The Pennsylvania State University
The Graduate School

PRESERVING SYSTEM INTEGRITY IN COMMODITY COMPUTERS

A Dissertation in
Computer Science and Engineering
by
Xi Xiong

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2012

The dissertation of Xi Xiong was reviewed and approved* by the following:

Peng Liu
Professor of Information Science and Technology
The Pennsylvania State University
Dissertation Advisor, Chair of Committee

Sencun Zhu
Associate Professor of Computer Science and Engineering
The Pennsylvania State University
Co-Chair of Committee

Trent Jaeger
Associate Professor of Computer Science and Engineering
The Pennsylvania State University

David Miller
Professor of Electrical Engineering
The Pennsylvania State University

Lee Coraor
Associate Professor of Computer Science and Engineering
The Pennsylvania State University
Director of Graduate Affairs

*Signatures are on file in the Graduate School.

Abstract

Today people rely more and more on commodity computer systems for storing and processing information. To make computer systems more trustworthy, it is highly demanding that these systems could have integrity protection mechanism as the security basis of computing. In this dissertation, we propose proactive and reactive approaches to preserve system integrity for commodity computer systems.

First, we explore reactive techniques to recover OS-level objects (e.g., processes and files) in an intruded computer system which already has integrity compromise. We design and implement SHELF, an intrusion recovery system that aims to preserve business continuity, availability and recovery accuracy. SHELF tracks activities of a computer system so that it can precisely determine which object of the system is compromised upon given an infection symptom. During the recovery phase, SHELF preserves accumulated clean state of infected objects, and it helps benign objects maintain their availability level to reduce system downtime.

The effort of repairing OS-level applications and files, however, must depends on a trusted and uncompromised OS kernel to provide correct functionality and abstractions. As commodity OS kernels are more and more becoming favorable targets for attackers, it is necessary to have proactive protection mechanism to secure the OS kernel and provide solid foundation for use-space security approaches. We study the problem of securing untrusted code executing in the kernel space, which is the major venue for OS kernel integrity compromise. We design and implement HUKO, a hypervisor-based integrity protection system that protects commodity OS kernels from untrusted extensions. In HUKO, untrusted extensions can safely run in the kernel space to provide desired functionality, but they are also confined by access control mechanisms, which significantly limit the attacker's ability to compromise the integrity of OS kernel.

Based on the hypervisor architecture provided by HUKO, we further propose SILVER, a comprehensive framework that offers transparent protection domain

primitives to achieve fine-grained access control and secure communication between programs in OS kernel. SILVER provides OS kernel developers the ability to specify security properties of their own code and data at the granularity of individual functions and data objects. Moreover, SIVER helps developers to prevent attacks exploiting kernel program communication, which cannot be effectively handled by typical isolation systems. To achieve such mechanism, we propose a novel resource management scheme of kernel data objects according to their security properties. Based on this organization, SILVER enforces access control and communication safety using hypervisor-based memory protection and run-time checks.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my PhD advisor Professor Peng Liu, for his continuous support and enormous guidance on my academic study. He has influenced and inspired me in so many ways, not only technically, but also philosophically. I have always admired his great personality: his dedication, enthusiasm and persistence.

I would also like to thank my committee member Sencun Zhu, Trent Jaeger, and David Miller for their helpful advice, feedback and discussion on my thesis work. Particularly, I learned a lot from attending Dr. Jaeger's lectures and I am amazed by his remarkable expertise on foundation and principles of operating system security.

In the summer of 2011, I did an internship at Microsoft Research Redmond. It was an extremely rewarding experience to remember. I would like to thank my mentors Jon Howell, Bryan Parno and John Douceur for hosting such a wonderful internship. I am really impressed and inspired by their extraordinary technical skills, profound knowledge and great hospitality.

I would like to thank my co-authors and colleagues, Shengzhi Zhang, Donghai Tian, Xiaoqi Jia, Jun Shao, Jun Wang and other labmates at Cyber Security Lab, for their collaboration and discussion on various research topics. I am also thankful to my friends at State College. Specially, I would like to thank Hengjing Yan, for making my life so colorful and enjoyable than ever.

Finally, I would like to express my deepest thanks to my parents, Zuoliang Xiong and Ye Liu, for bring me to this wonderful world, and for their endless love, support and constant encouragement in my entire life.

Dedication

This dissertation is dedicated to my mother Ye Liu and father Zuoliang Xiong, for their love and support throughout my life.

Chapter 1

Introduction

As the size and complexity of operating systems and applications are increasing, it is more and more difficult to achieve security in commodity computer systems. Among all the security principles, one of the most important principles is integrity, which means that the state and functions (e.g., code, data, control/data flows) of a system must be faithfully represented and carried out through its entire life cycle. There are many types of attacks on integrity that threaten users greatly. For example, (1) malwares (e.g., viruses, trojans, and worms) that compromise programs, data and configurations on victim's systems; (2) attacks that leverage code injection or data corruption to change the control and data flow of victim applications in undesired ways; (3) OS kernel rootkits and malicious device drivers manipulating metadata of OS objects to hide malicious activities.

Preserving integrity in a commodity system is a challenging task. A commodity computer system typically consists of several layers of subsystems with completely different semantics, and subsystems usually need to depend on (or trust) others in order to achieve correctness or security. For example, it is meaningless for an application to preserve the integrity of one of its file in case that the OS kernel, where all the file system metadata are stored, is already compromised and manipulated by an attacker. Hence, there is a chain of research questions to consider for these layers, for example: (1) How to secure the computer hardware from being tampered by an attacker? (2) How to protect the integrity of the OS kernel, especially in a commodity system where there are many untrusted programs running in the monolithic kernel environment? (3) How to design an OS-level mechanism

that enables system-wide integrity protection with the principle of least privilege, such that a compromised application would never hurt other part of the OS? (4) How to preserve the integrity of an application software in terms of not only static code and data, but also control transfers and data flows during execution?

These questions denote an important part of major research issues on protecting system integrity of a single host, yet there are even more related research issues in networking and distributed systems. Each one of these problems is a significant research topic, and achieving a secure system would require each of these questions to be addressed properly.

However, even if we have a "secure" system as described above, sometimes it still turns out to be that intrusions are inevitable in practice. Thus, in order to preserve integrity for computer systems, especially for critical infrastructures, we must consider the research problem in another important aspect: in case that a computer system has already been detected to have integrity compromise, how to recover system integrity from intrusions, and how to achieve this accurately, efficiently and without too much cost?

This thesis presents technical approaches to address a portion of problems described above. Our research has a strong emphasis on solving problems with commodity systems. However, it is worthy to mention that there are a number of research projects focusing on clean-slate designs of secure computer systems, which enforce their security principles from the scratch design. Examples of such projects in recent years include new secure hardware and architectures [1, 2], new secure operating system [3, 4], and new programming languages [5, 6]. While we applaud these research efforts, we also notice that it is infeasible to apply these approaches directly to commodity computer systems, primarily because of restrictions on practical deployment and adaptation of legacy code. Hence, it is also necessary to have approaches solving practical problem directly for commodity systems.

1.1 Overview of our approach

In this thesis, we describe security techniques designed and developed to achieve better integrity guarantee for commodity computer systems, in both proactive and

reactive manner. In specific, we mainly focus on two of the research problems stated previously: recovery from computer intrusions and integrity protection in OS kernel. The following statements summarize the thesis of my work:

- Reactive mechanisms can be developed to repair the integrity of a compromised computer system with high precision while minimizing the state loss and service downtime.
- By leveraging contemporary hardware and virtualization technologies, a hypervisor-based approach can effectively protect commodity OS kernels from untrusted kernel extensions in regards to both code, data and control transfer integrity.
- A set of security primitives can help programmers to achieve secure, fine-grained sharing and communication in commodity OS kernel without fundamental changes of the programming paradigm.

First, at the OS abstraction level, we explore reactive mechanism to repair a compromised computer system, which is the problem of intrusion recovery. We consider research questions such as: (1) How to minimize the loss of clean state while wiping out all the damages to the system? (2) How to reduce the downtime and maintain certain level of availability during the recovery procedure? To study such questions, we design and implement SHELF, an intrusion recovery system that aims to preserve business continuity, availability and recovery accuracy.

SHELF, as well as other OS-level security mechanisms, must rely on a trusted OS kernel not compromised by attackers. The security of the OS kernel is crucial not only because it is the piece of software with highest privilege, but also for the fact that other OS and application-level security mechanisms depend on it to function correctly. However, in practice, commodity OS kernels are becoming a favorable target for security attacks, and more and more vulnerabilities of kernel programs are being discovered and exploited. Hence, in order to keep reactive recovery mechanisms like SHELF correct and effective, it is desirable that the OS kernel could employ proactive protection mechanisms to preserve its integrity. We study the challenge of preserving integrity of commodity OS kernel - the Trusted Computing Base (TCB) for commodity computer systems.

In commodity systems, attacks against the OS kernel are often too stealthy to be detected. The cause of such vulnerability is mainly because of untrusted programs (e.g., third party device drivers) executing in the kernel space. In order to enhance the security of commodity OS kernel in the presence of untrusted code, we design and implement HUKO, a hypervisor-based integrity protection system that protects commodity OS kernels from untrusted extensions.

Given the strong isolation and mandatory protection provided by HUKO, we finally explore the question of how to design a set of security primitives so that OS kernel developers could benefit from it to build kernel programs that are inherently secure while enjoying the advantages of protection comprehensiveness, access control granularity and developer flexibility. Based on the hypervisor architecture provided by HUKO, we propose SILVER, a comprehensive framework that offers transparent protection domain primitives to achieve fine-grained access control and secure communication between programs in OS kernel. Compared to the mandatory protection provided by HUKO, SILVER allows principals in OS kernel with various trust relationship to exchange information, delegate privilege and export services in a more explicit, fine-grained, and controlled manner. Moreover, SIVER can also prevent attacks on kernel API integrity and confused deputy, which neither can be effectively handled by typical isolation systems.

In the following sections, we give an overview of each technical approach.

1.2 Preserving Business Continuity and Availability in an Intrusion Recovery System

The first problem we are focusing on is recovery from computer intrusions. The goal is to restore infected OS-level objects such as processes and files in a compromised system to clean state. Although the goal seems to be straightforward, even with a secured OS kernel, intrusion recovery is still a non-trivial job, especially for systems that run continuous services. Firstly, since today's intrusion detection system (IDS) is not fast and perfect, the time of detection of an intrusion symptom could be long time after the actually beginning of the intrusion. Given such a long time window for attack escalation, it is difficult to tell how the intrusion propagate

throughout the system and which part of the system is infected. For this reason, current intrusion recovery techniques often do not preserve the accumulated useful state of system objects (e.g., processes and files) during recovery, which results in great loss of useful effort and benign data.

Moreover, current intrusion recovery systems ([7, 8, 9, 10, 11]) generally require a dedicated recovery routine which largely increases the system downtime and decreases the availability level. This is also undesirable, especially for mission critical infrastructures such as servers and data centers. Business continuity and availability are crucial to them, and a small amount of state loss or downtime may cause great loss of money.

To address these shortcomings, in Chapter 3, we describe the design and implementation of SHELF, an on-the-fly intrusion recovery prototype system that provides a comprehensive solution to preserve business continuity, availability and recovery accuracy. SHELF preserves accumulated clean states for infected applications and files so that they can continue with the most recent pre-infection states after recovery. Moreover, SHELF leverages OS-aware taint tracking techniques to swiftly determine the sources of intrusion and assess system-wide damages caused by the intrusion. SHELF uses quarantine methods to prevent infection propagation so that uninfected and recovered objects can provide availability during the recovery phase. We integrate SHELF prototype in a virtualization environment to achieve user transparency and protection.

Chapter 3 also describes the evaluation of the SHELF prototype. We demonstrate SHELF's ability to dynamically assess damages after intrusion symptoms are detected and we also measure its run-time performance. Our evaluation shows that SHELF can perform accurate recovery on-the-fly effectively with an acceptable performance overhead.

1.3 Protection of Kernel Integrity for Commodity OS from Untrusted Extensions

As stated in Section 1.1, in order to have OS-level security mechanisms (e.g., intrusion recovery systems, anti-virus, host-based IDS...) function correctly, the

integrity of OS kernel must be protected to provide a solid security foundation. Hence, the next problem we are focusing on is to enhance the security of commodity OS kernel. In commodity operating systems, kernel-level extensions are widely used to extend the kernel's functionality. However, the extension interface is also the most prevalent source leveraged by attackers to tamper the integrity of the OS kernel. For example, attackers can install malicious extensions such as kernel rootkits to hide their activities in the system. These rootkits, once installed by the attacker, are often too stealthy to be detected. On the other hand, the existence of buggy third-party device drivers exposes many vulnerabilities which can be exploited by attackers to inject their malicious code into the kernel space. These untrusted extensions threaten the kernel integrity greatly, yet unfortunately in many cases users have to let them run in order to provide the desired functionalities and availability. Therefore, preserving the OS kernel integrity from the presence of untrusted extensions remains a challenging problem.

We develop HUKO, a hypervisor-based protection framework to secure the execution of untrusted kernel extension. HUKO allows users to execute untrusted extensions in the kernel space to provide desired functionalities. The behaviors of untrusted extensions, however, are confined by mandatory access control policies, which significantly limit the attacker's ability to compromise the integrity of OS kernel. The protection offered by HUKO covers multiple aspect of system integrity issues, which include code/data integrity, architectural state integrity, control flow integrity and stack integrity.

To guarantee such multi-aspect protection and enforcement, HUKO leverages contemporary hardware virtualization features to transparently *isolate* untrusted extensions from the OS kernel. Moreover, HUKO overcomes the challenge of mediation overhead by introducing a novel design named subject-aware protection state transition to eliminate unnecessary privilege transitions caused by mediating allowed accesses. Our approach is practical because it requires little change for either OS kernel or extensions, and it can inherently support multiple commodity operating systems and legacy extensions.

We describe the design, implementation and evaluation of HUKO in detail in Chapter 4.

1.4 Fine-grained and Transparent Protection Domain Primitives in Commodity OS Kernel

HUKO provides strong memory isolation and uses mandatory policies to confine activities of untrusted kernel extensions. However, even with strong isolation (e.g., memory protection, SFI) enforced, untrusted code in OS kernel could still subvert the integrity of OS kernel by abusing communication with OS kernel. For example, in commodity OSes like Linux, attackers could manipulate parameters passed to legitimated kernel API functions to launch confused deputy attacks. Hence, it is desirable to have a secure communication mechanism in commodity OS kernel. On the other hand, mandatory policies are limited in both granularity and flexibility for expressing access control rules that are close to program semantics. For kernel program developers, it is better to have security primitives that could express their own security concerns embedded in their programs, rather than having them set up by administrator and enforced externally.

Previous research efforts such as micro-kernel [12] and language-based operating systems [5] offers clean-slate model (e.g., multi-server IPC protocol and language contracts) to help developers ensure safe communication. However, these approaches are difficult to apply to commodity OSes, as they require developers to change both the development and the deployment paradigm of their software completely.

To address these challenges, Chapter 5 presents SILVER, a framework that offers transparent protection domain primitives to achieve fine-grained access control and secure communication between OS kernel and extensions. In SILVER, kernel program developers leverage SILVER's secure primitives to add light-weight annotations to their source code. These annotations indicates security properties (e.g., integrity levels and capability) of data objects and functions of the program. As a result, fine-grained access control policies and communication rules will be inferred and enforced by SILVER at run-time. To achieve this, SILVER provides clear resource management of kernel data objects according to their security properties. Based on this organization, it achieves access control enforcement and communication safety using hypervisor-based memory protection and run-time checks. Protection domains in SILVER are transparent, which allows developers to pre-

serve traditional programming paradigms (e.g., shared address space, function calls and reference passing) while obtaining desired protection. The primitives can be deployed incrementally and selectively, and protected programs are still compatible with unmodified kernel programs.

In Chapter 5, we describe the security model, design and implementation of the SILVER architecture. We also show how to apply SILVER to existing kernel programs for establishing protection and secure communication. Finally, we demonstrate SILVER's protection effectiveness by using security case studies of real-world threats to the Linux kernel.

1.5 Summary of Contributions

This dissertation makes the following contributions.

- A novel intrusion recovery approach that can comprehensively assess the damage to a compromised computer system and recover it to clean state with minimum loss of business continuity and availability.
- An run-time protection framework leveraging contemporary virtualization techniques for securing the execution of untrusted extensions and preserving the integrity of commodity OS kernel.
- A set of security primitives and OS enhancements designed for kernel program developers to achieve fine-grained access control and secure communication.

Chapter 2

Background and Related Work

This chapter presents the background and related researches in system integrity protection, access controls and integrity recovery. First, in section 2.1, we state our security goals in designing our approaches. Section 2.2 reviews various kinds of protection and access control systems, with a discussion of how HUKO and SILVER leverage these protection principles. In Section 2.3, we describe specific techniques that could be used to achieve such access control systems and enforce protection domains. We then present various categories of related work on operating system kernel security in Section 2.4. Finally we review background and research effort on intrusion recovery systems in Section 2.5.

2.1 What are the Security Goals?

In this section, we discuss what kind of security properties and requirements we would like to achieve in our systems. Defining clear security goals will provide principle-level guidelines for us to design and evaluate our systems.

Information Flow Integrity. Information flow model is probably the most famous classic model for describing security requirements such as secrecy and integrity. In information flow model, subjects and objects in the system are labeled into different categories, and information can potentially flow between subjects and objects via read and write operations. The model specifies constrains on where and how the information can flow. Denning [13] generalized the information flow

security problem using a lattice model, and Bell-LaPadula (BLP) [14] model and Biba [15] model provide specific requirements for secrecy and integrity protection, respectively. In this thesis, our major concern is integrity, and our major security goals are aligned with information flow integrity models [15, 16, 17].

Control Flow Integrity. In general, control flow integrity means that program execution must follow the same control-flow pattern [18] as intended by the programmer. For example, one requirement of CFI is that the only way to get into a function is through its entry point, and functions must return to the proper address of the caller site. These properties guarantee that control transfers in a program are not hijacked by attacks such as stack smashing, return-to-libc or return-oriented programming [19].

Principle of Least Privilege. The principle of least privilege [20] requires all security principals and components in a computer system can only have just enough information and resources to fulfill their tasks. Therefore, it limits the damage to the entire system in case that some principals or components are compromised by the attacker. In the context of OS kernel security, unfortunately most commodity OSes fail to achieve this principle since the core kernel and untrusted extensions are of the same exact privilege.

