS kernel in order to describe the constraints, ShadowXeck eases users' burden of policy description by providing a command for generating policy templates.

SBCFI [74] monitors kernel integrity at periodical intervals. The kernel integrity is checked according to a legitimate control flow graph generated from the source code of an OS kernel.

These systems do not control kernel behavior but instead detect modification of kernel object data periodically or on demand. Therefore, compromised OS kernels running on these systems are likely to execute malicious function pointers. In comparison, Shad-

owXeck can prevent compromised OS kernels from executing malicious function pointers at the time of execution.

4.6.4 Analysis of Kernel-level Malware

There are several systems for investigating the behavior of kernel-level malware [63, 89, 111, 121]. HookFinder [121], based on a dynamic tainting scheme, and K-Tracer [63], based on a dynamic slicing technique, analyze kernel-level malware to identify and extract how it hijacks an OS kernel's control flow, i.e., the hooking behavior of an OS kernel. PoKeR [89] also identifies information on modified kernel objects that are both statically and dynamically allocated. HookMap [111] identifies kernel hooks that can potentially be modified to hijack the kernel control flow through execution of anti-malware systems. HookMap's approach is based on the assumption that kernel-level malware will subvert anti-malware systems. Although all of these systems can precisely and easily obtain system-wide information on untrusted VMs by using the QEMU CPU emulator, they cause substantial performance degradation due to the instruction emulation by QEMU.

4.7 Summary

We have proposed ShadowXeck, a security system that controls the behavior of target OS kernels in the process contexts of target VMs. ShadowXeck controls the behavior of target programs without limiting the availability of kernel extensions and with minimal impact on the performance of programs that are not controlled. This is accomplished in two ways. First, the system prevents modification to read-only memory regions in kernel space, including code and read-only data. Second, it applies application-aware control to indirect jump instructions, i.e., indirect CALL and JMP instructions, issued by target OS kernels according to security policies. A security policy specifies which indirect jump instructions should be controlled and how they should be controlled. The system provides a command for automatically generating security policies.

Experimental evaluation demonstrated that Shadowall protected against two types of attacks on target OS kernels. The first type is tampering with the entries of a system call table, which are in read-only memory regions. The second type is tampering with the function pointers of the root and `/proc` file systems. Moreover, ShadowXeck could control target programs with much smaller performance penalties relative to a CPU emulator.

The results point toward two major, interesting aspects for future work. The first aspect is to provide a mechanism for controlling the behavior of loadable kernel modules (LKMs). In contrast with the code and data of OS kernels, the code and data of LKMs are dynamically loaded into the kernel space. To address this challenge, two functionalities

will be introduced into Shadowall. One is the functionality to control memory data at a relative address. The other functionality is to apply the dynamic binary instrumentation technique to LKMs at load time. The second aspect of the future work is to retrofit the security policy specification. As the experimental results for the effectiveness of ShadowXeck indicate, the destination of an indirect jump instruction cannot always be fixed. To ameliorate this problem, the specification will be extended. In the current specification, response actions are determined by only one indirect jump instruction at conditional expression. Instead, the system should support description of sequences of indirect jump instructions at a conditional expression, so that security policies can specify appropriate `fix` response actions for as many indirect jump instructions as possible.

Chapter 5

Conclusion

5.1 Summary of This Work

We have presented VMM-based security systems for enhancing the security of application programs from outside VMs. These systems effectively and efficiently create barriers for attacks on the security systems themselves, as well as on higher-level security systems running at the OS or application layer. The proposed systems *also* provide fine-grained control, i.e., control at application-program granularity. The systems identify application programs, control their behavior, and protect data relevant to the programs in OS-level semantics, while maintaining the two VMM properties of VM isolation and highest privilege control. More concretely, the proposed security systems *simultaneously* have four properties: (i) strong isolation from untrusted programs through VM isolation, (ii) highest privilege control of memory and virtual disk data, (iii) introspection at OS-level abstraction, and (iv) interposition on any events controlled by these systems. In addition, the users do not need to modify the source code of application programs.

Specifically, we have designed and implemented three security systems with different, non-overlapping security concerns for target application programs: *ShadowVox*, *Shadowall*, and *ShadowXeck*. ShadowVox monitors and controls the behavior of target application programs in user mode. Shadowall protects memory data in the user space of target application programs and file content related to these programs. ShadowXeck monitors and controls the behavior of target application programs in kernel mode.