Minimize Trusted Computing Base. In today's commodity applications and operating systems, the growing complexity and size make these software vulnerable to varies kinds of attacks. Hence, it is desirable that we could reduce their trusted computing base (TCB), which is the portion of code that the software must trust and rely on in terms of security. For example, the 2.6.24 Linux kernel contains more than 6 million lines of code, making vulnerabilities inevitable. In this thesis, we reduce the TCB of commodity OSes by sandboxing and shepherding the execution of kernel extensions that are much more likely to contain bugs/vulnerabilities with more exposure to attacks.

2.2 What Kind of Protection to offer?

Having the security goals identified and established, naturally the next question would be: what kind of protection we need to offer to a computer system in order to achieve these security goals? Historically, the center of gravity of the solution is the access control mechanism, which is also the primary topic stated in this thesis. In this section, we review major types of access control systems used in secure operating system construction, and describe how HUKO and SILVER incorporate these different mechanisms.

Discretionary Access Control (DAC) [21] system is a type of access control system in which security principals could specify access permissions of their own objects. A typical example of DAC system is traditional UNIX file permissions. In UNIX, every file in the operating system has an owner, which denotes the associated security principal. The owner of a file is capable of controlling read, write, and execute privileges of the file with regards to security principals such as the owner, users in owner's group and other public users.

DAC grants users with full control of their resources. However, DAC system alone cannot effectively enforce security goals of secrecy and integrity. The primary reason is that, in DAC, the protection state is completely decided by individual users, which also indicates that untrusted principals could easily influence the protection policy and make undesirable changes to the protection state eventually. This motivates the creation of Mandatory Access Control (MAC) systems, in which security policies are decided by administrators or security experts. In MAC systems, subjects (e.g., users and processes) and objects (e.g., files, sockets and devices...) are labeled with their security attributes by the administrator. Whenever there is an access to an object issued by a subject, the action will be mediated by a *reference monitor* [22], which will examine security properties of the subject and object, and make authorization decisions according to access control policies defined by administrator.

The reference monitor is an important concept and an essential component in protection systems. In specific, it must hold several necessary properties to guarantee the enforcement of access controls, as stated in [23, 24]:

- Complete Mediation. All security-sensitive operations must be mediated by

the reference monitor.

- Tamperproof. The mediation and enforcement mechanism cannot be subverted or abused by untrusted principals.
- Verifiable. The protection system could be analyzed and verified for its correctness and completeness.

Implementations of MAC systems can be dated back to Honeywell's SCOMP [25], which offers operating system support to enforce multi-level security. In recent years, SELinux [26] and TrustedBSD [27] are two representative MAC systems designed to enforce MAC policies for commodity operating systems such as Linux and FreeBSD. These mechanisms are achieved by placing various authorization hooks (e.g., LSM [28]) into the OS kernel, which supports dynamic policy set up and configurations.

The protection system in HUKO is designed to be a MAC system: subjects and objects in HUKO are clearly identified and labeled. The reference monitor is implemented in the hypervisor layer isolated from the guest operating system, making it difficult for attackers to penetrate and tamper with. Security sensitive operations, including cross-domain data access and control flow transfers, are intercepted by the hypervisor because of hardware protection mechanisms such as page table permissions. Labeling can only be done by the administrator as well as the trusted helper component, and access control polices are represented by a fixed access matrix hard-coded in the program.

While mandatory access control systems are effective to enforce system-wide policies and protect system resources, it still has the following shortcomings:

Firstly, in MAC systems, security polices are completely decided by security expert and administrator intervention. Software developers and users are not allowed to specify or modify policies at the development or deployment phase. As a result, it is difficult for MAC systems to support flexible and fine-grained policies that are close to program semantic and security needs. For example, to support various kinds of applications with reasonable granularity and flexibility, SELinux policies in current commodity systems involves more than thousands of types and authorization/transition rules, which are often too complex to be configured properly.

Secondly, it is difficult for MAC systems to address ambient authority in an application communicating with other principals, which often makes the application vulnerable to attacks such as confused deputy.

To address these shortcomings, new protection systems are proposed to allow the delegation of part of security decisions to developers and users. For example, by binding permissions to references to individual data objects, capability systems [29, 30, 31] enable programmers to assign permissions to their processes and data objects more precisely. Singularity [5] allows programmers to specify language-based verifiable contract to secure the communication channel with other principals. Decentralized information flow control (DIFC) systems such as Asbestos [3], Histar [4] and Flume [32] allows users and developers to create security categories and labels for their own security concerns. The protection state is no longer centrally controlled only by the administrator, instead, it is partially decentralized to security principals. The enforcement of labeling, tainting and declassification/endorsement rules, however, is still controlled by the reference monitor.

Like other MAC systems, HUKO also has shortcomings in supporting fine-grained and flexible policies, and attackers could still exploit kernel APIs to launch confused deputy attacks despite of memory isolation. These shortcomings motivate the SILVER framework. SILVER is built on top of HUKO's mandatory protection mechanism such as isolation, labeling and hypervisor-level reference monitor. In addition, with its OS subsystem, SILVER allows OS kernel developers to specify security properties of their data objects and functions. Those security properties are maintained explicitly in SILVER's OS subsystems, and they directly impact resource allocation and security decisions.

2.3 What are Techniques for Enforcing Protection?

In this section, we look deeper into techniques that actually enforce isolation and access control.

OS-level Protection Techniques. Multics [33] is the first major effort for building an advanced operating system. It proposed and developed many fundamen-

tal and crucial concepts, which significantly influence the design of subsequent computer system in many years. In the realm of protection and security, these revolutionary concepts include but not limited to: hardware-supported hierarchical rings of protection, protection domains and gates, and OS support for access control lists (ACL) and multi-level security policies. Protection rings in Multics provide separation of privilege as well as fault tolerance/handling, and it inspired contemporary hardware-assisted virtualization leveraged by HUKO and SILVER.

System call interception [34, 35, 36] is another OS-level confinement and moderation techniques. By mediating activities at the system call level, the reference monitor makes authorizing decisions, allowing or denying system calls issued by processes. The solution is simple and easy to be deployed. However, some security sensitive activities (e.g., memory mapping, ipc) are difficult to mediate at the system call level, and some are difficult to get enough information to make security decisions. Moreover, the mechanism itself is easy to be bypassed or tampered with, making it demanding to have a more resilient and comprehensive protection.

To achieve completeness and flexibility, modern operating systems adopt a technique named hook placement: inside the OS kernel, for every security-sensitive operation, software *hooks* are placed along the execution paths. The coverage of placement ensures complete mediation property of the reference monitor, and the flexibility of software hooks enables dynamic loading and changing various security policies on the fly. The most notable example of this category is the LSM [28], which enables many advanced OS-level access control system [37, 26, 32] to be built atop.

As previously stated, OS-level reference monitors can effectively mediate security-sensitive activities at the operating system abstraction level, where subjects and objects are system resources such as files, processes and network connections. However, they are limited at the program abstraction level, in case that programmers need to establish protection and access control inside their own programs. Moreover, the same protection cannot be applied to protect the OS kernel, since the OS kernel does not operate on OS-abstractions (in fact, it holds the meta data that define such abstraction) and there is no way to place authorization hooks for direct memory accessing kernel objects. In the following, we review techniques that focus on enforcing access control and isolation at the program scope.

Memory Protection. In modern operating systems, virtual memory offers address space isolation for different processes, and this is usually achieved by hardware-based memory protection such as segmentation and paging. In virtualization, memory protection techniques is often used to isolated virtual machines from each other. Shadow page tables (SPT) is a common techniques for hypervisors to manage guest-to-machine mapping and enforce isolation, yet it has shortcomings of unnecessary VM exits during page table updates. To facilitate paging in virtualization, hardware-assisted paging (HAP) is proposed by AMD [38] and Intel [39] in recent years. HAP extends the original paging mechanism by adding another layer of page tables and translation, and it leverages dedicated hardware to do page table walks and compute page table entries. Due to these reasons, HAP is very desirable for enforcing isolation in a transparent manner.

While HAP is commonly used to establish isolation between multiple instances of guest virtual machines, HUKO and SILVER novelly adopted to enforce isolation and memory protection in the kernel address space of a single guest. Specifically, we create multiple HAP tables for each protection domain in OS kernel to enable memory isolation and we leverage HAP permissions to enforce integrity protection and mediate protection domain transfers. In this way, we ensure that the entire reference monitor mechanism is isolated and protected from the guest layer, and achieving this does not need to affect the OS-level paging mechanism as it is using another layer of indirection.

Although memory protection systems effectively take advantages of advanced hardware features, it still have shortcomings. The first shortcoming is granularity. In commodity operating systems on commodity hardware, the page size is usually 4KB or higher and permissions can only be set up per page basis. This is usually undesirable since there are various kinds of data objects residing on the same page in both program stack and heaps. Research efforts such as Mondrix [40, 1] could provide finer granularity

The second shortcoming is the "semantic gap". In specific, there is a semantic gap between the page layout and the programmer's view of application data. It is extremely unreasonable and errot-prone to require a programmer to align their data along page boundaries and setting up page permissions by themselves. Thus, how to effectively let programmers take advantages of the memory protection provided

by hardware remains a challenging problem.

We designed SILVER to address these challenges for protection in commodity OS kernel. The OS subsystem of SILVER handles the translation from security properties of program data to page permissions, and it leverages a novel organization and placement mechanism to achieve access control granularity. We further discuss the design and implementation of SILVER in 5.

Software Fault Isolation Besides hardware protection, there are also software solutions for enforcing isolation. A notable solution is software fault isolation (SFI) [41, 18, 42, 43, 44], which leverages software approaches such as binary rewriting and compilers to isolate a piece of untrusted code from the main program usually residing in the same address space. The restricted environment that contains untrusted code is usually called a "sandbox". The software approach does not rely on hardware protection mechanism to perform address space isolation, however, it still guarantees that any unsafe instruction would not have undesirable effect to outside of sandbox.

Compared to hardware protection, SFI systems have their advantages. First, SFI approaches have flexibility in deployment and development process. It is self-contained, and requires little support from specific hardware, virtual machines or operating systems. Second, although SFI introduced additional inline checks for memory access, it avoids costly protection domain switches in hardware protection mechanisms. According to previous experiments [42, 44], this could be a potential performance gain, especially in case that the sandboxed code interact frequently with the main program.

One downside of SFI is that malicious software module can subvert SFI's protection mechanism by abusing legitimate interfaces, for example, calling functions in wrong order or with undesired parameters. As shown in Chapter 5, this could lead to confused deputy attacks in OS kernel. LXFI [45] is a SFI system that addresses this problem by requiring developers to declare capability for its data objects and functions. Another downside is due to SFI's self-contained nature: the mechanism is agnostic to outside accesses. Thus, it is possible to launch TOCTTOU (time of check to time of use) attacks to exploit those inline access checks used in SFI systems, especially in a multi-threaded and concurrent usage environment.

Language-based Approaches Language-based approaches can help programmers enforce fine-grained access control in regard to program internal data and semantics. Typically these approaches integrate security notions into the type systems of the programming language, and/or use verification methods and compiler-inserted checks to enforce security policies and access control. Jif [46] and Joe-E [47] are Java language extensions that implements DIFC and capability security primitives, respectively. Laminar [48] is a Java language-based DIFC system with the OS support to handle OS abstractions. Singularity [5] is a research operating system which leverages language (C# extension) support and static verification to achieve isolation and controlled communication. The downside of language-based approaches is that they often require fundamental changes to the programming paradigm, and it generally takes significant effort to make legacy programs to adapt a new language.

2.4 Related Work on Operating System Kernel Security

The idea and design of HUKO and SILVER draw inspiration from a variety of topics of past research work related to OS kernel reliability, protection and security. In this section, we review these specific previous research efforts.

Kernel integrity protection. There are a number of previous research efforts aiming at protecting the integrity of the operating system kernel, such as code integrity protection [49, 50, 51], data integrity protection [52, 53] and control data/flow integrity protection [54, 55, 56]. Secvisor [49] is a hypervisor based protection system which guarantees the life-time code integrity of the kernel. It leverages advanced features from AMD processors, which are analogous to those used in HUKO. HUKO differs from Secvisor in the following aspects: Firstly, Secvisor is intended to prohibit any untrusted code executing in the kernel space, while HUKO does allow untrusted kernel extensions running securely to provide functionality and availability. Thus HUKO needs to enforce additional protection such as data integrity and control flow integrity to restrict the behavior of untrusted extensions. Secondly, Secvisor's tiny hypervisor design renders the system a very

small TCB, which grants the system a more secure foundation which is easier to be verified. In comparison, HUKO is based on Xen hypervisor with a larger TCB, yet it saves deployment and configuration effort for existing Xen virtual machines.

Kernel malware analysis. Several recent projects such as Panorama [57], K-Tracer [58], HookFinder [59], HookMap [60], and Poker [61] focus on analyzing the behavior of kernel-level malwares. These research work are complementary to HUKO protection system because they provide extensive knowledge of how malwares damage the integrity of the kernel. These knowledge would further help HUKO to enforce more effective access control policies on various kinds of kernel objects to offer comprehensive protection.

Device driver isolation. Another major category of related research work is on isolating buggy device drivers to improve the reliability of operating systems. Examples of these systems include Nooks [62], MINIX 3 [63], and SafeDrive [64]. Micro-kernel OSes [12, 63, 65, 66] removes device drivers from kernel space and execute them as userspace server applications. Opal [67] is a micro-kernel based system which supports multiple protection domains for the entire application universe within a single address space. However, despite their elegant design, it is generally difficult to retrofit these approaches in commodity OSes. Mondrix [68] is a hardware protection approach for compartmentalizing Linux and providing memory isolation for unsafe kernel extensions. Access control in Mondrix can achieve the granularity of memory words but it requires a specific designed processor architecture to support its protection mechanism. Such systems are mainly targeted for fault resistance and dependability, and they could effectively prevent system crashes caused by design defects and programming mistakes of device drivers.

Nooks [62] is a comprehensive protection layer that leverages hardware protection to isolate faulty device drivers within Linux kernel and recover them after failures.Our system resembles Nooks since both approaches establish hardware-enforced protection domains to isolate kernel components. However, by the time Nooks was designed, there was no supporting hardware features such like NX bits, EPT, VPID, IOMMU, etc. By leveraging these advanced features, HUKO significantly reduces the amount of OS modifications and has a better performance. Also, HUKO offers more protection from malicious extensions, e.g., it preserves

architectural state from being modified by untrusted extensions. Since its primary focus is fault resistance rather than security, Nooks does not address attacks such as manipulating architectural state. As a VMM-based approach, HUKO has a smaller TCB and attack surface compared with OS-based approaches. Also, Nooks does not provide the flexibility to specify security properties of individual data. Language-based approaches such like SafeDrive provide type enforcement and prevent memory errors, though they often require the source code of extension for recompilation, which limits their applicability for binary drivers. In contrast, HUKO can support unmodified legacy extensions.

Mandatory access control. HUKO enforces mandatory access control policies over subjects and objects in the OS kernel. There are many systems that are designed for improving operating system security by adding mandatory access control, e.g., LOMAC [17], SELinux [26], AppArmor [37], UMIP [69] and Loki [2]. These systems provide flexible, powerful and fine-grained protection to preserve system-level integrity. However, they are all enforcing MAC at the OS abstraction level and cannot be applied to mediate the activities of kernel-level objects.

Address space separation. As part of our design, HUKO isolates untrusted extensions from the OS kernel using the memory virtualization mechanism provided by VMMs. There are also a number of systems achieving different research goals using various techniques that isolate two entities which previously belong to the same address space. MMP [40, 1] achieves address space isolation and fine-grained permission mapping by extending the hardware architecture. XFI [42] provides permission management within system address spaces using binary rewriting. NativeClient [44] offers sandboxing and isolation to native x86 modules by leveraging x86 segmentation and code validation. SIM [70] proposes a secure In-VM monitoring approach which places the kernel-level monitor in a protected address space using shadow paging. Overshadow [71] and Bastion [72] leverages multiple shadow tables to protect application data from the rest of the system. In comparison, HUKO focuses on protecting the integrity of the OS kernel. Also HUKO is based on hardware-assisted paging rather than software-based shadow paging mechanism to reduce the number of VMEXITs and improve the TLB performance.

SILVER leverages a VMM as another layer of indirection to mediate cross-

protection-domain activities. VMMs are also widely used for protection systems to enhance the security of application and OS kernel. TrustVisor [73] protect the integrity and secrecy of an application even in case that the OS kernel is compromised. Secvisor [49] and NICKLE [50] are hypervisor-based systems which guarantee that any unauthorized code will not be executed in the operating system kernel. Hooksafe [54] protects kernel control data (i.e., hooks) from being tampered by kernel-level rootkits. In comparison, SILVER aims to provide a more comprehensive protection with the integrity guarantee of both code, data and control flows.

Protection domains. In practice, protection domains are widely used for addressing security problems such as securing program extensions [74], privilege separation [31], implementing secure browsers [75, 76], safely executing native code in a browser [74, 44] and mobile application deployment [77]. In this section, we review previous research efforts related to protection domains and OS kernel security, categorized by the approach to achieve their goals.

One major mechanism to achieve protection is through software fault isolation [41, 42, 43, 44], which rewrites binary code to restrict the control and data access of the target program. XFI [42] leverages SFI to enable a host program to safely execute extension modules in its address space by enforcing control flow integrity (CFI [18]) and data integrity requirements. While these approaches are efficient and effective for securing program extensions, they have difficulties for inferring and verifying system-wide resource and multi-principal access control rules in a static manner. Moreover, protection domains could also be achieved by language-based approaches. Singularity [5] is an experimental operating system that achieves strong isolation and controlled communication by advanced language features such as type checking and static verification without any hardware protection support. Compared with Singularity, SILVER achieves similar high-level goals for data object management by relying on a complete run-time approach, which emphasizes on compatibility with commodity OSes and avoids the complex effort of static resource verification.

LXFI [45] is probably the closest related work with SILVER. It addresses the problem of data integrity and API integrity in SFI systems, using a completely dif-

ferent approach (compiler rewriting) than SILVER. Compared to LXFI, SILVER's run-time approach is more resilient to attacks that fully compromise a untrusted module and execute arbitrary code. Moreover, security enforcement of SILVER is more tamper-proof since the isolation and access control are carried out by the hypervisor.

Run-time protection approaches are mostly achieved by access control mechanisms to constrain the behavior of untrusted programs. Depending on the abstraction and granularity levels, these approaches mediate security-sensitive abstractions ranging from segmentation [78, 79, 44] and paging protection [62], system calls interposition [80, 74] to high-level APIs such as JNI calls [81]. These events are regulated by a set of access control policies.