ShadowVox controls the system calls executed by a target application program through two basic techniques: VM introspection and dynamic binary instrumentation. Even if target application programs themselves and privileged programs are taken over, ShadowVox prohibits compromised target programs from illegally executing system calls, and it prevents hijacked privileged programs from interfering with target application programs that normally run. VM introspection is a technique for bridging the semantic gap

between the OS-level semantic views of security systems and the hardware-level semantic views exported by a VMM. Dynamic binary instrumentation is a technique for intermediating any event at processor-instruction granularity. VM introspection and dynamic binary instrumentation are commonly used in all three of the proposed security systems. This work has shed light upon the information about OS kernels required for VM introspection, i.e., information about process management and system calls. Furthermore, a support program is provided for automatically generating data required for VM introspection. The proposed system requires the source code of target OS kernels since the control mechanism depends on information on process management and system calls. Note, however, that it does not require modifying the source code. Although the current system supports only Linux as a target OS kernel, the proposed approach can be applied to other OS kernels whose source code is open (e.g., UNIX-like OS kernels).

Shadowall protects memory and virtual disk data relevant to a target application program by concealing them from other, untrusted programs, including OS kernels, while leaving the target application program in the same VM as the untrusted programs. The system hinders victimized programs, including OS kernels, from enabling leakage and tampering with data involved with target programs. Toward this goal, it incorporates a memory protection scheme that provides different views according to execution modes, without cryptographic techniques, and it integrates the scheme with a file protection scheme that manages files and emulates file manipulations by outside VMs.

ShadowXeck controls the behavior of OS kernels by providing write-protection at page granularity and controlling indirect jump instructions. More precisely, the system controls OS kernels in the contexts of target application programs, while maintaining the availability of kernel extensions and lowering the impact on the performance of other, uncontrolled programs. It thwarts the illegitimate execution of OS kernel code and function pointers in the contexts of target programs. To enable application-aware control for indirect jump instructions, this work has provided a memory multiplexing scheme that generates different views according to the process context.

In this thesis, we have discussed experimental evaluation results for the proposed systems. The results showed that ShadowVox could control the behavior of a wide variety of application programs. They also demonstrated that ShadowVox prevented a compromised, privileged program from disabling ShadowVox's control mechanism for a Web server program running in the same VM as the compromised program. Next, the experiments demonstrated that Shadowall successfully disabled attacks attempting to modify the memory and virtual disk data of application programs. Finally, the experiments demonstrated that ShadowXeck disabled modification by kernel-level malware. As a results of the evaluations, we believe that these three systems can enhance the security of application programs within acceptable performance penalty levels.

5.2 Future Directions for This Work

There are several directions for this work in the future. First, the three proposed security systems do not deal with one remaining security concern: protection for writable data in the kernel space. None of the systems can hinder attackers from corrupting writable data for the purpose of stopping the security systems (e.g., modification of run queues to remove target application programs) through a methodology known as direct kernel object manipulation (DKOM). Attackers can also break the VM introspection technique, that is, they can cause the security systems to misidentify the inner states of VMs by corrupting process management data such as `task_struct` instances. We should provide a mechanism to protect target application programs from DKOM attacks.

Another interesting task for future work is to make the in-VMM mechanisms for controlling and protecting target application programs smaller, with the goal of reducing the VMM complexity. More concretely, parts of the in-VMM mechanisms that strongly depend on target OS kernels should be moved into target VMs. The VMM reinforces the security of the mechanisms inside the target VMs themselves.

Finally, the approaches described here can also be applied to VMMs based on hardware-assisted virtualization technologies such as Intel VT [29] and AMD-V [14]. The security systems should be implemented with attention to the differences of in memory management between software-based VMMs and those based on hardware-assisted virtualization. Whereas a software-based VMM resides in the address space of target VMs, a VMM based on hardware-assisted virtualization does not reside in a target VM's address space but in its own address space.

Appendix A

Security Policy Syntax of ShadowVox

The security policy in ShadowVox specifies which system calls are controlled and how they are controlled by using pattern matching of system call arguments. Figure A.1 shows the policy syntax in ShadowVox.