2.5 Previous Approaches for Repairing a Computer System

In this section, we review previous research approaches that are related to restoring the integrity of a compromised computer system.

There are two previous research works which are mostly related to the SHELF system: Taser [11] and RFS [10]. They both have the ability to track the information flow by analyzing the runtime log and to perform selective recovery afterwards. The primary difference between them and SHELF is that Taser and RFS are only focused on repairing the persistent data on a compromised system. They do not keep the useful process state during the recovery procedure. Moreover, they have dedicated recovery routines that do not preserve availability during recovery. In contrast, SHELF is an on-the-fly solution that coordinates both file and process recovery to achieve business continuity and availability.

Taser assumes an immutable file system state at the beginning of the recovery procedure. Usually it is achieved by rebooting the system into a dedicated recovery environment, which means all the running services are forced to shutdown and their useful states are wiped out. On the other hand, SHELF cares innocent processes and keeps positive influence of even infected process instead of restarting them.

RFS uses a backward recovery strategy. It achieves recovery by undoing the

contaminated operations so that the file system can rollback to a clean state. It has the advantage for avoiding the effort for periodical checkpointing. However, we argue that backward recovery is not suitable for recovering process states. While it is relatively easy to construct undo log for file systems, rolling back process state by performing undo operations is hard to achieve because the process state is changed by sequences of instructions, which are hard to record and reverse.

For the aspect of system construction, both Taser and RFS adopt a host-based architecture. Taser is implemented as a Linux kernel module and a backend system. RFS requires modification to the Linux kernel. In contrast, SHELF is a complete user space solution which does not require any privilege level or changes on host system. Moreover, SHELF is a VM-based system that can provide additional resistance to attacks than those host-base approaches.

There are other techniques that can be used for recovery purpose, like journaling file systems [82], process checkpointing and rollback ([83, 84]), process migration [85] and replayable systems ([86, 87]). Compared with SHELF, these systems have limited ability to assess damage automatically, which includes detangling bad operations from good ones and performing selective forward correction. Our recovery idea is also inspired by Liu's work on intrusion tolerant database systems that can continue its transaction processing even in the presence of active attacks [88].

Intrusion detection systems provide intrusion symptoms to SHELF as the starting point of the recovery procedure. There are various kinds of systems performing intrusion detection by different techniques. For example, signatures of system call sequence are studied by many researchers to detect intrusions ([89, 90]). For virtual machine based methods, Garfinkel [91] proposes an intrusion detection architecture through VM-based introspection. VMwatcher [92] is a system that detects malwares through VM-based OS-level semantic view reconstruction.

SHELF is focused on recovering user-space processes and persistent data. Nooks [62] is a complementary system to SHELF, whose aim is to protect and recover from kernel damage or failures. Also there are other techniques that can be applied to protect the integrity of the operating system kernel, such as VM-based active monitoring [93] and Panorama [94].

Information flow analysis techniques are widely used for intrusion detection

([89, 95]), malware analysis [94], and intrusion recovery systems ([11, 10]). SHELF uses a method similar to BackTracker [95] to perform automatically damage assessment. Nowadays, fine-grained dynamic analysis techniques are widely used for system security purposes ([96, 94]). Although fine-grained analysis can track damage more precisely and comprehensively, those schemes exact very heavy performance penalty so that they are not suitable for on-the-fly recovery solutions like SHELF.

Chapter 3

SHELF: Preserving Business Continuity and Availability in an Intrusion Recovery System

In this chapter, we focus on the problem of integrity recovery. Specifically, we study how to preserve business continuity, availability and recover accuracy in repairing a computer system that has already been compromised.

Nowadays, business continuity and availability are increasingly crucial to servers and data centers. In some business services like online banking and MMO game hosting, a small amount of loss of business continuity and availability may cause great loss of money. Intrusion recovery is an important security task that servers and data centers have to perform when the system is compromised by intruders. Unfortunately, current intrusion recovery systems do not bear the concerns of business continuity and availability in their design principles. For example, snapshot-based file system ([7, 97]), one-button-recovery feature of some laptop computers ([8, 9]) and Norton Ghost require rebooting the entire system or remounting file systems during the recovery procedure. Research efforts on automatic recovery ([10], [11], [98], [99], [95]) reduce human efforts in analyzing the impact of intrusions as well as providing higher accuracy in recovering infected files and removing malwares. However, they still have the following shortcomings in preserving business continuity and availability.

Firstly, current intrusion recovery techniques often do not preserve useful ap-

plication states when they perform recovery. The primary focus for many recovery systems ([10], [11], [98], [99]) is on recovering persistent data (e.g. files and registry) and removing malwares. In their methods, all the changes made by the intrusion, including applications that are infected, are detected and wiped out from the system. In such case, services provided by infected applications are discontinued and all the useful states of these services are completely lost even though these services are restarted after recovery. This could cause a great loss of business continuity in commercial systems.

Secondly, current intrusion recovery systems ([7, 8, 9, 10, 11]) generally require a dedicated recovery routine which largely increases the system downtime and decreases the availability level. Usually the recovery routine begins with one or several actions of the following: (1) restarting the whole system, (2) unmounting file systems, (3) closing outgoing connections, (4) shutting down applications and services. After that, the system dedicates to the recovery procedure, which takes a long period of time if there are lots of infected files. This results in a considerable system downtime and availability loss.

To address the above limitations, in this chapter, we describe the design and implementation of SHELF, an on-the-fly intrusion recovery prototype system that provides a comprehensive solution to preserve business continuity, availability and recovery accuracy. SHELF has three unique features: First, to maximize business continuity, SHELF preserves accumulated useful states for applications and data, even if they are infected by the intrusion. At the recovery phase, instead of restarting an infected object, SHELF enables it to continue from the most recent pre-infection state. Second, by synthesizing backward taint tracking and forward taint tracking techniques, SHELF can swiftly determine the system-wide damage caused by the intrusion when infection symptoms are detected. In other words, SHELF has good recovery accuracy that can dynamically distinguish changes made by the attacker from those made by legitimate users. Third, SHELF maintains a reasonable availability level for the system during the recovery phase. This means that uninfected objects remain intact and can continue functioning while infected objects are being recovered. To achieve this, SHELF leverages dynamic quarantine and de-quarantine techniques and adopts a transparent background recovery procedure.

We implement SHELF on top of a light-weight virtual machine, which is the User Mode Linux [100]. SHELF performs most of its functionalities at the VMM layer. Thus it provides a transparent environment which imposes minimal interference on the guest. SHELF also makes good use of security benefits provided by the virtual machine to achieve better isolation and protection than existing host-based methods.

In summary, this work makes three main contributions. First, we propose an on-the-fly intrusion recovery framework with the concern of business continuity, availability and recovery accuracy. Second, we design and implement SHELF, which is a prototype of our approach. SHELF comprehensively leverages techniques such as application-level state recording and rollback, backward/forward taint tracking and infection quarantine in a virtualization environment. Third, we evaluate SHELF's recovery performance using real world intrusions and benchmarks. Our results show that SHELF can perform accurate on-the-fly recovery effectively with an acceptable performance overhead.

3.1 Model and Assumptions

In this work, we assume a UNIX-like operating system in which most OS-level entities can be abstracted into two kinds of objects: processes and files. A *process object* is a running instance of an application. For a stateful application, its *state* is usually determined by the run-time state of the corresponding process object. A *file object* can be used for representing various kinds of OS-level entities such as regular files in the persistent storage, volatile memory mapping areas, sockets for managing network connections, pipes for doing IPC and many device interfaces. The operating system provides a uniform set of operations (such as the `read`, `write` system calls) towards file objects, making it convenient for us to track the state changes of files. We use the terminology "objects" throughout this chapter to refer processes and files in the system.

Normally intrusions begin from one or several objects in the system, then propagate to other objects via object interaction events. In our model, we name these objects which initiate the intrusion "*attack seeds*". In most cases, attack seeds are created by attackers and their behavior is malicious from the very beginning of the

intrusion. An attack seed can be a process, for instance, a malicious login shell created from backdoor or injected codes, a service program compromised by remote attacks or worms, or an illegal ssh session. Also, attack seeds can be files such as viruses, malwares, user-level rootkits, undesired install packages or corrupted data. As time goes by, attack seeds create attack objects and infect other innocent objects in the system via certain OS events (e.g. IPC, read/write or socket operations). In this way, the intrusion is propagated throughout the system.

Generally an intrusion recovery system requires an intrusion detection system (IDS) to discover the intrusion and initiate the recovery procedure. The IDS is responsible for detecting one or several objects that are compromised by the intrusion. In our model, we name these victim objects *intrusion symptoms*. Examples of intrusion symptoms can be applications that received malformed network packets, modified system binaries or data, and malware processes or files. Note that we do not assume the IDS is fast or perfect, nor we need it to provide us with the attack seeds. In fact, due to the limitations of current techniques, the detection of the intrusion often happens after attack escalation and is far from enough to assess the system-wide damage. SHELF does backward taint tracking to identify the source of the intrusion. We describe these issues in Section 3.3.2.

To systematically evaluate the impact of an intrusion, SHELF classifies all the objects in a compromised system into three categories:

malicious objects: Objects created directly or indirectly by the attacker. Malicious objects include attack seeds and objects that are created by malicious processes and infected processes.

infected objects: Objects that are originally good but are infected by interacting with malicious objects or infected objects in the system.

uninfected objects: Good objects in the system which are not yet infected by the intrusion.

3.2 Overview of Our Approach

As previously stated, the basic goal of SHELF is to preserve business continuity, availability and recovery accuracy in an automatic intrusion recovery system. To preserve business continuity, SHELF periodically records accumulated useful

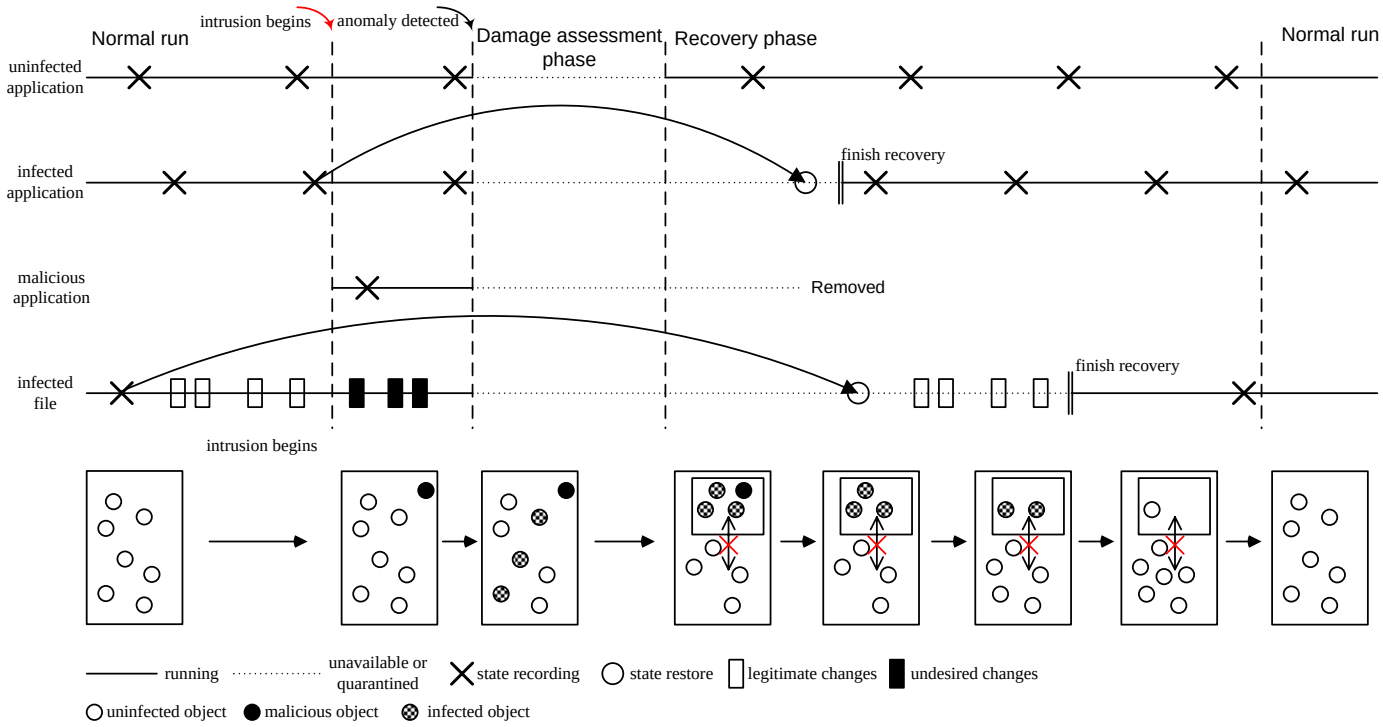

Figure 3.1. Overview of SHELF's workflow. The upper part of the figure shows that SHELF applies different policies for recovering different kinds of objects. The solid line means the object is functional and the dashed line means that the object is not available during that period. The lower part of the figure demonstrates a system-wide view of objects. The intrusion starts from a malicious object (attack seed), and then propagates to other objects. During the recovery phase, SHELF quarantines malicious and infected objects from uninfected objects to to prevent propagation of the infection (as indicated by the red-colored X symbols) . Objects are in turn healed and de-quarantined so that the system becomes clean.

states for each stateful running application in concern. Thus during recovery, a compromised application can be restored to the most recent clean state to avoid significant business continuity loss which in many situations can be caused by blindly restarting a stateful application in a stateless way. Regarding files that are infected by the intrusion, SHELF first restores them to a previously recorded clean version, then replays the state-changing operations towards them until their content reaches the state just before they are infected by intrusion. The recovery activities of infected files and application processes are carefully synchronized to eliminate inconsistency between repaired file and process states.

On the other hand, SHELF coordinates system-wide damage tracking and quarantine operations to precisely determine and quarantine malicious and infected objects in the system quickly after the intrusion symptoms are detected. In con-

trast, uninfected applications could keep running with caution during the recovery procedure and uninfected files can be properly accessed. In this way, the overall system availability is maintained to a desirable extent.

As Figure 3.1 shows, SHELF always operates on three phases: the normal run phase, the damage assessment phase, and the recovery phase. During the normal run phase, SHELF does the periodically state recording for each object in the system. Simultaneously, SHELF logs essential system-wide events to track the objects and their interactions at multiple levels. A taint analysis engine is responsible for dynamically analyzing and maintaining the dependencies among objects. When intrusion symptoms are detected by the IDS or an administrator, SHELF enters the damage assessment phase and it swiftly determines the malicious objects and infected objects from the maintained object dependencies. Based on the classification of objects and pre-defined policies, SHELF prepares scripts for performing quarantine and recovery on each object. After that, SHELF begins the recovery phase. During the recovery phase, all the uninfected objects remain their functionality while the infected and malicious objects are quarantined. Quarantined objects are deactivated so that they are incapable of infecting other objects. Furthermore, operations that access the quarantined objects are also regulated. Quarantined objects are then properly recovered according to the recovery policies, which we discuss in detail in Section 3.3.3. Once an object is healed, SHELF de-quarantines it so that it can interact with other uninfected objects in the system. The recovery phase is completed when all the quarantined objects are eliminated or recovered. After that the system enters the normal run phase again.

3.3 Design of SHELF

Figure 3.2 shows the framework of our system. In SHELF there are three separated layers in the system: the guest layer, the VMM layer and the host layer. SHELF's components are in the VMM layer and the host layer, which are isolated from the guest layer that is vulnerable to attacks. These components include: (1) the state recording and restore module, (2) the logging and reconstruction module, (3) the dynamic damage assessment engine, (4) the quarantine enforcer, and (5) the recovery engine. We describe each of the components below.

3.3.1 State Recording and Restore

The state recording and restore component is responsible for periodically recording states of objects in the normal run phase and rolling back infected objects to previously recorded clean states in the recovery phase. The basic technique used here is the checkpoint and rollback mechanism, which is not new. However, considering our specific on-the-fly recovery environment, our design should have properties such as flexibility, correctness and efficiency.

In order to provide flexibility for handling object-level recovery and to optimize storage consumption, SHELF allows asynchronous checkpoints when performing state recording, which means that we can record the states of different objects in the system at separate time points. Also we assign different checkpoint intervals to different applications and files.

The next concern is correctness. One major problem with asynchronous checkpoints is that inter-dependent processes and files often reach different states after recovery. This can cause incorrect results and cascaded rollback among those unsynchronized objects. For example, a file may suffer from a double-replay error - legitimate file operations are replayed, but they will be executed again by the rolled-back process.

To address this problem, for two processes that have inter-dependencies, SHELF records their states at the same time to eliminate inconsistencies. For process-file inconsistencies, we propose two solutions. The first is to shorten the file operation replay period of each related file until its state after replay is synchronized with the least recent recorded state of the related process. In other words, during recovery, a file is rolled-forward to the same time as the related process rather than the time just before it is infected. Hence the double-replay error is avoided. However, this method sacrifices a certain amount of useful state. The other solution uses deterministic replay [86] to advance the rollbacked process state to the last uninfected state. This method minimizes the useful state loss, but it needs to add new facilities for deterministic recording and replay. Currently we use the first method to solve the synchronization problem, and we plan to implement the second solution as our future work.