Here, "DefMacro" indicates the *path*, the path name of a header file defining the macros used in the policy rules. For example, a user can use the macro names `EPERM`, in place of the number `1` in the error code rule, and `SIGTERM`, in place of the number `15` in the signal rule.

The `default:` field at the top level indicates the response action taken for a system call that does not conform to any pattern. Subsequently, the syntax specifies the policy options, through "PolOption," and the policy rules for each system call, through "ModuleSpec". This appendix explains the rule for each system call, followed by its options.

The "ModuleSpec" section specifies how each system call is controlled. In ShadowVox's policy, system calls are classified into eleven groups, called *modules*. Each system call necessarily belongs to only one module. The `processMod` module includes system calls related to process operations (e.g., `execve`). The `fileMod` module includes system calls related to file operations (e.g., `open`). The `networkMod` module includes system calls related to network operations (e.g., `socket`). The `ipcMod` module includes system calls related to interprocess-communication operations (e.g., `semctl`). The `signalMod` module includes system calls related to signal operations (e.g., `sigaction`). The `fsMod` module includes system calls related to file-system operations (e.g., `getdents`). The `idMod` module includes system calls related to ID operations (e.g., `getuid`). The `memoryMod` module includes system calls related to memory operations (e.g., `mmap`). The `systemMod` module includes system calls related to system-management operations (e.g., `uname`). The `timeMod` module includes system calls related to time operations (e.g., `nanosleep`). Lastly, the `unimplementedMod` module includes system calls that are not implemented (e.g., `vserver`). In addition, the `default:` field in "ModuleSpec" signi-

PolicyFile	→	DefMacro* `default:` DefAction PolOption* ModuleSpec*
DefMacro	→	`includeMacro(`*path*`)`
DefAction	→	Action | `skip`
Action	→	`allow` | `deny(`*errno*`)` | `ask` | `killProc(` *signame*`)`
		| `killThread(`*signame*`)` | `policyChange(`*policyfile[,logfile]*`)`
PolOption	→	`execByPtracingProc:` PtAction | `signalMask(`*signames*`)`
		| `controlChild: detachProc`
PtAction	→	`detachProc` | `createNewMonitor`
ModuleSpec	→	ModuleName `default:` DefAction SysCallSpec*
ModuleName	→	`processMod` | `fileMod` | `networkMod` | `ipcMod` | `signalMod`
		| `fsMod` | `idMod` | `memoryMod` | `systemMod` | `timeMod`
		| `unimplementedMod`
SysCallSpec	→	*syscallName* `default:` DefAction ControlExpr*
ControlExpr	→	Cond* Action
Cond	→	ProcessCond | FileCond | NetworkCond | IdCond
		| MemoryCond | CurrIdCond | `argEq(`*argnum,value*`)`
		| Cond `and` Cond | Cond `or` Cond
ProcessCond	→	`cloneFlagsEq(` *cloneflags*) | `ptraceRequest(`*requests*)
FileCond	→	`fileEq(` *argnum,path*`)` | `filePrefixEq(`*argnum,pathprefix*`)`
		| `fileFlagsEq(`*fileflags*`)`
NetworkCond	→	`socketDomainEq(`*domain*`)` | `socketTypeEq(`*socktype*`)`
		| `socketProtocolEq(`*sockprot*`)` | `ipaddrEq(`*ipaddr*`)`
		| `netaddrEq(`*netaddr*`)` | `portEq(`*portnum*`)`
		| `unixsockEq(`*unixsock*`)` | `unixsockPrefixEq(`*unixsock*`)`
		| `ifindexEq(`*ifindex*`)` | `packetTypeEq(` *packettype*`)`
		| `sockoptEq(`*level,optname*`)`
IdCond	→	`uidEq(`*idnum*`)` | `gidEq(`*idnum*`)`
		| `euidEq(`*idnum*`)` | `egidEq(`*idnum*`)`
MemoryCond	→	`memProtEq(`*protections*`)` | `memFlagsEq(`*memflags*`)`
CurrIdCond	→	`uidEq(`*idnum*`)` | `gidEq(`*idnum*`)`
		| `euidEq(`*idnum*`)` | `egidEq(`*idnum*`)`

Figure A.1: Security policy syntax in ShadowVox

fies the response action taken for a system call in a particular module that does not match any pattern.