In the production environment where SHELF runs, we must require that SHELF's mechanism has good performance, in terms of runtime overhead and storage con-

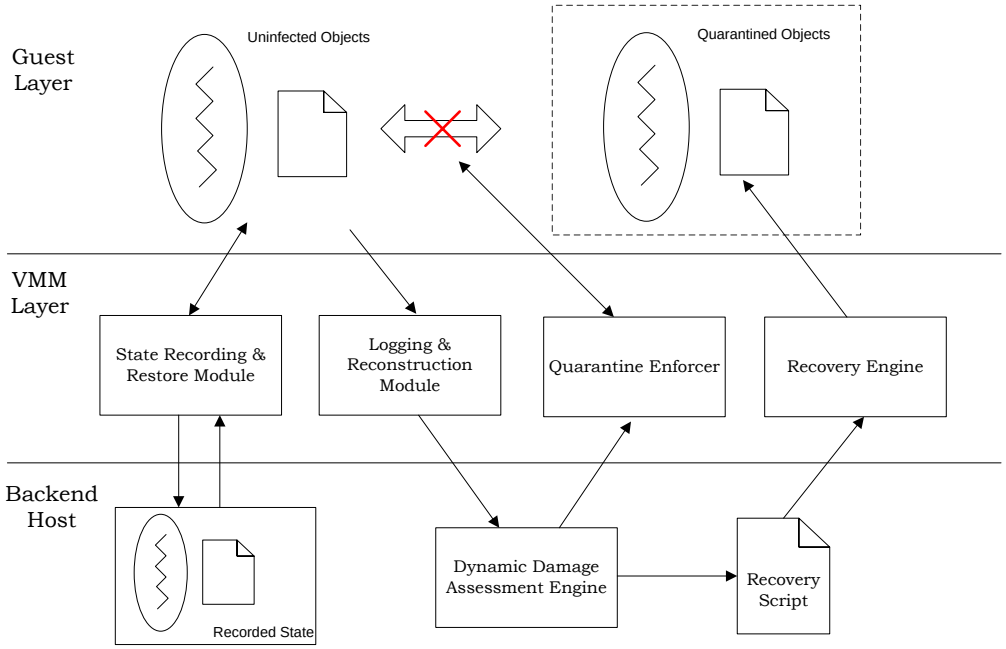

Figure 3.2. SHELF's architecture

sumption. In SHELF's design, when recording the state of a guest application, SHELF creates a backup process in the host layer to store a snapshot of the entire address space of the guest process. By preserving the recorded state in host memory, we not only avoid the cost of disk access, but also isolate the recorded state from the guest system for better security. The creation of the backup process is in a copy-on-write page sharing fashion, which is similar to the `fork` operation in UNIX systems. The difference with the `fork` is that standard `fork` operations do page sharing at a single abstraction level, while SHELF does page sharing across two abstraction levels: the virtual environment layer (i.e., the guest layer) and the host layer. Usually these two layers use different page tables and are isolated from each other. We describe the implementation of this *cross-layer* page sharing technique in Section 3.4.3. Adopting this technique, the time taken by performing a state recording operation towards a guest process is almost equivalent to a single `fork` operation on host. The time to rollback a process to a recorded state is almost instant. The memory consumption is also largely minimized by copy-on-write sharing mechanism.

3.3.2 Logging and Dynamic Damage Assessment

The dynamic damage assessment capability is achieved by tracking the inter-dependencies between objects in run-time. Through these dependencies, we know how the infection is initiated and how it propagates from one object to another. The facility for this functionality consists of two parts: the logging and reconstruction module implemented at the VMM layer, and the dynamic damage assessment engine located at the host layer.

Logging and Dependency Tracking The logging and reconstruction module logs the events that may cause object inter-dependencies during the normal run phase. For performance and availability reasons, we mainly monitor and analyze execution flows and data flows at the system call level. We adopt similar dependency rules as used in other works ([95, 11, 99]) which utilize dependency tracking techniques, and we list these rules in Table 3.1. SHELF uses a virtual machine to audit various kinds of system calls corresponding to the events that cause object inter-dependencies. SHELF also needs to reconstruct object and system call information from the VMM layer to identify an object in its lifetime and prepare replay or undo data for recovery. These information includes system call names and arguments, filenames/paths, inode numbers, process PIDs, write contents, and so forth. SHELF also records the timestamp for each system call entry.

Using these recorded object inter-dependencies, SHELF generates and maintains dependency graphs [95] during the normal run phase. In a dependency graph, each vertex represents an object in the system, while each graph edge represents the event that causes objects dependency. Each graph vertex is associated with an object ID and each graph edge is associated with a timestamp of the event. Since the graph may grow very large and produce false-positive results on taint propagation, SHELF performs graph pruning to reduce the storage size and false-dependencies. For example, we do not consider situations like independent process termination, irrelevant signals, or accessing dummy objects like `stdin`/`stdout` and `/dev/null`.

Backward and Forward Taint Tracking

The dynamic damage assessment engine implemented on the host layer does

Table 3.1. Dependency Rules

Dependency	**Events**	**System calls**
File → Process	Read or execute file object	read, readv, execve, socketcall(recv),etc.
Process → File	Create or write file object	write, writev, create, socketcall(send),unlink, etc.
Process → Process	Process creation, IPC, shared memory	fork, pipe, mmap, kill, etc.

backward and forward taint tracking to determine both malicious and infected objects in the system. It operates on two steps: first, upon receiving intrusion symptoms reported by the IDS, SHELF performs backward taint tracking starting from the intrusion symptoms to trace the chains of events that are on the intrusion path, and eventually identifies the source of the intrusion (attack seeds). In the second step, SHELF performs the forward taint tracking starting from the attack seeds to comprehensively assess the system-wide damage and classify objects for quarantine and recovery.

The reason to perform backward taint tracking is that the intrusion detection may often happen after attack escalation so that the intrusion symptoms reported by IDS are not necessarily the source of intrusion. In general, we use a method similar to the BackTracker [95] paper to determine the attack seeds. The technique mainly performs a backward search on the maintained dependency graph, and uses filtering rules to find the source of the intrusion. In SHELF's environment where attackers cannot get physical access to the system, we can refine the attack seeds to processes that have opening connections to the outside, for example, *httpd* sessions. Moreover, administrators can also specify the set of vulnerable services and vulnerable ports to further refine this procedure. Besides attack seeds, SHELF also records the intrusion timestamp, which is identified as the relative time when the attack seeds perform the first action which eventually propagates to the intrusion symptoms detected (e.g. executing injected codes).

Once the attack seeds are decided, the next step is the forward taint tracking. At first SHELF marks the attack seed as tainted, and assigns its infection timestamp to be the intrusion timestamp that is determined during the backward taint tracking step. Then SHELF performs a forward search on the filtered dependency

graph. For each dependency event that is represented by a graph edge, we do the following check: if there exists dependency A→B and A has been marked as tainted, we mark B as tainted only when A's infected timestamp is smaller than the timestamp of that dependency event. SHELF keeps doing this procedure until there is no more new tainted object in the system. At the same time, SHELF determines malicious objects and infected objects from the tainted object set according to the following rules: (1) The attack seeds are classified as malicious objects. (2) Objects that are created by tainted objects are classified as malicious objects. (3) Other remained tainted objects are infected objects.

3.3.3 Quarantine and Recovery

The motivation of the quarantine procedure is to prevent the malicious and infected objects from infecting other objects in the system during the recovery phase. Thus uninfected objects and objects that are repaired and de-quarantined can continue functioning to provide availability. The quarantine enforcer in SHELF is implemented at the VMM layer. During the quarantine procedure, it manipulates the operating system kernel to perform the quarantine task according to quarantine policies.

Table 3.2 shows the default policies and mechanisms that SHELF adopts in the quarantine procedure. For malicious objects, there is no need to recover them and they should be destroyed completely. For infected objects which are worthy to be recovered, we must ensure: (1) They should be suspended or deactivated because they are currently in corrupted states. (2) Operations that access the quarantined infected objects should be regulated to prevent the infection from propagating to other objects.

One important issue is that regulating the operations that access quarantined objects may affect the availability of the system. This happens when a quarantined object is accessed by an uninfected or de-quarantined process. If we let the process wait until the quarantined object is repaired and de-quarantined, the process would lose availability during the waiting period. To minimize this availability loss, besides the waiting policy, SHELF has an alternative deny policy for regulating the operations during the quarantine period. In specific, if SHELF detects a

system call that accesses a quarantined object, SHELF would intercept that system call from the VMM layer so that the system call would never reach the guest kernel, then SHELF modifies the system call return value to an error value which informs the guest process that this access attempt is denied. Though this may affect the correctness of some applications, in our observation, most service-oriented and user-oriented applications return a failed request message and can continue to deal with other requests correctly.

While quarantining infected objects in the system, SHELF recovers them. The recovery engine at the VMM layer, cooperating with the state recording and restore module, performs the recovery procedure. As mentioned in Section 3.3.1, the primary task for recovering an object is to restore the object state to the most recently before-infection-state. Once an object is recovered, SHELF de-quarantines it so that it can resume its pre-intrusion functions as if it was an uninfected object. Also, in order to assure correctness, dependent processes and files are clustered into groups: objects that are in the same group are simultaneously de-quarantined and rolled-back to the states with the same timestamp.

3.4 Implementation Issues

We implement SHELF prototype to demonstrate its capability to perform on-the-fly recovery. The VMM in SHELF's design is User Mode Linux (UML) [100], which is a light-weight VMM. Leveraging SHELF's techniques in UML poses a variety of challenges. In this section, we first briefly describe the key architecture of UML and the virtualization techniques that our prototype takes advantage of. Then we discuss our primary implementation issues through the rest of the section.

The total amount of code in our prototype is approximately 3900 lines, which include 2310 lines of code for the VMM layer components and 1600 lines of the host layer components. We did not make changes to the guest system.

3.4.1 User Mode Linux

User Mode Linux (UML) is a VMM that lets a guest Linux kernel run in the user mode. UML adopts a OS-on-OS structure, which ports the entire Linux system

Table 3.2. Quarantine Policies

Objects	**Policies**	**Mechanism**
Malicious Objects	Destroy objects	Process: send SIGKILL signal. File: remove the file.
Infected Objects	Suspend and disallow access	Process: remove from the scheduling queue (`run-list`), disallow reading the shared memory region. File: Nullify or block system calls that read, write or execute the file.
Uninfected Objects and De-quarantined Objects	Regulate access: wait Regulate access: deny	Block the violating system call until the object is de-quarantined. Nullify the violating system call and return a permission denied error.

from hardware interfaces to host OS services like system calls. We intensively studied the structure of UML running on the SKAS0 mode. From the viewpoint of the host, the UML kernel is a user process which has a complete separated address space from its guest processes. Each guest process has a corresponding host process that is traced by the UML kernel process. The UML kernel remotely manipulates address spaces and handles page fault for its guest processes. This is done by inserting stub pages that contain system call information and signal handlers into the address space of the corresponding host process, right above its stack. UML is also responsible for handling system calls issued by guest processes.

In SHELF, to establish communication channels between the host and the VMM layer, we modified the `uml_mconsole` protocol to send control commands and receive responses. In addition, we create files that are memory-mapped between both host layer components and the VMM to transfer non-control data.

3.4.2 Reconstruction and Monitoring

In order to track the inter-dependencies of objects in the system at the normal run phase, SHELF must perform the following actions: first, SHELF identifies each object in the system at run-time, which we call dynamic reconstruction; Second, SHELF monitors and records system events that cause inter-dependencies.

To accurately identify objects, we need to preserve OS-aware information (e.g. to reconstruct process descriptors, file paths and inode numbers) at run-time while doing monitoring. In virtual machine systems, usually there is a semantic gap [101] when we are trying to get OS-level semantics from the VMM layer. In UML's design, the VMM and the guest Linux kernel are within the same address space on host, making the effort to bridge this semantic gap easier. By adding some codes to the VMM, we are able to directly refer OS kernel data structures. We retrieve each process descriptor (`task_struct`) from the `all-task` double linked list, and the inode number of file objects from the open file table (`files_struct`). Moreover, SHELF keeps track of events that are related to the object creation and destruction. In this way, SHELF maintains a list of concerned objects in the system. For each object, SHELF associates it with a unique id (i.e., the hashed value of the object's PID or inode number) and two timestamps for recording the creation and destruction time of that object.

Besides the object list, SHELF also maintains a list of events that cause inter-dependencies of objects. To monitor and record these events, we modifies the VMM of UML by adding new functions to its system call interception facility. When a system call is issued by a guest process, the VMM which is tracing that guest process via `ptrace` will be notified by a special signal `SIGTRAP + 0x80`, and a trap handler will take control. From the trap handler, we intercept system calls and retrieve system call numbers and arguments from the register sets, which reside in the `thread` field of the process descriptor of that guest process. Furthermore, since the VMM and other guest processes are in separated address spaces on the host, we have to use `ptrace` with `PTRACE_PEEKDATA` option to retrieve system call information like file paths and writing content from the host address spaces.

3.4.3 State Recording and Rollback

In SHELF, regarding process state recording, the snapshot of a process state consists of the kernel part and the user part. The kernel part includes the process descriptor, `thread_info` structure, the kernel stack, pending signals and open file handlers, etc. Moreover, SHELF records some architecture-dependent data structures that are UML-specific such as `mmu_context`. SHELF reconstructs these data structures and stores them in a data structure named `thread_control_block`. SHELF keeps `thread_control_block`s in the host memory for isolation and good performance. During the recovery phase, each entry of the `thread_control_block` is copied back into the corresponding data structure of the guest kernel.

On the other hand, the user part of the snapshot, which is the entire address space of the process, is often very large and costly to record. As mentioned in 3.4.1, each running guest application in UML has a corresponding host process to provide the address space for it. To make a snapshot of a guest process, SHELF firstly creates a backup process on host which shares the address space with the host process that the guest application corresponds to. The method is similar to the `fork`, but it is done by the VMM and uses UML's trampoline code so that the backup process has stub pages and is ptraced by the VMM. The backup process is invisible to the guest kernel and is forced to sleep upon its creation. In order to restore the address space of an infected guest application during recovery, SHELF changes the `context->mm_id` of the infected guest process to the PID of the backup process on host. Thus the guest kernel will recognize the backup process as the new address space provider. Then SHELF invokes the `switch_mm` routine to modify the guest page table entries for establishing new mappings. After reinstating the `thread_control_block`, SHELF activates the backup process and sends a SIGKILL signal to destroy the disengaged host process which corresponds to the infected address space. By adopting this cross-layer page sharing mechanism, the state recording time of a guest application is largely reduced (1ms - 10ms for most applications), and the state restore operation is almost instant.

To achieve background recovery, SHELF does not replay the file state changing events at the VMM layer. Alternatively, it invokes a user mode helper process to do that job. Thus the recovery routine can be scheduled along with other uninfected and repaired processes in the system.

Table 3.3. Damage Assessment Statistics

	Backward Taint tracking		**Forward Taint tracking**				
Intrusion	Infected Symptoms	Attack Seed	Malicious Files	Malicious Processes	Infected Files	Infected Processes	Events logged
Malware install	Modified system binary (`netstat`)	remoted login (`sshd`)	14	15	21	4	453K
Internet worm	Malware script (`1i0n.sh`)	`bind` program that has TSIG vulnerability	85	27	20	6	1604K

3.5 Evaluation

In this section, we describe the experimental evaluation of our SHELF prototype. We have two goals for our evaluation. First, we want to test SHELF's ability to dynamically assess damages after intrusion symptoms are detected. Second, we want to measure the performance of SHELF. This includes the efficiency of state recording, the run-time overhead during the normal run phase, and the availability level of the damage assessment phase and the recovery phase.

Our experiments are conducted on a machine with a 2.13 GHz Intel Core Duo 2 CPU with 2 GB memory. The host operating system is Fedora 7 with Linux kernel version 2.6.22.9. The version of User Mode Linux kernel is 2.6.24.2 (SKAS0) and the guest system is Ubuntu Hardy Heron. The guest memory is configured to be 1024M.

3.5.1 Damage Assessment

We evaluate SHELF's damage assessment capability by launching two real attacks against honey-pot systems that are protected by SHELF. Then we measure the results of SHELF's damage assessment. We describe each scenario below.

Malware install The attacker logs into the system by `ssh` using an unprivileged user account. Then she launches the `sendmail` local escalation exploit to gain root access. The attacker uses that root shell to download the ARK rootkit, which replaces system binaries with backdoored versions. These binaries include `syslogd`, `login`, `sshd`, `ls`, `ps`, `netstat`, etc. The attacker uses a modified version of `netstat` to hide all the connections for her own uid. The IDS of the system

detects the modification of system binaries by the integrity check, then it notifies SHELF to begin the damage assessment phase.

Internet worm Our victim machine which runs the `bind` service is attacked by the `lion` internet worm via exploiting a buffer overflow vulnerability of the service. The worm runs several shell scripts to do the following in turn: adding itself to startup, deleting related system logs, rewriting several system programs to trojans, scanning and attacking vulnerable hosts, reading the password files (`/etc/password` and `/etc/shadow`) and sending them out through the `mail` program, downloading code from a remote site and finally leaving an open root shell open. The IDS detects a malware script (`/dev/.lib/1i0n.sh`) as an intrusion symptom.

Table 3.3 shows the results of SHELF's damage assessment. Starting from an intrusion symptom, SHELF successfully performed backward taint tracking to distinguish the attack seed, which is the login session of that malicious user and the hacked `bind` service respectively. After that, SHELF performed forward taint tracking to identify different kinds of objects in the system. The statistics are also shown in Table 3.3.

3.5.2 Performance

Runtime overhead. During the normal run phase, SHELF records dependency-making events for every process it monitors. Also it does reconstruction to provide OS-aware information for object identification and future analysis. Furthermore, a taint propagation graph is maintained for keeping track of inter-dependencies of objects. These three operations make up the system running overhead which degrades the overall performance of the system. (In our measurement, the process state recording usually takes less than 10ms so that it makes up little portion of the runtime overhead.) In most cases, the auditing overhead is not constant and it largely depends on applications that we trace. In general, SHELF poses a larger overhead to I/O-intensive applications than computational-intensive applications. We focus on the situation when the system is heavily-loaded with file system operations. We run the following workloads as benchmarks to evaluate the run-

time overhead: (1) Extracting a Linux 2.6.24.2 source tarball (.tar.bz format). (2) Building a Linux 2.6.24.2 UML kernel from source. (3) `dbench 3.0`: A filesystem performance benchmark which measures the throughput of the system as a file server (single client, 300s). (4) `Apache ab` test, which benchmarks the system performance as a HTTP server by measuring the average response time and the transfer rate. We set up 200 concurrent clients, with each client generating 2000 requests to obtain a 95KB file from a web server that runs in SHELF system. We set the process state recording interval to be 2 seconds. We conduct these benchmarks 10 times separately in three environments: host machine, UML and our SHELF system. In workload (5) we do not test the performance of native Linux since the UML uses TUN/TAP device for virtual networking so that the networking performance is not comparable to the native host which has direct network access. Table 3.4 shows the results of our conducted experiments. We can see that the overhead introduced by SHELF is from 1.076x to 1.65x.

Storage consumption. In order to record system call results as well as file state modifications (e.g. user space content for `write` system call) for future selective event replay, SHELF requires relatively large disk space for storing the logged events. Table 3.5 shows the raw and compressed event log file sizes for the two workloads we described above. We can see that the compression of the log data can effectively reduce the storage consumption of SHELF. One advantage of SHELF over host-based approaches is that the event log is stored at the host file system so that it does not consume the disk space that allocated to the guest system.

Availability. From the point that intrusion symptoms are detected to the time when the three sets of objects are determined, SHELF must suspend all the suspicious processes in the system to prevent taint propagation during the damage assessment phase. Fortunately, this system down time is very short in most cases. Essentially, SHELF first searches the object dependency graph backwardly to decide the attack seeds, then searches the object dependency graph forwardly to determine malicious and infected objects. The time complexity can be expressed as $O(\mid E \mid + \mid V \mid)$ since each vertex and each edge will be explored twice in the worst case. To further reduce the damage assessment cost, SHELF dynamically

maintains and prunes the object dependency graph and forces filtering rules for finding the attack seeds. In our measurement, for attacks that we evaluated which involve less than 500 objects, the damage assessment phase is 0.17 seconds at most. Moreover, we simulated a random object dependency graph which has 1000 objects and 100000 dependencies (this is too dense for real cases), the damage assessment time is 1.18 seconds. We believe that optimization of the analysis algorithm and adding more filtering rules can further reduce the system downtime effectively.

Regarding applications that are functional during the recovery phase, the availability level is still less than the normal run phase since SHELF is doing background recovery jobs which hurt the system performance. We measure the throughput of the `dbench` and the `Apache ab` while the system is performing intensive recovery jobs. The results are 32.92MB/s for the `dbench` and 17333Kb/s for the `Apache ab`. Compared to the throughput under the normal run phase, the losses are 21.4% and 12.4% respectively.

3.5.3 Discussion and Limitation

Porting SHELF to other virtual machines. SHELF's functionalities can be ported to other virtualization environments, as long as the virtual machine monitor satisfies the following requirements: (1) It is convenient to intercept and record system calls at the VMM layer. (2) It is convenient to access the guest process address space and the guest file system with the help of the VMM. (3) It is possible to reconstruct user-level objects and kernel-level data structures at the VMM layer. To our knowledge, current open source virtual machines such as Xen and KVM satisfy these requirements so that SHELF can be ported to these virtualization environments without much effort.

Detecting and subverting SHELF. To achieve the on-the-fly intrusion recovery in a more attack-resilient and elegant way, SHELF adopts a virtualization-based system architecture. Although it is generally difficult to penetrate a virtual machine, studies show the possibility that some virtual machines can be detected and eventually subverted by exploiting their design defects and software bugs. More

Benchmark	**Native Kernel**	**UML**	**SHELF**	**SHELF's addon overhead**
Kernel Decompression	19.52s	31.89s	52.46s	1.65x
Kernel Build	217s	379s	410s	1.08x
Dbench throughput	294.18 MB/s	63.36 MB/s	41.91 MB/s	1.52x
Apache ab response time	N/A	44.73ms	48.11ms	1.076x
Apache ab transfer rate	N/A	21290 Kb/s	19797 Kb/s	1.075x

Table 3.4. Runetime Overhead of SHELF

Benchmark	**Events Logged**	**Log Size (Raw data)**	**Log Size (Compressed)**
Kernel Decompression	133308	541.0MB	82.7MB
Kernel Build	1344712	445.1MB	89.1MB

Table 3.5. Storage consumption

specifically, regarding the User Mode Linux, we can detect the existence of VMM by issuing special instructions to query stats from the processor or by examining kernel debug information. Moreover, a proof-of-concept code is proposed to crash the UML kernel [102]. We believe that these problems can be fixed by improving the design and software quality of the VMM.

3.6 Summary

Preserving business continuity and availability in an automatic intrusion recovery system is a highly-desired but very challenging goal to achieve. In this chapter, we proposed SHELF, an VM-based on-the-fly intrusion recovery prototype system that provides a comprehensive solution to preserve business continuity, availability and recovery accuracy. One unique feature of SHELF is that it can do coordinated file/process state recording, damage tracking, quarantine and recovery without sacrificing too much availability. Our evaluation showed that SHELF can perform accurate recovery on-the-fly effectively with an acceptable performance overhead.

We believe that a system such as SHELF can provide accurate recovery results and effectively reduce human efforts in computational environments that have requirements of business continuity and availability.

Chapter 4

Protection of Kernel Integrity for Commodity OS from Untrusted Extensions

Kernel-level extensions are widely supported in commodity operating systems to extend the kernel's functionality. However, the extension interface could also be leveraged by attackers to tamper the integrity of the OS kernel. For example, attackers can install malicious extensions such as kernel rootkits to hide their activities in the system. On the other hand, the existence of buggy third-party device drivers exposes many vulnerabilities which can be exploited by attackers to inject their malicious code into the kernel space. These untrusted extensions threaten the kernel integrity greatly, yet unfortunately in many cases users have to let them run in order to provide the desired functionalities and availability. Therefore, preserving the OS kernel integrity from the presence of untrusted extensions remains a challenging problem.

Previous research efforts on protecting the OS kernel primarily target at one aspect of kernel integrity protection, such as code integrity [49, 50], data integrity [52, 53] and control flow/data integrity [54, 55, 56]. While these approaches are effective against certain categories of attacks, the lack of multi-aspect protection renders the system's incapability to deal with multiple types of malicious activities. For example, systems that only guarantee the integrity of kernel code and hooks are vulnerable to DKOM (Direct Kernel Object Manipulation) attacks. Similarly,

protecting kernel code and data is not enough for defeating new control flow attacks such as return-oriented rootkits [19, 103]. Moreover, current approaches are also limited in countering advanced attacks such as direct kernel stack manipulation in commodity systems, in which the attacker manipulates control and/or non-control data in the kernel stack shared by all code entities in the OS kernel.

Another difficulty is about making the protection scheme practical and generic. Several proposals [49, 50, 51] preserve kernel code integrity by preventing untrusted code from executing in the kernel space to defeat code injection and malwares. However, they also eliminate all the benign functionalities and availability provided by untrusted extensions. Quite a few security approaches [55, 52, 61, 104, 105] utilize the knowledge of kernel data structures to achieve fine-grained auditing and intrusion detection. However, these approaches are dependent upon data structure semantics of a specific kernel, making them difficult to adapt different OS kernels with another version or from other venders. Moreover, the performance overhead induced by dynamically reconstructing and tracking fine-grained kernel objects makes these approaches not that suitable for an online protection system.

To achieve tamperproof and transparency in a system that protects the OS kernel, a common approach is to leverage the virtual machine monitor (VMM), which provides another layer of indirection. In such systems, to protect a security sensitive-kernel object, the VMM intercepts all the events that access this object and validates each event based on the protection policy. This approach is effective for protecting a small number of crucial objects in the kernel. However, severe performance problem arises once the quantity of protected objects becomes large, say, the entire kernel code and data area. The reason is that, no matter how VMMs are trapping these events (e.g., via instruction instrumentation or page protection), performing mediation for each event will always cause control transfers between the VMM and the guest, which will need multiple time-consuming privilege transitions (e.g., ring faults or VMEXITs). Researchers have proposed techniques such as *hook indirection* [54] to mitigate the performance problems for hook protection. However, this approach is only useful for protecting objects that are scattered across page boundaries, yet still cannot be applied to the entire kernel code and data.

This chapter presents HUKO, a hypervisor-based integrity protection system

designed to protect commodity operating system kernels from untrusted extensions. HUKO allows users to execute untrusted extensions in the kernel space to provide desired functionalities. The behaviors of untrusted extensions, however, are confined by mandatory access control policies, which significantly limit the attacker's ability to compromise the integrity of the kernel. In order to achieve multi-aspect protection, HUKO leverages hardware assisted paging to *transparently isolate* untrusted extensions from the OS kernel so that it could mediate all interactions (including memory modification, control transfers and DMA) between extensions and the kernel. Regarding kernel stack integrity, HUKO's approach includes a VMM-level *private stack* with lazy synchronization to offer a transparent and efficient stack separation and permission management for unmodified OS kernels. To address the challenge of mediation performance, HUKO introduces a design named *subject-aware protection state transition* to eliminate unnecessary privilege transitions caused by mediating benign accesses. HUKO is a *practical* approach because it requires little change for either OS kernel or extensions. Also it does not depend on semantic knowledge of kernel data structures so that it can inherently support multiple commodity operating systems and legacy extensions.

We have implemented HUKO prototype based on the open source Xen hypervisor. To facilitate HUKO's design, we leverage contemporary hardware virtualization techniques such as Intel's EPT, VPID and VT-d[1] [39, 106]. We evaluated HUKO's protection effectiveness by running malicious kernel extensions in both Linux and Windwos. Our experiments show that HUKO can protect the kernel integrity in the presence of various kinds of malicious extensions, including DKOM and return-oriented rootkits. In terms of mediation performance, the evaluation results show that the average performance overhead in application level benchmarks is ranged from less than 1% to 21%. Even for extreme cases when HUKO isolates the entire `ext3` file system (the largest module in our Linux OS) from the kernel, the mediation overhead for extracting a Linux kernel tarball is about 21%, with the protection state transfer rate at 390,000 per second.

We believe that HUKO provides a generic and transparent framework for running untrusted code in OS kernel with enhanced integrity protection for commodity systems. Also, this framework could be used to enforce mandatory access control

[1]AMD also has similar techniques with different names.

policies inside commodity OS kernels with an acceptable impact on performance.

The remainder of this chapter is organized as follows. We first describe the threat model, the integrity properties that HUKO enforces and our assumptions in Section 4.1. Section 4.2 provides an overview of the design of HUKO. Section 4.3 details the design and implementation of the entire architecture. Our evaluation experiments for both the protection effectiveness and performance of HUKO are shown in Section 4.4. We discuss limitations and future work of our system in Section 4.5. Finally, Section 4.6 concludes.

4.1 Kernel Integrity Threat Model

In this work, we focus on attacks that the adversary utilizes the kernel extension interface to compromise the kernel integrity, which is the most common method to attack a commodity OS kernel. To specifically illustrate the threats, we present three different attack scenarios as follows: (1) The attacker gains the root privilege of the entire system, then he loads malicious extensions such as kernel-level rootkits into the OS kernel. (2) The attacker exploits a vulnerability existed in a benign kernel extension (e.g., a buggy device driver) to inject malicious code and therefore changes the extension's behavior. (3) A careless normal user loads an unverified kernel extension (e.g., a third-party device driver), which contains malicious code. There are various ways in which these malicious code could damage the control flow integrity and data integrity of the kernel, for example, direct modification of kernel code, modifying control data (e.g., system call table, IDT and function pointers), modifying non-control data (e.g., process descriptors and file system metadata), writing to the kernel space via malicious DMA requests, and stack manipulation (e.g., return-oriented attacks).

We classify subjects in an operating system kernel into three categories. The first category is the OS kernel, which HUKO aims to protect. The second category consists of trusted kernel extensions, which are kernel extensions trusted by the system administrator. Generally their code need to be attested and verified to guarantee security. The third category is untrusted extensions, which are extensions that may be compromised or inherently malicious. Rootkits and unverified device drivers belong to this category.

HUKO protects the integrity of the OS kernel by enforcing the following properties in a mandatory protection system:

- ***Kernel code/data integrity***: code, static data and dynamic data of the OS kernel are protected from being modified by untrusted extensions via direct memory access or DMA access.

- ***Architectural state integrity***: architectural environment describing the execution state of the OS kernel such as segment registers, control registers and certain flag registers cannot be altered by untrusted extensions.

- ***Control flow integrity***: (1) control transfers from untrusted extensions to the OS kernel, including function calls, jumps and preemptions, are restricted to a set of kernel service functions named *trusted entry points* (TEPs) specified by the OS provider or the administrator; (2) function call consistencies such as call-return consistency are strictly enforced.

- ***Stack integrity***: (1) malicious code cannot be injected into stack frames belonging to the OS kernel; (2) For an untrusted extension, manipulating control data (i.e., function pointers, return addresses) in its own stack frames cannot subvert control flow integrity stated above; (3) non-control data (i.e., saved registers, parameters and variables) and control data in stack frames owned by OS kernel or other extensions cannot be corrupted by an untrusted extension.

For practical and usability reasons, the default mandatory access control policy of HUKO does not prohibit the OS kernel from reading information from untrusted extensions, which is different from classic integrity models such as Biba. However, if there is a need to satisfy this strict integrity requirement, the flexible mediation and enforcement mechanism in HUKO can still support system administrators to write policies with appropriate exceptions to enforce the "*no read down*" property.

HUKO is designed to be an added-on layer which provides an enhanced integrity protection for various operating system kernels with an affordable performance cost. As a design principle, HUKO relies on as little semantics of any specific kernel

as possible. On the other side, HUKO is not the elixir for every kernel security threats. For example, HUKO is limited in verifying the correctness of function parameters and general data passed between the OS kernel and extensions, which could open certain avenues that impact kernel integrity in indirect ways. Also our system does not prevent the untrusted extension from abusing the privilege granted by the OS kernel in current stage. We discuss these limitations and possible solutions in Section 4.5.

This work is focused on protecting the integrity of OS kernels. Other security issues, such as attacks on secrecy (e.g., information leakage) and availability (e.g., interrupt flooding, abuse of resource) of OS kernels are not in the scope of this work. Also, this work concentrates on dealing with threats from the kernel extension interface, and we assume that the hardware is trusted for the OS kernel. Regarding attacks to the kernel directly from the userspace, HUKO prevents untrusted kernel extensions from executing user-level content and prohibits user programs to write kernel memory. Previous work such as Secvisor [49] provides in-depth research on protecting the OS kernel from userspace intrusions using a hypervisor, and we believe that its method can be effectively integrated with HUKO to achieve a more comprehensive protection. At last, in HUKO system, the hypervisor is the trusted computing base which we assume its integrity is preserved.

4.2 HUKO Overview

4.2.1 Design Principles

The following paragraphs describe three major principles which motivated our research and guided our design process of the HUKO system.

- ***Multi-aspect Protection.*** The architecture must guarantee that the kernel integrity properties stated in Section 4.1 are enforced with mandatory protection. Security-sensitive operations that involve interactions between untrusted extensions and the OS kernel, including memory reference, DMA, control transfers and stack modification, must be mediated and validated upon mandatory integrity policies.

- ***Performance.*** The architecture must not have high performance impact due to mediation, object reconstruction/tracking or enforcing protection.

- ***Ease-of-Adoption.*** The architecture should support multiple commodity operating systems and any unmodified legacy kernel extension. The architecture should not change the semantics of either OS or the extensions. Also, the architecture should be a layered approach which requires little deployment efforts.

4.2.2 Design Overview

HUKO provides a transparent protection environment for commodity OS kernels in which untrusted kernel extensions can run with an enhanced protection. In HUKO system, we name all the kernel objects that are supposed to be protected by our mechanism *security-sensitive objects*. These objects are labeled and tracked by the labeling component in HUKO's hypervisor. Depending on the various purposes of deploying HUKO integrity protection, security-sensitive objects can be labeled as 1) the entire kernel code and data region, or 2) a given set of kernel objects that may be tampered by attackers to achieve specific goals, for example, hiding a malicious process by manipulating hooks and process descriptors. To guarantee multi-aspect protection and generality, in our design, by default we label and track the entire kernel code and data region as security-sensitive objects.

The following paragraphs abstractly explain various challenges we faced in designing the system as well as key features of HUKO.

Mediation Overhead. Regarding how to achieve the mandatory access control mechanism, an intuitive way is to intercept every access to security-sensitive objects, then to validate whether the access is permitted by the policy or not. This approach is straightforward and convenient for out-of-boxed monitoring, however, it is not practical because the mediation overhead is considerable even if the number of objects to be monitored is relatively small. We observed that many security-sensitive objects in the kernel are highly frequently accessed by operating system kernel itself. For example, in Linux, `task_struct` is a typical security-sensitive

data object because it can be manipulated by rootkits to perform process hiding and privilege escalation. On the other hand, `task_struct` is also a crucial accounting and scheduling data structure which would be modified several times by the scheduler during each context switch. Posing mediation on these legal accesses through an external reference monitor (i.e., VMM) causes enormous amount of unnecessary privilege transitions (e.g., page faults, ring faults and `VMEXIT`), which result in serious impact on performance.

To overcome this limitation, HUKO adopts a design named *subject-aware state transition* which divides the system workflow into multiple protection states. The behavior of the protection mechanism is determined by the current protection state, which is further determined by precisely distinguishing the type of current subject in the guest system context. Specifically, if the current subject is an untrusted extension, HUKO does complete mediation on all accesses to security-sensitive objects in order to protect the kernel integrity. By contrast, in the case when the OS kernel is executing, HUKO poses minimal interposition on object accesses. It only needs to audit control transfer events that cause a protection state transition. In this way, the total number of privilege transitions caused by mediation is significantly reduced, which grants HUKO much better mediation performance. Table 4.1 illustrates an example of different protection behaviors that are associated with different protection states. From it we could see that the number of events that lead to privilege transitions (presented in grey cells) is minimized due to the subject-aware state transition mechanism in HUKO.

Figure 4.1 is the state diagram which shows the various protection states of HUKO system as well as the state transition events. Currently HUKO has four protection states, which correspond to the OS kernel, trusted extensions, untrusted extensions, and the user space, respectively. The state transition events include inter-subject function calls, various types of jump, interrupt handling, preemptions, system calls and associated returns from these routines. Mediating these events is essential to guarantee comprehensive control flow integrity, which we further discuss in Section 4.3.5. Tracking the state transition is mainly achieved by the isolation mechanism in HUKO, which we describe in Section 4.3.3.

Transparent Isolation. As we stated above, HUKO should have the ability

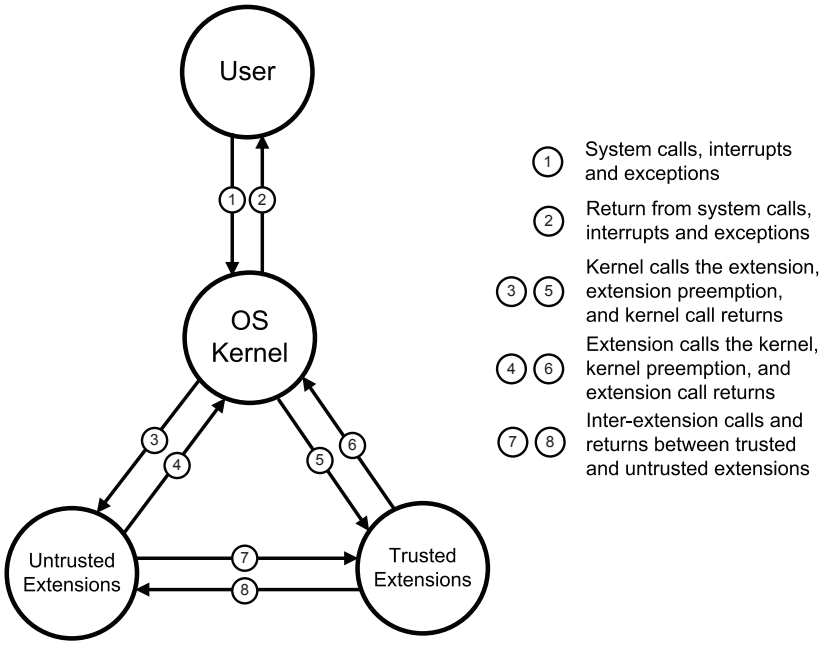

Figure 4.1. The protection state transition diagram.

to (1) distinguish the current subject in the guest context, (2) track all state transition events, (3) support different access control policies for different subject categories, and (4) mediate data modification flows and control flows between subject categories. Achieving these is non-trivial for commodity monolithic-kernel operating systems (e.g., Linux and Windows) since the OS kernel and its extensions reside within the same address space, and it is even more challenging especially considering our two design principles: external approach and good performance.