Next, the "SysCallSpec" section indicates how each discrete system call is controlled. The *syscallName* field specifies the name of a system call. The `default:` field in "SysCallSpec" signifies the response action taken for a case that does not match any pattern of system call arguments. The "ControlExpr" label specifies patterns of system call arguments that ShadowVox controls ("Cond") and the response actions taken for cases matching the specified patterns ("Action").

The argument patterns ("Cond") include patterns of process-related system calls ("ProcessCond"), of file-related system calls ("FileCond"), of network-related system calls ("NetworkCond"), of ID-related system calls ("IdCond"), and of memory-related system calls ("MemoryCond"). The "currIdCond" label represents a pattern of the current user, group ID, effective user, or effective group ID. The `argEq` pattern is used to specify an arbitrary pattern with a pair consisting of an argument number *argnum* and a value *value*.

In the response action ("Action"), `allow` and `skip` permit execution of the system call. For `allow`, the SV-core notifies a control program in the control VM. For `skip`, on the other hand, it does not notify the control program. For `deny`, the execution of an intercepted system call fails with an error code name *errno*. For `ask`, the user determines a response action at runtime, from among those other than `ask`. For `killProc` and `killThread`, the target process and target thread, respectively, receive the signal *signame*. Finally, for `policyChange`, the current policy is replaced with the policy of the *policyfile* file.

ShadowVox provides three types of policy options. The first option is `execByPtracingProc`, a rule for when the `execve` system call is executed by a target process that controls the behavior of child processes by using the `ptrace` system call. Systems and programs using `ptrace` (e.g., Systrace [86] and the *strace* command-line program) monitor arbitrary programs specified by their users. Thus, the policies for systems and programs using `ptrace` also depend on the system calls executed by the programs monitored by `ptrace`. To address this, ShadowVox provides `execByPtracingProc`, which enables the user to describe policies that are independent of the behavior of programs monitored by `ptrace`. The user specifies `detachProc` or `createNewMonitor` as the `execByPtracingProc` response action. When `detachProc` is specified, ShadowVox does not control the program monitored by `ptrace`. When `createNewMonitor` is specified, ShadowVox launches a new control program for the program monitored by `ptrace`. Note that the user needs to register the security policy for the program monitored by `ptrace` beforehand using the `sv_vstart` command. The second policy option, `signalMask`, specifies which signals sent to a target program from other programs have to be disabled. *signames* indicates the names of signals to be disabled. Even if an attacker sends a `SIGKILL` signal to a target program, ShadowVox can prevent the sending of

`SIGKILL` by specifying `signalMask(SIGKILL)` in the security policy for the target program. Unlike ShadowVox, systems using `ptrace` cannot disable signals from processes that the security systems do not target, since these systems monitor only the system calls executed by the target program. The third policy option, `controlChild : detachProc`, indicates the child processes that are not controlled.

Appendix B

A Part of Security Policy for *Apache* in ShadowVox

B.1 For IA-32

```
### apache.pol
includeMacro("asm-generic/fcntl.h")
includeMacro("linux/socket.h")
...
default: allow
processMod default: ask
   execve default: deny(EPERM)
      fileEq(1,"demo-cgi.cgi") and currUidEq(33)
         policyChange("cgi.pol")
   clone skip
   ...
fileMod default: ask
   open default: ask
      filePrefixEq(1,"/etc/apache2/") and fileFlagsEq(O_RDONLY)
         allow
      fileEq(1,"/etc/apache2/conf.d")
         and fileFlagsEq(O_NONBLOCK|O_LARGEFILE
                         |O_DIRECTORY|O_NDELAY)
         allow
      ...
   poll default: skip
```

```
   ...
networkMod   default: ask
   socket default: ask
      socketDomainEq(AF_INET)
        and socketTypeEq(SOCK_STREAM) allow
      ...
   bind   default: ask
      sockaddrFamilyEq(AF_INET) and portEq(80) allow
...

### cgi.pol (xxx.xxx.xxx.xxx : IP address)
includeMacro("asm-generic/fcntl.h")
includeMacro("linux/socket.h")
includeMacro("asm-generic/mman.h")

default: ask
fileMod default: ask
   open default: ask
      fileEq(1,"demo-cgi.cgi")
         and fileFlagsEq(O_LARGEFILE)
         and currUidEq(33) allow
   ...
networkMod default: ask
   connect default: ask
      socketDomainEq(AF_INET)
         and ipaddrEq(xxx.xxx.xxx.xxx)
         and portEq(53)
         and currUidEq(33) allow
      ...
   ...
memoryMod default: ask
   mmap default: ask
      memProtEq(PROT_READ|PROT_EXEC)
         and memFlagsEq(MAP_PRIVATE)
         and currUidEq(33) allow
      ...
   ...
timeMod   default: ask
   gettimeofday default: skip
   time skip
```

```
...
```