To tackle this challenge, we design an *isolation component* in HUKO's VMM to transparently isolate the extensions from the OS kernel. The isolation mechanism leverages hardware-assisted paging (HAP), which is a hardware-based virtualization technique supported by many modern processors. In our scheme, the enhanced memory virtualization component in HUKO's VMM maintains separate sets of HAP tables for each protection state in the system. These sets of HAP tables are synchronized with each other so that their corresponding entries are mapped to the same machine frame. Moreover, regarding security-sensitive objects, different HAP tables are reflecting different access rights according to the subject category and mandatory access control policies. Switching between these HAP tables is swift because it only involves a change to the HAP base pointer.

In addition, HUKO significantly reduces the number of TLB flushes involved in each HAP table switch by utilizing Intel's Virtual-Processor Identifiers (VPIDs) technology. The multiple HAP table design renders efficient and practical isolation between the OS kernel and extensions, and it enforces separate access control policies for each type of subject accessing various kernel objects such as dynamic data structures, I/O buffers and kernel functions. Regarding kernel stack integrity, HUKO leverages the multiple HAP tables to achieve a VMM-level *private stack* with lazy synchronization mechanism to offer a transparent and efficient stack separation, which we discuss in Section 4.3.4.

Object Labeling. In mandatory protection systems, objects are labeled indicating their security properties to facilitate mediation. HUKO does *object labeling* in order to let the VMM identify security sensitive objects in the kernel. The labeling procedure is at the page granularity in the way that the labeling component assigns labels to the specific physical pages that contain security sensitive objects. There are two reasons for this. First, according to our design principles, HUKO is intended to rely on as little semantic knowledge of operating system as possible. Second, for a hypervisor-based approach, fine-grained dynamic object tracking in kernel often introduces too much reconstruction and tracking overhead, which is not practical for an online protection system. On the other hand, to ameliorate problems caused by the protection granularity gap, HUKO has mixed page labeling mechanism for handling pages that contain mixed code and data, as well as pages that are shared by both kernel and extensions.

Another issue is about how to track dynamic data for both kernel and extensions. To address this, HUKO inserts a trusted driver (labeled as a trusted extension) into the operating system to notify the hypervisor about the allocation and reclamation of the kernel memory. The driver is also aware of the owner subject of each page and reports updates to the hypervisor during runtime. We further discuss mixed page handling and dynamic content tracking in Section 4.3.2.

Protection Workflow. Table 4.1 shows a sample protection policy that regulates the data accesses as well as code executions of untrusted extensions. In this policy, the policy maker needs to specify a set of kernel functions as the trusted

Object Label	Subject Category / Protection State								
	OS Kernel			Trusted Extensions			Untrusted Extensions		
	Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
Trusted Entry Points	allow	allow	allow	allow	allow	audit allow	allow	deny	audit allow
Other OS Code	allow	allow	allow	allow	allow	audit allow	allow	deny	deny
OS Data	allow	allow	allow	allow	allow	audit allow	allow	deny	deny
Trusted Extension	allow	allow	audit allow	allow	allow	allow	allow	deny	deny
Untrusted Extension	allow	allow	audit allow	allow	allow	audit allow	allow	allow	allow
Private Stack Frames	allow	allow	deny	allow	allow	deny	allow	allow	deny
Other Stack Frames	allow	allow	deny	allow	allow	deny	allow	deny	deny
Trusted DMA	allow	allow	allow	allow	allow	audit allow	allow	deny	deny
Shared DMA	allow	allow	allow	allow	allow	allow	allow	allow	allow
User Space Content	allow	allow	audit allow	allow	allow	audit allow	allow	allow	deny

Table 4.1. A sample MAC policy for preventing extensions from writing to kernel or executing unauthorized kernel code. The shaded cells indicate the corresponding events are mediated by the VMM and involve privilege transitions. Other events do not cause privilege transitions in HUKO. The write operation includes both normal write and DMA write. The not-listed "user" protection state is simply configured to deny any write to the kernel space.

entry points. In practice, trusted entry points can be exported functions in the kernel symbol table or picked specifically by the system administrator. To preserve control flow integrity, besides kernel function calls, kernel preemption and return instructions should also be considered, which we will discuss in Section 4.3.5. In addition, this policy also prevents untrusted extensions from directly writing to the OS kernel or any trusted extensions, no matter the write is performed via memory instructions or DMA transfers.

HUKO enforces mandatory access control over the entire life period of any untrusted extension. To achieve this, HUKO tracks the lifetime of an extension by hooking the extension allocation, loading and unlinking routine of the kernel. These events will be trapped to the hypervisor and the labeling component will manipulate the corresponding page labels to perform dynamic tracking. Unless specified by the administrator, HUKO labels all newly loaded extensions as untrusted. During the protection process, if any event that violates the access control policy happens, HUKO will trigger a protection alarm from the hypervisor and provide essential information (e.g., type of policy violation and the execution context) to the system administrator for making proper security decisions.

4.3 Architecture Design and Implementation

Figure 4.2 provides the overview of the HUKO Architecture. There are four major components corresponding to principle functionalities in HUKO's design: object labeling, transparent isolation, stack integrity protection, as well as mediation and enforcement. In the following subsections we first provide a brief background on Hardware-Assisted Paging (HAP) technology used in our prototype. Then we discuss each major component in detail. In Section 4.3.6, we briefly describe the implementation of HUKO prototype on the Xen hypervisor.

4.3.1 Hardware-Assisted Paging Overview

To achieve memory virtualization, a common design for VMMs is to load shadow page tables (SPT) into the hardware MMU, which translate from guest linear addresses (GLA) to machine-physical addresses (MPA). However, to maintain this indirect mapping, the hypervisor must intercept and do SPT synchronization upon guest CR3 switches and each update of the guest page table (GPT). The hardware-assisted paging (HAP) technology is introduced to avoid the software overhead incurred under shadow paging. One implementation of HAP is Intel's Extended page tables (EPT) technology [39]. When this feature is turned on, the ordinary IA-32 page tables (referenced by control register CR3) translate from GLA to guest-physical addresses (GPA). In addition, the hardware MMU maintains a separate set of page tables (the EPT tables) which translate from guest-physical addresses (GPA) to the machine-physical addresses (MPA) that are used to access machine memory. As a result, guest OS can be allowed to modify its own IA-32 page tables and directly handle page faults. This allows a VMM to avoid the VMEXITs associated with shadow paging, which are a major source of virtualization overhead.

The reason why HUKO is built atop hardware assisted paging rather than the software-based shadow paging mechanism is two fold. The first reason is for better performance, which we just stated. Secondly, in SPT, access rights in SPT entries are synchronized with the corresponding GPT entries. Hence, changing the access rights in SPT entries for our protection purpose may potentially affect the correctness of guest OS for handling its own access rights. By contrast, in HAP, access rights in HAP entries and access rights in GPT entries are two completely

different sets. Moreover, the HAP violation handling is transparently separated from the page fault handling mechanism of the guest OS, which makes it more flexible and easier to guarantee correctness.

4.3.2 Object Labeling

As shown in Table 4.1, in order to enforce the MAC policy, HUKO assigns various kinds of security labels to different kernel objects. The object labeling component is responsible for identifying kernel objects from the physical memory and managing security properties of these objects. As stated in Section 4.2.2, based on our design principles, HUKO directly associates object labels to the corresponding HAP entries. In specific, the labeling component makes use of a set of reserved bits in EPT entries. These reserved bits are never utilized by default so that changing these bits does not affect the hypervisor's functionalities. By encoding labels using these bits, HUKO currently can support 32 different potential object labels, providing flexibility and extendability to the protection scheme. This mechanism also reduces the time and memory space involved in every mediation and authorization action.

Handling Mixed Pages. In a commodity operating system kernel such as Linux, memory regions for kernel code, kernel data and extensions are usually page aligned, which facilitates the labeling procedure in HUKO. However, there are still existences of mixed pages in which different objects co-exist together. To ensure comprehensiveness and correctness of the protection, the labeling component must be able to track objects within two categories of mixed pages: (1) pages containing both kernel code and kernel data, and (2) pages containing both the kernel and extensions.

A major type of mixed pages in the kernel is large sized page (e.g., 2MB superpage). In most cases, different objects reside in the same superpage, yet their boundaries are still aligned to the 4KB address regions. Based on this observation, given a large mixed page, HUKO *splits* the corresponding EPT superpage entry into multiple subpage entries (e.g., 2MB page entry to 512 4KB sub-entries) and assigns individual object labels to each subpage. Splitting EPT superpage entries

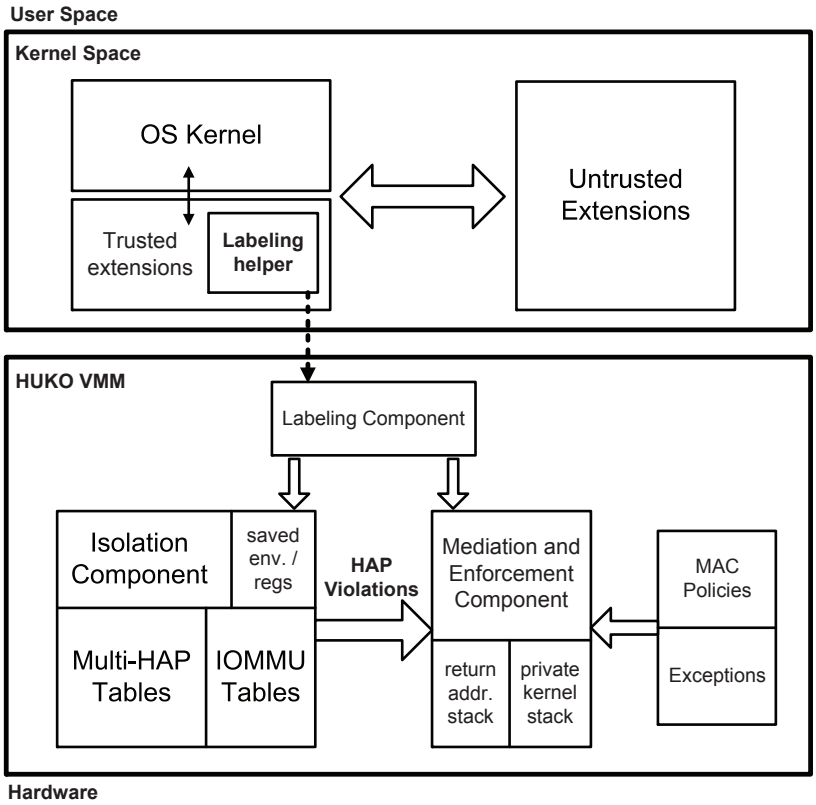

Figure 4.2. Overview of the HUKO Architecture.

improves the granularity of labeling and eliminates a majority of mixed page problems without changing the guest page table (GPT) entries. On the other hand, regarding mixed pages of 4KB size, HUKO assigns each of them with a mixed label. For example, considering a mixed page that has a mixed label of both kernel data and extension code, the hypervisor would trap all events that modify this page regardless of the current protection state. Then HUKO examines the physical address to see if it is in the range of extension text area and finally determines the object identity.

Tracking Dynamic Contents. Associating kernel objects to HAP page frames requires dynamically tracking of these objects. For static objects such as kernel code, static kernel data (including global variables), and trusted entry points, HUKO tracks them by leveraging the kernel symbol table (e.g., `Systemmap` file in Linux). On the other hand, for dynamic contents such as dynamic kernel data,

stack and heap region, and loadable extensions, it is difficult and time consuming to track them at the hypervisor layer because of the semantic gap. HUKO tackles this problem by loading a trusted extension named *labeling helper* into the guest kernel. The labeling helper is responsible for letting the hypervisor be aware of the allocation and deallocation of kernel dynamic pages as well as the owner subject of each kernel page. This component is the only OS-dependent part in our system and we implemented a prototype in Linux. Specifically, dynamic data owned by an extension come from two major sources in Linux: (1) the page frame allocator for allocating bulk of pages, and (2) the SLAB allocator for allocating fixed sized of registered cache objects. For both cases, the labeling helper hooks the allocation and deallocation events and gathers information from the SLAB allocator (i.e., `kmem_cache_alloc`), the free page allocator, and the `load_module` routine. This information includes owner subject of the page (e.g., OS kernel or extension), the content type (e.g., kernel data or extension code), the guest page frame number, the virtual address range (for handling mixed pages), and the timestamp of each event. Then the labeling helper passes these information to HUKO via the hypercall interface, and the labeling component labels the corresponding EPT entries accordingly. To guarantee tamperproof, the labeling helper itself is labeled as a trusted extension at the load time so that it is protected by HUKO. Furthermore, HUKO prohibits read accesses to the labeling helper to prevent the leakage of protection information.

4.3.3 Isolation Component

The isolation component in HUKO is responsible for achieving complete mediation by establishing separate address spaces for different categories of subjects (i.e., the OS kernel, trusted extensions and untrusted extensions) to reside in. Subjects can freely access code and data in their own address spaces without interposition from the hypervisor. However, inter-address-space activities such as data writing and control transfer must be mediated and controlled by the VMM.

Multi-HAP Construction. The isolation component is built upon our enhanced memory virtualization mechanism named *multi-HAP*. Multi-HAP enables exten-

sions and the kernel to share the same virtual-to-physical mapping of the entire kernel space, while it also enables the hypervisor to set different object access rights for different subject categories. In this scheme, the hypervisor maintains separate sets of HAP tables for each protection state (refer to Figure 4.1) in the system.[2] Figure 4.3 illustrates the architecture of the multi-HAP mechanism. For simplicity, only two sets of HAP tables are shown here, corresponding to the OS kernel state and the untrusted extension state, respectively. There is a HAP base pointer which points to the root level of a HAP table. During a protection state switch, HUKO changes the value of the HAP base pointer to another HAP table root, which represents another set of access rights. The access rights in HAP table entries are determined by the object label of the entry as well as the access control policy, and are updated when any object label changes.

To intercept control transfer events between different subject categories, for each protection state, HUKO manipulates the execution bit of its HAP table entries so that all the pages that do not belong to the subject category (corresponding to the protection state) are not executable. Attempts to execute content on these pages would cause HAP violations and are handled by the hypervisor. Section 4.3.5 describes this procedure in detail.

Synchronization. An important difference between multi-HAP and user-level page tables managed by the kernel is that, each HAP table in multi-HAP must maintain the entire mapping of the whole kernel space, rather than the address space associated with the protection state. This is because HUKO should allow the OS kernel and extensions to read each other's address space freely without any interposition. Therefore, the isolation component should always synchronize the entire kernel address mappings among HAP tables. We modify the hypervisor code so that changes to one HAP table (including allocating a new entry, changing an entry and removing an entry) always propagate to other HAP tables.

Optimize TLB Flushes. Considering the enormous function calls and returns between the OS kernel and extensions, the protection state transition rate in HUKO is very high (see Section 4.4.3). If the hypervisor flushes TLB on every

[2]It can be extended to support separate HAP tables for each subject, if needed.

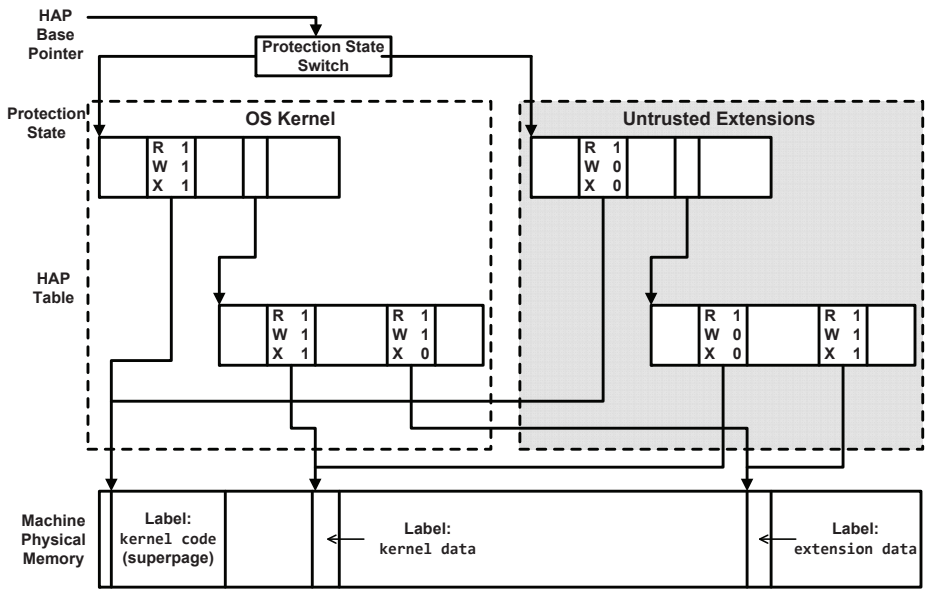

Figure 4.3. The multiple HAP tables for achieving isolation and mediation.

page table switch during a state transition, the performance degradation due to the TLB misses caused by flushing is substantial. To mitigate this problem, HUKO takes advantage of Intel's Virtual-processor identifiers (VPIDs) technology, which enables a logical processor of the hypervisor to manage cache information for multiple linear-address spaces. In HUKO's VMM, we associate each protection state with a 16-bit VPID so that mappings and access rights are tagged according to the VPID in the address translating cache. During the state transition time, the EPT table switch does not cause flush of the entire translating cache - it only flushes entries with specific VPIDs, which significantly reduces the TLB misses and improves the performance.

Preserving Architectural State. Sometimes malicious or compromised extensions could subvert certain invariants of the architectural state to fulfil their attacks. For example, a malicious extension could change the `GS` segment selector to point to its own version of processor data area (`pda`), which provides the kernel with incorrect information about the kernel stack, MMU state and IRQ processing. Therefore, HUKO must enforcing the integrity of system environment by preserving these invariants of architectural state.

Our approach takes advantage of the fact that, during a privilege transition,

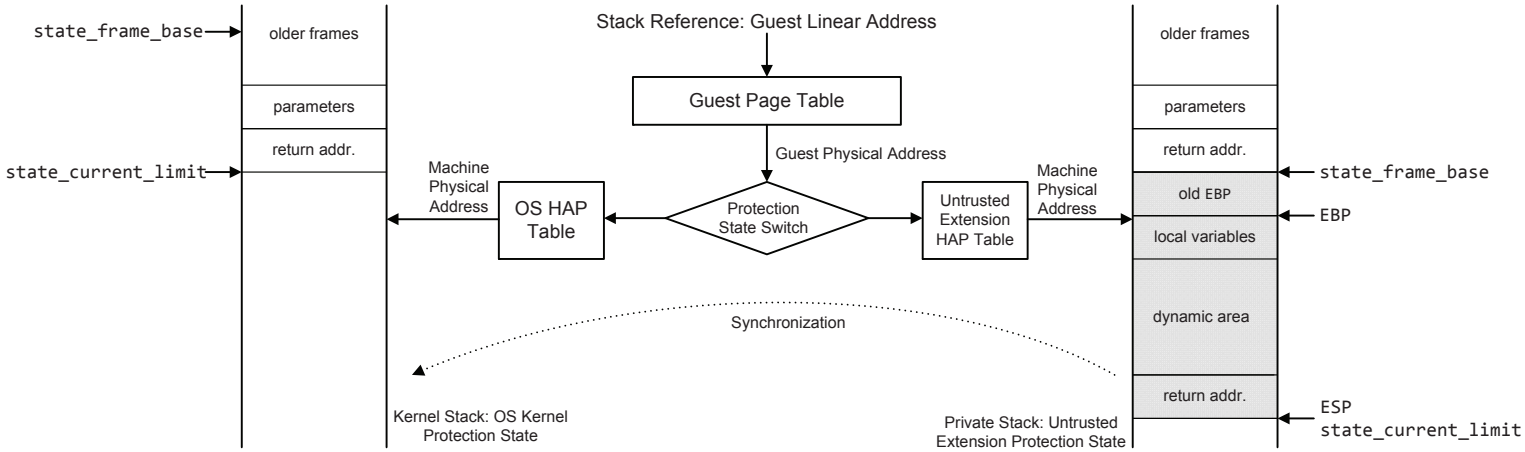

Figure 4.4. The transparent separated stack design supported by multi-HAP. The figure illustrates the two stacks at the time of protection state transfer in case an untrusted extension is making a call to the OS kernel. The shaded indicates the active stack frames (owned by the untrusted extension) which are going to propagate to the OS kernel stack.

the architectural state is saved in the virtual machine descriptor (i.e., `VMCS` for Intel VT) and a virtual CPU struct (i.e., `vcpu` for Xen) of the VMM for future reloading. Hence we could straightforwardly integrate the architectural state protection with our subject-aware protection state design. In specific, at the time when the kernel enters untrusted extension protection state, HUKO saves the architectural state from the `VMCS` and `vcpu` to its own memory space. When the kernel is switching from untrusted extension state back to the OS kernel state, HUKO restores all the architectural state invariants by writing the saved values to the virtual machine descriptor and the virtual CPU struct.

4.3.4 Kernel Stack Integrity

Besides code, static and heap data, there is another important avenue which malicious extensions could exploit to subvert OS kernel integrity: the kernel stack. In specific, adversaries could perform the following actions to compromise the property of stack integrity stated in Section 4.1: (1) injecting malicious code into the stack; (2) manipulating control data (i.e., function pointers, return addresses) in its own stack frames to subvert control flow integrity of the OS kernel. For instance, return-oriented and jump-oriented attacks belong to this category; (3) corrupting

non-control and control data (i.e., saved registers, parameters and variables) in stack frames owned by OS kernel or other extensions. For example, a malicious extension could change the local variables and function parameters on the stack frame to let a certain kernel function return a false data value, or it may manipulate kernel IRQ and exception stack frames to change the behavior that OS kernel handles interrupts and exceptions.

For case (1), by setting the NX bit of corresponding HAP entries of kernel stack frames, HUKO ensures that code on kernel stack frames could never be executed. Regarding case (2), HUKO mediates the protection state transfers and maintains a dedicated return address stack to guarantee the control flow integrity, which we will describe in Section 4.3.5. To defend against attacks in category (3), HUKO grants untrusted extensions read permission to the entire kernel stack, but only gives them write permission to its own stack frames.

To efficiently manage kernel stack permissions in an unmodified commodity OS (e.g, Linux) is a non-trivial job, because of the following reasons: first, in such system, there is only one kernel stack for all kernel control paths associated with each user thread. Moreover, the stack frames are not page-aligned, making it difficult to set permissions for individual stack frames using current architecture. On the other hand, in terms of performance, it is not affordable to validate each stack modification made by untrusted extensions because stack modifications are too frequent.

The stack protection design of HUKO overcomes the above limitations. In order to preserves single kernel stack semantic and support unmodified commodity OSes, during the protection state of untrusted extensions, HUKO creates and maintains a private copy of the current kernel stack at the VMM layer, which is transparent and not observable from the guest OS. By manipulating GPA to MPA mappings in the Multi-HAP table, HUKO casts the same linear address range of the kernel stack to different machine frames for OS kernel and untrusted extensions. In this way, an untrusted extension is given a "faked" view that it shares the same kernel stack with other code entities in the kernel, however, its stack operations are automatically redirected to the private kernel stack copy placed on shadow machine frames reserved by HUKO. On the other hand, to protect stack integrity in an efficient manner, HUKO adopts a "lazy synchronization" design: instead

of checking permissions each time the stack is accessed, HUKO only performs stack synchronization when current protection state is switching between untrusted extensions and the OS kernel. During synchronization, HUKO propagates stack modifications from the private stack to the real kernel stack with the following rule enforced: only changes made to its own stack frames are propagated to the real kernel stack, while updates outside its own stack frames are discarded.

In the following we use Linux as an example to illustrate the private stack design achieved by multi-HAP tables, which is shown in Figure 4.4. In Linux, each user process is associated with a two-page sized kernel stack. The scope of the current kernel stack can be determined by the `ESP` register and the per-CPU data structure pointed by the `GS` segment selector. HUKO maintains two data values for each protection state: `state_frame_base` and `state_current_limit`, respectively. These two values designate the active stack frames associated with each protection state, and only in these stack frames modifications are propagated to the other stack. During each protection state transfer, HUKO updates `state_frame_base` and `state_current_limit` based on the values of `EBP` and `ESP` registers at that time point.

4.3.5 Mediation and Enforcement

The goal of the mediation and enforcement component is to audit all the write flow and control transfer events between untrusted extensions and the kernel. Also it is responsible for validating these events to enforce integrity protection according to mandatory access control policies.

EPT Violation Handling. HUKO relies on the EPT violation mechanism to achieve mediation and protection enforcement. Figure 4.5 depicts the work flow of how HUKO handles various kinds of EPT violations. When an EPT violation occurs, HUKO first checks if the physical frame is labeled as a valid kernel object. If yes, then it checks if the violation is caused by our protection mechanism or by emulated MMIO and log-dirty events. An EPT violation caused by HUKO's protection mechanism indicates a sensitive control transfer event or a sensitive data access. To properly handle it, HUKO first examines the following information: (1)

the qualification bits which reveal the actual type of the violation, (2) the current state, and (3) the label of the faulting frame. Then it determines whether to allow the operation or to trigger a protection alarm based on information collected and the access control policies.

As we stated in Section 4.2.2, subjects in HUKO can freely read and write their own code and data. Also, inter-subject read accesses are always allowed in our default policy. These allowed events do not cause any EPT violation so that they cannot be logged by the hypervisor. However, for forensics purposes, the system administrator may want to audit some types of crucial events yet still allow these events to happen. Hence, HUKO adds another action named *audit allow* to enable logging of these specific data accesses. To implement the audit allow mechanism, HUKO sets the access rights in the corresponding EPT entries so that audit-allowed events would cause EPT violations and be audited by the hypervisor. Then HUKO *emulates* the offending instructions without changing the previously set access rights. In this way, the audit allow operation is completed and the EPT entries can still be used to trap further events of the same kind.

Protecting Control Flow Integrity. As previously stated, HUKO sets the execution bits of multi-HAP entries so that only untrusted extension code can be executed in the untrusted extension protection state. When an execution violation indicating a control transfer from an untrusted extension to the OS kernel occurs, HUKO enforces the control flow integrity rules under the following conditions: (1) the untrusted extension is calling the kernel via `call` and `jmp` instructions. In this case, HUKO allows the operation only when the violating address belongs to a trusted entry point. This prevents untrusted extensions from accessing unauthorized kernel functions or jumping to arbitrary positions in the kernel. (2) The kernel preempts the untrusted extension for higher priority interrupts. In this case, HUKO ensures that the violating address belongs to an interrupt handler routine in the IDT table. (3) The extension returns to the kernel from a previous call. This could be leveraged by return-oriented rootkits to divert the control flow to a sequence of return-oriented instructions in the kernel. To tackle this problem, HUKO maintains a separate return address stack to keep track of the call/return sequences between the OS kernel and untrusted extensions. In this way, we guar-

antee the return address to the kernel must correspond to the address of the kernel code that made the call. Also, the sequence of return addresses must satisfy the last-in-first-out property. Considering the fact that most return-oriented attacks need an initial return to the first return-oriented instruction sequence, our approach provides an effective counter method.

Handling DMA writes. Besides memory writes performed by CPU instructions, DMA is another way for extensions to write data into the kernel memory. Previous proposals [62] have limited capability of handling DMA because the data transfer is not controlled by the processor or memory controller. Fortunately, the introduction of hardware IOMMUs (Intel's VT-d and AMD's IOMMU) brings the possibility to efficiently mediate and control DMA memory access. When used in virtualization, the IOMMU can enable pass-through device models which support independent address translations using IOMMU page tables for DMA activities.

In HUKO prototype, we leverage the DMA remapping mechanism provided by Intel's VT-d technology [106] to protect the kernel integrity from DMA writes. Currently we explicitly set the IOMMU page tables so that pages labeled as OS kernel and trusted extensions cannot be used in DMA. On the other hand, HUKO allows DMA activities on the pages that are labeled as untrusted extensions. Our ongoing work employs multiple IOMMU page tables and switch facilities for different protection states, which is very similar to the multi-HAP mechanism. This scheme introduces new DMA object labels shown in Table 4.1 and allows the kernel and all extensions to do DMA in a protected manner. Another more flexible optimization is to integrate the IOMMU page tables with the multi-HAP page tables so that IOMMU can utilize the guest-to-machine physical address translation as well as access control enforcement provided by the multi-HAP mechanism.

Supporting Exceptions. Given the complexity of commodity operating system kernels and the variety of enormous extensions in the wild, it is necessary for HUKO to support exceptions for access control enforcement. There are three types of exceptions in HUKO. The first type offers an untrusted extension the privilege to write into specific objects in the kernel. The second type allows an untrusted extension to make certain calls to the kernel, but not through trusted

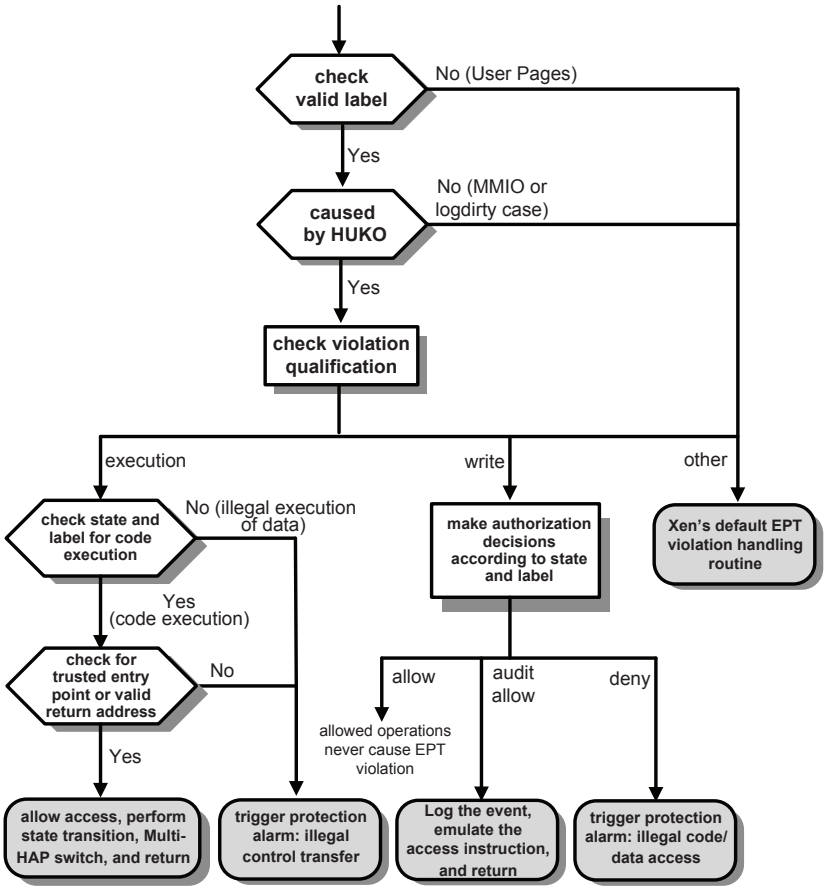

Figure 4.5. The EPT violation handling diagram of HUKO.

entry points. The third type of exception is about exporting write permissions in kernel stack frames. These exceptions are provided by the administrator to achieve specific needs on flexibility and performance, and they are stored and protected in the VMM memory space. Section 4.4.1 provides a further discussion in Linux OS.

In our current prototype implementation HUKO uses mixed page labels to handle exceptions. Pages that contain exception objects are labeled as "*mixed exception*", and the hypervisor will check the virtual address upon each violation to determine whether the event is an exception. This approach has bad performance in case the number of exceptions is large or exceptions occur frequently. We have an optimized design for handling exceptions and mixed pages. In that scheme, HUKO copies all the exception objects onto a set of allocated exception pages. By dynamic patching of instructions, HUKO redirects all the operations accessing

exception objects to the corresponding copy on the exception pages at the run time. This method reduces the total number of EPT violations on exception pages and mixed pages. We plan to implement this optimization in our future work.

4.3.6 Modifications to Xen

We implemented HUKO by modifying the Xen hypervisor (version 3.4.2 x86-64 HVM Guest), which is a full-fledged open source hypervisor commonly used in various enterprise systems. The HAP mechanism used in the isolation and labeling component is based on Intel's EPT, yet it does not require much effort to adapt AMD's NPT. The total amount of code added to the Xen hypervisor is approximately 3,300 lines. And the Linux implementation of the labeling helper trusted extension consists of about 450 lines of code.

A major effort of our prototype implementation is to extend the memory virtualization sub-system of Xen to support the multi-HAP mechanism. In HUKO prototype, each HAP table is essentially a four-level EPT paging structure. The root-level index of each paging structure is stored in an array named `huko_phys_table_index`, which is placed in the architecture-specific per-domain structure `arch_domain`. To construct multi-HAP tables, HUKO first traverses all the existing physical-to-machine (p2m) mappings from the domain's `page_list`. Then it allocates EPT entries using free pages maintained by `p2m_freelist`, which are Xen's reserved pages for storing p2m mappings. The security label of each GFN is stored in bits 61:57 of the corresponding EPT entry and managed by the labeling component. HUKO then decides the access rights of an EPT entry from its security label, the MAC policy as well as the protection state which it belongs to. HUKO keeps this allocation process until all HAP tables are established. During each state transition, HUKO switches among multiple EPT paging structures by changing the EPTP pointer and associated VPID in the VMCS fields.

For each protection state, we introduced a security control block (SCB) which is linked to the `domain` structure. The SCB stores essential information for tracking a protection state, such as the identity of the current subject, the virtual address range of the subject's code and data, the previous protection state, the address of the last entry point, a copy of stack pointers, and a link to its return address stack.

To achieve mediation and policy enforcement, we added additional routines to the paging violation handler of EPT and the Vt-d pass-through (IOMMU) driver, which are `ept_handle_violation()` and `iommu_page_fault()`, respectively. We exported two new hypercalls to the labeling helper for delivering run-time information to the labeling component.

4.4 Evaluation

In this section, we describe the deployment and experimental evaluation of the HUKO prototype. There are two goals of our evaluation. The first is to evaluate HUKO's effectiveness for defending against various real-world malicious extensions that damage the OS kernel integrity in different ways. The second goal is to measure the performance cost introduced by HUKO using both application-level and micro benchmarks.

All experiments were conducted on a Dell PowerEdge T310 Server with a 2.4GHz Intel Xeon X3430 and 4GB memory. The Xen hypervisor version is 3.4.2. The dom0 system is fedora 12 with kernel version 2.6. We used a 64bit Ubuntu Linux (8.04.4) with kernel version 2.6.24 as our guest OS. All Linux partitions were configured to use the `ext3` file system. For Windows experiment, we chose Windows XP SP2 as our guest system.

4.4.1 Deploying HUKO

As stated in Section 4.2.1, HUKO is intended to minimize the required effort for deploying the protection system. Instead of establishing protection domains at the OS layer [62] or at the hardware architecture layer [1], the implementation of almost all the functionalities (i.e., memory protection and access control) in HUKO is at the virtualization layer, which makes the protection mechanism guest-independent, adaptive, and easy-to-undeploy. Moreover, HUKO does not enforce access control for specific kernel objects, and it only has several generic types for object labeling. While this approach sacrifices the benefits of semantic-rich access control at finer granularity, it does offer a much easier configuration compared to rich-typed protection system such as SELinux [26]. In the following paragraphs

Untrusted Extension	Behavior	Violation Triggered	Violating Object Label
EnyeLKM	add binary code to kernel	Illegal code access	OTHER OS CODE
all-root	DKOM (modify task_struct) modify control data (sys_call_table)	Illegal data access	OS DATA
adore-ng	modify function pointers	Illegal data access	OS DATA
hp	DKOM (modify task_struct linked list)	Illegal data access	OS DATA
lvtes	call unauthorized function (module_free)	Invalid code execution	OTHER OS CODE
return-oriented extension	modify return addr. on the stack	Invalid return address	Return addr. stack
FUTo (Windows)	DKOM (modify PspCidTable)	Illegal data access	OS DATA
TCPIRP (Windows)	modify function pointers	Illegal data access	OS DATA
basic_int (Windows)	add binary code to kernel	Illegal code access	OTHER OS CODE

Table 4.2. Protection effectiveness of HUKO against a collection of malicious extensions.

we use the Linux OS as an example to briefly describe the deployment of HUKO.