B.2 For AMD64

```
### apache.pol
includeMacro("asm-generic/fcntl.h")
includeMacro("linux/socket.h")
...

default: allow
processMod default: ask
   execve default: deny(EPERM)
      fileEq(1,/home/koichi/htdocs/cgi-bin/demo-cgi.cgi)
         and currUidEq(33)
         policyChange(cgi_skip.pol,log.cgi_skip)
      clone default: allow
         cloneFlagsEq(CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID)
         allow
         cloneFlagsEq(CLONE_VM|CLONE_FS|CLONE_FILES
                      |CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM
                      |CLONE_SETTLS|CLONE_PARENT_SETTID
                      |CLONE_CHILD_CLEARTID|CLONE_DETACHED)
         and currUidEq(33)
         allow
      set_tid_address default: skip
      arch_prctl default: skip
      ...
fileMod default: ask
   open default: ask
      filePrefix(1,/etc/apache2/) and fileFlagsEq(O_RDONLY)
         allow
      fileEq(1,/etc/apache2/conf.d/)
         and fileFlagsEq(O_NONBLOCK|O_DIRECTORY|O_NDELAY)
         allow
      ...
   poll default: skip
   ...
```

```
fsMod default: ask
chdir default: ask
fileEq(1,/home/koichi/htdocs/cgi-bin/) and currUidEq(33) allow

networkMod default: ask
   socket default: ask
      socketDomainEq(AF_INET)
         and socketTypeEq(SOCK_STREAM) allow
         ...
      bind default: ask
         sockaddrFamilyEq(AF_INET) and portEq(80) allow
...

### cgi.pol (xxx.xxx.xxx.xxx : IP address)
includeMacro("asm-generic/fcntl.h")
includeMacro("linux/socket.h")
includeMacro("asm-generic/mman.h")

default: ask
fileMod default: ask
   open default: ask
      fileEq(1,/home/koichi/htdocs/cgi-bin/demo-cgi.cgi) allow

networkMod default: ask
   connect default: ask
      sockaddrFamilyEq(AF_INET)
         and ipaddrEq(xxx.xxx.xxx.xxx)
         and portEq(53) allow
      ...
   ...
memoryMod default: ask
   mmap default: ask
      memProtEq(PROT_READ|PROT_EXEC)
         and memFlagsEq(MAP_PRIVATE) allow
      ...
   ...
timeMod default: ask
   gettimeofday default: skip
   time skip
...
```

Appendix C

A Part of Security Policy for *netstat* in ShadowXeck

```
defAction : allow log
   currIP: 0xffffffff8010bd6d
     (destIP: 0xffffffff8010bdfc, memLoc: *, action: allow)
     (destIP: 0xffffffff8010c420, memLoc: *, action: allow) ...
   ...
 currIP: 0xffffffff8018f4b8  # do_lookup()
     (destIP: *, memLoc: 0xffffffff80366f48,
      action: fix(0xffffffff801b8280))  # proc_root_lookup()
     (destIP: *, memLoc: 0xffffffff8036a528,
      action: fix(0xffffffff801bbad0))  # proc_lookup()
     (destIP: *, memLoc: 0xffffffff803671a8,
      action: fix(0xffffffff801cd280))  # ext3_lookup()
   ...
   currIP: 0xffffffff801a38f1  # seq_read()
     # udp4_seq_show()
     (destIP: 0xffffffff802cf740, memLoc: *, action: allow)
     # tcp4_seq_show()
     (destIP: 0xffffffff802c82b0, memLoc: *, action: allow)
     # raw_seq_show()
     (destIP: 0xffffffff802ce380, memLoc: *, action: allow)
     # unix_seq_show()
     (destIP: 0xffffffff802e4920, memLoc: *, action: allow)
     (destIP: *, memLoc: *, action: raiseException)
   ...
```

The `fix` response actions are specified at the function pointer *do_lookup()* (`currIP: 0xffffffff8010bd6d`) because the destination addresses are uniquely determined. On the other hand, the `raiseException` response action is specified at the function pointer (*seq_read* (`currIP: 0xffffffff801a38f1`) because the destination address cannot be uniquely determined.

Bibliography

[1] *Adobe Flash player code execution vulnerability (VU#395473).* http://www.kb.cert.org/vuls/id/395473.

[2] *Clam AntiVirus.* http://www.clamav.net/.

[3] *Debian and Ubuntu OpenSSL packages contain a predictable random number generator (VU#925211).* http://www.kb.cert.org/vuls/id/925211.

[4] *Linux Intrusion Detection System.* http://www.lids.org/.

[5] *Linux Kernel "exit_notify()" CAP_KILL Verification Local Privilege Escalation Vulnerability.* http://securityfocus.com/bid/34405/.

[6] *Microsoft Security Bulletin MS09-065.* http://www.microsoft.com/technet/security/bulletin/MS09-065.mspx.

[7] *Remote Code Execution in Samba's nmbd (CVE-2007-5398).* http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-5398.

[8] *Snort - the de facto standard for intrusion detection/prevention.* http://www.snort.org/.

[9] *Sophos Anti-Virus.* http://www.sophos.com/.

[10] *Building a Secure Computing System.* Van Nostrand Reinhold, 1988.

[11] A. Acharya and M. Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *Proceedings of the 9th USENIX Security Symposium*, Denver, August 2000.

[12] adore-ng. *Announcing full functional adore-ng rootkit for 2.6 Kernel*, 2004. http://www.lwn.net/Articles/75991/.

[13] National Security Agency. *Security-Enhanced Linux.* http://www.nsa.gov/research/selinux/.

[14] AMD. *AMD Virtualization Technology*. `http://www.amd.com/us/products/technologies/virtualization/Pages/virtualization.aspx`.

[15] AMD. AMD Platform for Trustworthy Computing. In *Windows Hardware Engineering Conference 2003*, New Orleans, May 2003.

[16] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, May 1997.

[17] 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*, Ottawa, June 2006.

[18] J. Athey, C. Ashworth, F. Mayer, and D. Miner. Towards Intuitive Tools for Managing: Hiding the Details but Retaining the Power. In *Proceedings of the 2007 Security Enhanced Linux Symposium*, Baltimore, March 2007.

[19] A. Baliga, P. Kamat, and L. Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, Oakland, May 2007.

[20] 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*, New York, October 2003.

[21] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow Anomaly Detection. In *Proceedings of the 2006 USENIX Symposium on Security and Privacy*, Oakland, May 2006.

[22] D. Brumley and D. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, August 2004.

[23] M. Carbone, D. Zamboni, and W. Lee. Taming Virtualization. *IEEE Security and Privacy*, 6(1):65–67, 2008.

[24] P. Chen and B. Noble. When Virtual Is Better Than Real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, Schloss Elmau, May 2001.

[25] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, August 2005.

[26] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, March 2008.

[27] chkroot. *chkroot – locally checks for signs of rootkit*. `http://www.chkrootkit.org`.

[28] Intel Corporation. *Intel Trusted Execution Technology*. `http://www.intel.com/technology/security/`.

[29] Intel Corporation. *Intel Virtualization Technology*. `http://www.intel.com/technology/virtualization/`.

[30] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003.

[31] C. Cowan, S. Beattle, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *Proceedings of the 14th Conference on Systems Administration*, New Orleans, December 2000.

[32] K. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, January 1998.

[33] D. Zovi. Hardware Virtualization Based Rootkits, August 2006. Blackhat 2006.

[34] Dazuko project. *A Stackable Filesystem to Allow Online File Access Control*. `http://dazuko.dnsalias.org/wiki/index.php/Main_Page/`.

[35] F. Bellard. *QEMU CPU Emulator*. `http://fabrice.bellard.free.fr/qemu/`.

[36] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing Sensitivity in Static Analysis for Intrusion Detection. In *Proceedings of the 2006 USENIX Symposium on Security and Privacy*, Oakland, May 2004.

[37] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, Oakland, May 2003.

[38] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, May 1996.

[39] G. Hoglund. *Kernel Object Hooking Rootkits*. http://www.rootkit.com/newsread.php?newsid=501/.

[40] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, San Diego, February 2003.

[41] 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*, New York, October 2003.

[42] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the 11th Annual Network and Distributed System System Security Symposium*, San Diego, February 2004.

[43] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium*, San Diego, February 2003.

[44] J. Giffin. *MODEL-BASED INTRUSION DETECTION SYSTEM DESIGN AND EVALUATION*. PhD thesis, University of Wisconsin-Madison, 2006.

[45] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, July 1996.

[46] T. Harada, T. Horie, and K. Tanaka. Access policy generation system based on process execution history. Network Security Forum 2003, http://sourceforge.jp/projects/tomoyo/docs/nsf2003-en.pdf.

[47] B. Hayes. Cloud Computing. *Communications of ACM*, 51(7):9–11, 2008.

[48] Hewlett-Packard. *NetTop*, 2004. http://www.hp.com/hpinfo/newsroom/press_kits/2004/security/ps_nettopbrochure.pdf.

[49] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[50] J. Butler. *Dynamic Kernel Object Manipulation*. http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf.

[51] J. Rutkowska. Subverting Vista Kernel For Fun And Profit, August 2006. Blackhat 2006.

[52] T. Jaegar, R. Sailer, and X. Zhang. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, June 2001.

[53] T. Jaegar, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003.

[54] S. Jain, F. Shafique, V. Djeric, and A. Goel. Application-Level Isolation and Recovery with Solitude. In *Proceedings of the 3rd European Conference on Computer Systems*, Glasgow, May 2008.

[55] 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*, Alexandria, October 2007.

[56] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the 2006 USENIX Annual Technical Conference*, Boston, May 2006.

[57] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification using Lycosid. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Seattle, March 2008.

[58] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, October 2005.

[59] N. Petroni Jr, T. Fraser, A. Walters, and W. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver,B.C., August 2006.

[60] G. Kim and E. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, Fairfax, November 1994.

[61] S. King, P. Chen, Y. Wang, C. Verbowski, H. Wang, and J. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, Oakland, May 2006.

[62] C. Kruegel, W. Robertson, and G. Vigra. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*, Tucson, AZ, December 2004.

[63] A. Lanzi, M. Sharif, and W. Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, San Diego, February 2009.

[64] L. Li, J. Just, and r. Sekar. Address-Space Randomization for Windows Systems. In *Proceedings of the 22th Annual Computer Security Applications Conference*, Miami Beach, December 2006.

[65] D. Lie, C. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, New York, October 2003.

[66] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowits. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, November 2000.

[67] Linux. *Kernel Based Virtual Machine*. `http://www.linux-kvm.org/pub/Main_Page/`.

[68] L. Litty, H. Largar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[69] J. McCune, T. Jaeger, S. Berger, R. Caceres, and R. Sailer. Shamon: A System for Distributed Mandatory Access Control. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, Miami Beach, December 2006.

[70] Trend Micro. *Internet Security*. `http://us.trendmicro.com/us/products/personal/internet-security/`.

[71] Microsoft. *Hyper-V Server*. `http://www.microsoft.com/hyper-v-server/en/us/`.

[72] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[73] Jr N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, August 2004.

[74] Jr N. Petroni and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, November 2007.

[75] J. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

[76] Inc Novell. *AppArmor Application Security for Linux*. `http://www.novell.com/linux/security/apparmor/`.

[77] Oracle Corp. *Oracle VM*. `http://www.oracle.com/us/technologies/virtualization/`.

[78] Parallels. *Parallels Server*. `http://www.parallels.com/products/server/`.

[79] C. Parampalli, R. Sekar, and R. Johnson. A Practical Mimicry Attack Against Powerful System-Call Monitors. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, Tokyo, March 2008.

[80] S. Patil, A. Kashyap, G. Sivathanu, and E. Zadok. I^3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Conference on Security Administration*, Atlanta, November 2004.

[81] B. Payne, M. Carbone, and W. Lee. Secure and Flexible Monitoring of Virtual Machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference Symposium on Information, Computer and Communications Security*, Florida, December 2007.

[82] B. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, Oakland, May 2009.

[83] G. Popek and R. Goldberg. Formal Requirements for Virtualizable Third Generation Architecture. *Communications of the ACM*, 17(7):412–421, 1974.

[84] S. Potter, J. Nieh, and M. Selsky. Secure Isolation of Untrusted Legacy Applications. In *Proceedings of the 21st Large Installation System Administration Conference*, San Diego, November 2007.

[85] ProFTPD. *ProFTPD - Highly configurable GPL-licensed FTP server software*. `http://www.proftpd.org/`.

[86] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003.

[87] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting. System Call Monitoring Using Authenticated System Calls. *IEEE Transactions on Dependable and Secure Computing*, 3(3):216–229, 2006.

[88] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, Cambridge, September 2008.

[89] R. Riley, X. Jiang, and D. Xu. Multi-Aspect Profiling of Kernel Rootkit Behavior. In *Proceedings of the 4th European Conference on Computer Systems*, Nuemberg, April 2009.

[90] J. Robin and C. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, Denver, August 2000.

[91] N. Rosenblum, G. Cooksey, and B. Miller. Virtual Machine-Provided Context Sensitive Page Mappings. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Seattle, March 2008.

[92] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. Doorn. Building a MAC-based Security Architecture for the Xen Opensource Hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference*, Tucson, December 2005.

[93] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. Technical Report RC233511, IBM Research, 2005.

[94] J. Sawazaki. Detecting Creation of Processes and Execution of Applications without Modifying Guest OS Kernel on Virtual Machine Monitor, 2008. *Senior Thesis*.

[95] P. Saxena, R. Sekar, and V. Puranik. Efficient Fine-Grained Binary Instrumentation with Applications to Taint-tracking. In *Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization*, Boston, April 2008.

[96] Secunia. *Linux Kernel "vmsplice()" System Call Vulnerabilities*, 2008. `http://secunia.com/advisories/28835/`.

[97] SecurityFocus. *Linux Kernel "exit_notify()" CAP_KILL Verification Local Privilege Escalation Vulnerability*, 2009. `http://securityfocus.com/bid/34405/`.

[98] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Oakland, May 2001.

[99] 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 21th ACM Symposium on Operating Systems Principles*, Stevenson, October 2007.

[100] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. In *Proceedings of the 1st European Conference on Computer Systems*, Leuven, April 2006.

[101] Sun Microsystems, Inc. *Virtual Box*. `http://www.virtualbox.org/`.

[102] Symantec. *Norton Anti-Virus*. `http://www.symantec.com/norton/`.

[103] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, November 2006.

[104] VMware. *VMware ACE*. `http://www.vmware.com/products/ace/`.

[105] VMware, Inc. *VMware ESX/ESXi Server*. `http://www.vmware.com/products/esx/`.

[106] M. Vouk. Cloud Computing - Issues, Research and Implementations. *Journal of Computing and Information Technology*, 16(4):235–246, 2008.

[107] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Oakland, May 2001.

[108] D. Wagner and P. Sato. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, Washington, DC, November 2002.

[109] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, December 1993.

[110] Y. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealthy Software with Stride GhostBuster. In *Proceedings of the 2005 International Conference on Dependable System and Networks*, Yokohama, June 2005.

[111] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering Persistent Kernel Rootkits Through Systematic Hook Discovery. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, Cambridge, September 2008.

[112] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, Chicago, November 2009.

[113] R. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *Proceedings of the 1st USENIX Workshop on Offensive Technologies*, Boston, August 2007.

[114] J. Wei, B. Payne, J. Giffin, and C. Pu. Soft-Timer Driven Transient Kernel Control Flow Attacks and Defense. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, Boston, December 2008.

[115] C. Weinhold and H. Härtig. VPFS: Building a Virtual Private File System with a Small Trusted Computing Base. In *Proceedings of the 3rd European Conference on Computer Systems*, Glasgow, May 2008.

[116] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 195–209, Boston, December 2002.

[117] M. Xu, X. Jiang, R. Sandhu, and X. Zhang. Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, Sophia Antipolis, June 2007.

[118] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, July 2006.

[119] J. Yang and K. Shin. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Seattle, March 2008.

[120] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, Oakland, May 2009.

[121] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, February 2008.

[122] H. Yin and D. Song. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, November 2007.

[123] X. Zhao, K. Borders, and A. Prakash. Towards Protecting Sensitive Files in Compromised System. In *Proceedings of the 3rd International IEEE Security in Storage Workshop*, San Francisco, December 2005.