The first step is to set up the basic information about kernel layout, objects and TEPs. In Linux, most of these information could be acquired from the kernel symbol table associated with the specific kernel. For example, the address range of Linux kernel code is determined by kernel symbol `_text` and `_etext`. Similarly, the boundaries of initialized and uninitialized kernel static data can be identified by symbol `_edata` and `_end`. At runtime, the labeling helper is responsible for collecting dynamic information for object labeling. For instance, the code and data range for an extension could be retrieved from the accounting data structure `module` when the extension is being loaded into the kernel.

In Linux, most kernel APIs and global data are exported to the kernel symbol tables using the `EXPORT_SYMBOL` macro. The address of kernel symbols can also be retrieved in the `System.map` file. In this way we could collect all the entry addresses for exported kernel functions. In our current prototype, we treat all the exported kernel APIs as the Trusted Entry Points (TEPs). In our future work, we are expecting to extend HUKO to achieve the least privilege property, by which we infer and enforce the set of kernel APIs that a specific extension can call. We do a further discussion on this issue in Section 4.5.

Besides common settings, administrators sometimes also need to provide extension-specific exceptions to make an extension run correctly. There are mainly three types of exceptions in a HUKO system. The first type of exceptions consists of non-exported functions. In Linux, certain kernel functions are not explicitly exported, instead, they are accessed by direct address reference or address assigning

to function pointers. Fortunately, these cases are not recommended nowadays and getting rare in recent Linux kernels. To deal with them, the administrator should manually specific the entry address of these kernel APIs as TEPs. The second category of exceptions consists of OS kernel data of which the kernel intentionally grants write permission to extensions. In many cases, the shared data are used as various kinds of buffers and caches in the kernel, and they are usually still page-aligned. The labeling helper notifies the hypervisor when these data are allocated, and HUKO assigns `Shared_Data` type to these pages in the multi-HAP table to allow write access for both OS kernel and untrusted extension protection states. Shared data that are not page-aligned with non-shared kernel data are required to set up exceptions using mixed pages. Regarding write-sharing for kernel global variables, the administrator could specify their address in the exceptions according to the kernel symbol table. The third category of exceptions belongs to stack permission which OS kernel needs to grant extensions write permission to its local variables on the stack. For example, OS kernel could pass the address of a local variable to an extension in parameters during a function call. To address these situations, the administrator should specify the addresses of functions that require stack exceptions and how many previous frames need to be modified by each function. Then at the time that control returns to these functions, instead of synchronizing only its own stack frames of the extension, HUKO synchronizes all the necessary previous stack frames specified by the given exception.

4.4.2 Protection Effectiveness

We evaluated the effectiveness of HUKO for kernel integrity protection with a collection of malicious extensions on both Windows and Linux. These extensions include 8 real-world rootkits and one self-implemented malicious extension for return-oriented attacks, which are shown in Table 4.2. As a result, all of these malicious extensions triggered protection alarms once they attempted to damage the kernel integrity. In the following paragraphs we describe three representative experiments in detail.

Code Integrity. EnyeLKM [107] is a Linux kernel rootkit which modifies the

kernel text by putting "salts" inside `system_call` and `sysenter_entry` handlers. With HUKO protection, an illegal code modification alarm was triggered when either `set_idt_handler` or `set_sysenter_handler` was called. Both functions were trying to add binary text to kernel object labeled as `OTHER_OS_CODE`.

Data Integrity. The `all-root` [108] rootkit is a simple DKOM Linux kernel rootkit that modifies both control and non-control data to achieve privilege escalation. In its initialization routine `init_module`, this rootkit replaces the `sys_getuid` entry of the `sys_call_table` with its own function `give_root`, which changes the uid, gid, euid and egid field of the current `task_struct` to 0 (root). In this attack, the first modified data belongs to static control data while the latter belongs to dynamic non-control data. When we launched this attack in a system protected by HUKO, it immediately triggered a protection alarm indicating an illegal data access (caused by the first modification) from untrusted extensions to an object labeled as `OS_DATA`. In order to test the second data modification, we deliberately made decisions to allow the first modification and let the system continue to run. Then we executed a `getuid` system call from the user space to trigger the malicious replacement function. Again, HUKO triggered an illegal data access alarm, which was also caused by directly modifying dynamic non-control kernel data (labeled as `OS_DATA`) at the "untrusted extension" protection state.

Control Flow Integrity. Besides malicious extensions that modify control-data (e.g., function pointers) or make illegal call/jump to the kernel, the return-oriented attack is another way of tampering control flows in the kernel. To evaluate HUKO's effectiveness in countering such attacks, we implemented a return-oriented malicious extension in our experiment. Upon called, this extension modifies its return address on the stack to an arbitrary point in the kernel text area, which is recognized as a return-instruction gadget. We loaded this extension to a Linux system protected by HUKO. As a result, HUKO successfully prevented the control flow diversion caused by the modified return address, since the LIFO property of the return address stack was no longer kept.

Benchmark	**Untrusted Extensions**	Number of Protection State Transitions	Native Performance	HUKO Performance	**Relative Performance**
Dhrystone 2	`8139too + ext3`	N/A	10,855,484 lps	10,176,782 lps	0.94
Whetstone	`8139too + ext3`	N/A	2,270 MWIPS	2,265 MWIPS	1.00
Lmbench (pipe bandwidth)	`8139too + ext3`	N/A	2,535 MB/s	2,213 MB/s	0.87
Apache Bench (throughput)	`8139too`	56,037	2,261 KB/s	1,955 KB/s	0.86
Kernel Decompression	`ext3`	17,471,989	35,271 ms	44,803 ms	0.79
Kernel Build	`ext3`	148,823,045	2,804 s	3,106 s	0.90

Table 4.3. Performance results of application-level benchmarks.

4.4.3 Performance Overhead

To measure the performance cost introduced by HUKO, we ran a set of benchmarks to compare the performance of a guest system protected by HUKO with one that does not. For each benchmark, we labeled one or several relevant kernel extensions as untrusted so that they were isolated from the kernel. For all workloads we enforced the sample policy showed in Table 4.1. To fully test HUKO's performance overhead under stressed conditions, we chose two largest and most active kernel extensions in our Linux system: `8139too` and `ext3`. The `8139too` is the network interface card driver and the `ext3` extension is the file system module. These extensions are invoked multiple times for each network I/O requests or file system operations so that they have the highest control transfer rates with the OS kernel. Hence, marking them as untrusted generally represents the worst-case performance of HUKO when the system is performing I/O intensive tasks.

The application benchmarks and their configuration are presented as follows: (1) Dhrystone 2 of the Unix Bench suite [109] using register variables. (2) Double-Precision Whetstone of the Unix Bench. (3) LmBench [110] pipe bandwidth measuring the performance of IPC interface provided by the kernel. (4) Kernel Decompression by extracting a Linux 2.6.24 kernel gzipped tarball using `tar -xzf` command. (5) Building a 2.6.24 Linux kernel using default configurations. (6) Apache Bench configured to have 5 concurrent clients issuing 20 http requests (16KB HTML) per client.

Table 4.3 presents the results of these application level benchmarks. The second column indicates which extension is labeled as untrusted, while the third column shows the total number of protection state transitions in each workload. Some numbers are not available because the corresponding workload is part of a continuous benchmark suite. From the results, we can see that the performance of HUKO system is from 0.79 to 1.00 of the baseline. We also found that the performance overhead added-on by HUKO largely depends on the frequency of control transfers between untrusted extensions and the kernel. Hence, if the workload is CPU-bound, the performance cost is minimal. The overhead gets higher only when an untrusted extension is responsible for highly frequent operations such as disk I/O. In the kernel decompression experiment, the protection state transfer rate reaches about 39,0000 per second, which renders HUKO the worst case of performance: 0.79 of the baseline.

Besides application level benchmarks, we also performed several micro-benchmark tests on process creation with Lmbench. We labeled `ext3` and `8139too` as untrusted extensions in our system protected by HUKO. Regarding the test item `process fork + exit`, it took HUKO system 100.31 μs to complete the operation while the native system took 92.87 μs. For `process fork + execve`, HUKO system spent 377.47 μs compared to the native time of 296.47 μs. For `process fork + /bin/sh -c`, it took HUKO system 884.57 μs compared to the native time of 697.38 μs.

4.5 Limitations and Future Work

We believe that HUKO provides a transparent security layer which greatly enhances the integrity protection for commodity operating system kernels. Nonetheless, it also has limitations in defending against certain security threats. In the following, we discuss these limitations and possible solutions as our future direction.

Kernel APIs. In HUKO system, controls from untrusted extensions to the OS kernel are restricted to a set of trusted entry points, which are essentially legitimate kernel APIs that exported to kernel extensions. However, in commodity operating

systems, the kernel is usually not designed to tolerate or defend against malicious extensions, which may results in the lack of robustness and security of kernel APIs. Moreover, programming languages used to build commodity OS kernels security do not support features like type enforcement. For these reason, it is possible that attackers can exploit the "legitimate" kernel interface to subvert the integrity of kernel. Examples of such attacks include: (1) calling legitimate kernel APIs with undesired object reference to compromise kernel objects, (2) abuse of privileges, (e.g., video cam driver accesses kernel APIs for the networking stack), and (3) exploiting memory and type bugs of the kernel API functions. Comprehensively addressing these issues would require major design improvements on specific kernel (e.g., [30, 5, 66]), such as kernel object model, access control model, type enforcement, verification and privilege separation. In addition, these approaches can be layered atop HUKO, which serves as a VMM-level reference monitor for mediating kernel object access, checking API calls and their parameters.

To obtain a better mandatory security policy, we are looking for a deeper understanding of the behavior of the OS kernel. In specific, we are interested in figuring out security-sensitive kernel data along the execution path of each TEP. This could be achieved by static program analysis with security annotations. Based on the properties such as privilege, availability level and resource category of these kernel data, we could achieve a good classification of TEPs in terms of resource manipulation and privilege. In this way, the security and resource semantics of TEPs are further revealed, which could help improve the security of TEPs whose privileges are originally unified in commodity OSes.

Information flow. Another category of possible attacks is through explicit and implicit information flow. For instance, OS kernel may explicitly grant write access to extensions on its own data objects (e.g., via shared memory, API or messages), on the other hand, extensions may write low integrity data to some places where kernel may read afterwards. Both situations violate the traditional integrity model. It is known that there is no existing information flow control inside commodity OS kernels since tracking fine-grained information flow is costly in regard to current programming language and architecture. Alternatively, we plan to investigate applying end points such as input filters and verifiers between OS kernel and ex-

tensions to regulate the function parameters and information passed to the OS kernel.

4.6 Summary

We have presented the design, implementation and evaluation of HUKO, a hypervisor-based layered system that comprehensively protects the integrity of commodity OS kernels from untrusted extensions. HUKO leverages several contemporary hardware virtualization techniques as well as its novel software design to achieve its design principles: multi-aspect protection, acceptable performance and ease-of-adoption. Our experiments show that HUKO can effectively protect the kernel integrity from various kinds of malicious extensions with an acceptable performance overhead. We believe that HUKO provides a practical framework for running untrusted extensions in OS kernel with enhanced integrity protection for commodity systems.

Chapter 5

SILVER: Fine-Grained Privilege Separation in OS Kernel

5.1 Introduction

As commodity operating systems are becoming more and more secure in terms of privilege separation and intrusion containment at the OS level, attackers have an increasing interest of directly subverting the OS kernel to take over the entire computer system. Among all avenues towards attacking the OS kernel, untrusted kernel extensions (e.g., third-party device drivers) are the most favorable targets to be exploited, as they are of the same privilege as the OS kernel but much more likely to contain vulnerabilities. From the security prospective, these untrusted extensions should be treated as *untrusted principals* in the kernel space. In order to prevent untrusted extensions from subverting kernel integrity, many research approaches [62, 42, 43, 111] are proposed to isolate them from the OS kernel. These approaches enforce memory isolation and control flow integrity protection to improve kernel security and raise the bar for attackers. However, in many situations, strong isolation along is still inadequate and inflexible to secure interactions between OS kernel and untrusted principals, for the following reasons:

Firstly, in commodity OSes such as Linux, kernel APIs (i.e., kernel functions legitimately exported to extensions) are not designed for the purpose of safe communication. Thus, even if untrusted extensions are memory-isolated and constrained

to transfer control to OS kernel only through designated kernel functions, attackers can still subvert the integrity of the OS kernel by manipulating parameter inputs of these functions. For example, an untrusted extension could forge references to data objects that it actually has no privilege to access. By providing such references as input of certain kernel functions, attackers could trick the OS kernel to modify its own data objects in undesired ways.

Secondly, either OS-based or VMM-based memory protection mechanism can only enforce page-level granularity on commodity hardware, which provides avenues for attackers exploiting such limitation. For example, attackers can leverage buffer/integer overflow attacks to compromise data objects of OS kernel by overflowing adjacent data objects from a vulnerable driver in the same memory slab. It is difficult for a page-level access control mechanism to address this problem for its inability to treat data objects on the same page differently.

Finally, current isolation techniques are limited to support sharing and transfer of data ownership in a flexible and fine-grained manner. Considering situations that the OS kernel would like to share a single data object with an untrusted device driver, or accept a data object prepared by a driver, in case of strong isolation, it often requires the administrator to manually provide exceptions/marshaling to move data across isolation boundaries. Although there are clean-slate solutions such as multi-server IPCs in micro-kernels [12] and language-based contracts [5] to address this problem, these approaches are difficult to apply to commodity systems, for the reason that they both require developers to change the programming paradigm fundamentally.

To address these shortcomings, we have the following insight: beside isolation, protection systems should provide a clear resource management of kernel objects, as well as a general method for secure communication. In OS-level access control mechanism such as LSM [28], the kernel maintains meta-information (e.g., process descriptors and inodes) for OS-level objects like processes, files and sockets, and it also provides run-time checks for security-sensitive operations. Such mechanism enables powerful reference monitors such as SeLinux [26] and Flume [32] to be built atop. In contrast, there is little security meta data maintained for kernel-level data objects, nor security checks for communication between OS kernel and untrusted kernel principals.

This chapter presents the design and implementation of SILVER, a framework that offers transparent protection domain primitives to achieve fine-grained access control and secure communication between OS kernel and extensions. SILVER's key designs are two-fold: (1) SILVER manages all the dynamic kernel data objects based on their *security properties*, and achieves fine-grained access control with the support of memory protection and run-time checks; (2) Communication between OS kernel and various untrusted kernel extensions is governed and secured by a set of unified primitives based on existing information flow integrity models without changing programming paradigm significantly. Protection domains in SILVER are enforced by the underlying hypervisor so that they are transparent to kernel space programs. Hence, from the perspective of kernel developers, the kernel environment remains as a single shared address space, and developers can still follow the conventional programming paradigm that uses function calls and reference passing for communication. Kernel program developers could utilize SILVER to ensure neither the integrity of their crucial data would be tampered nor their code would be abused by untrusted or vulnerable kernel extensions, thus prevent attacks such as privilege escalation and confuse-of-deputy.

SILVER employs several novel designs to enable our protection domain mechanism. First, in SILVER, protection domains are constructed by leveraging hardware memory virtualization to achieve transparency and tamper-proof. The hypervisor-based reference monitor ensures that security-sensitive cross-domain activities such as protection domain switches will eventually be captured as exceptions in virtualization. Second, we propose a new kernel slab memory allocator design, which takes advantages of SILVER's virtualization features such as page labeling and permission control, with a new organization and allocation scheme based on object security properties. The new memory management subsystem exports API to developers to allow them managing security properties of its allocated objects, and enforce access control rules throughout their life time. Finally, SILVER introduces two new communication primitives: transfer-based communication and service-based communication for securing data exchange and performing reference validation during cross-domain function calls.

We have implemented a prototype of SILVER for the Linux kernel. Our system employs a two-layer design: a VMM layer for enforcing hardware isolation,

reference monitoring and providing architectural support for page-level security labeling, as well as an OS-subsystem for achieving the high-level protection mechanism and offering APIs to kernel programs. We have adapted real-world Linux device drivers to leverage SILVER's protection domain primitives. The evaluation results reveal that SILVER is effective against various kinds of kernel threats with a reasonable overhead on memory consumption and run-time performance.

The rest of this chapter is organized as follows. Section 5.2 illustrates the threats and presents our solution using an abstract security model. Section 5.3 describes the design and implementation of SILVER architecture in detail. Section 5.4 covers the evaluation of our prototype from aspects of deployment, security and performance. We explain the limitations of our prototype and propose our future work in Section 5.5. Section 5.6 concludes.

5.2 Approach Overview

In this section we first present several examples of kernel threats to illustrate shortcomings stated in Section 5.1. We then describe our threat model, and give an overview of our approach.

5.2.1 Motivating Examples

Kernel heap buffer overflow. Jon [112] illustrates a vulnerability in the Linux Controller Area Network (CAN) kernel module which could be leveraged to trigger controllable overflow in the SLUB memory allocator and eventually achieve privilege escalation. The exploit takes advantage of how dynamic data are organized in slab caches by the SLUB allocator. In specific, the attack overflows a `can_frame` data object allocated by the CAN module and then overwrites a function pointer in a `shmid_kernel` object, which is owned by the core kernel and placed next to the `can_frame` object. Although there are many ways to mitigate this particular attack (e.g., adding value check and boundary check), the fundamental cause of such kind of attack is that the OS kernel is not able to distinguish data objects with different security properties. In this case, data object `shmid_kernel` is owned

by OS kernel principal, and it is of high integrity because it contains function pointers that OS kernel would call with full privilege. On the other hand, data object `can_frame` is created and owned by the vulnerable Controller Area Network kernel module principal with a lower integrity level. Unfortunately, Linux kernel does not manage the owner principal and integrity level of dynamic data objects, which results in placing these two data objects on the same `kmalloc-96` SLUB cache with the vulnerability.

Kernel API attacks. As mentioned in Section 5.1, even with strong isolation and control flow integrity protection, untrusted extensions can still subvert the integrity of OS kernel through manipulating kernel APIs. For example, let us consider a compromised NIC device driver in Linux which has already been contained by sandboxing techniques such as hardware protection or SFI. Due to memory isolation, the untrusted driver cannot directly manipulate kernel data objects (e.g., process descriptors) in kernel memory. However, the attacker could forge a reference to a process descriptor and cast it as `struct pci_dev *` type, which he would use as a parameter to invoke a legitimate function (e.g., `pci_enable_device`). By carefully adjusting the offset, the attacker could trick the OS kernel to modify that particular process descriptor (e.g., change the `uid` of the process to be zero to perform privilege escalation) and misuse its own privilege. We consider such threat as a confused deputy problem caused by insufficient security checks in Linux kernel APIs. Thus, to ensure kernel API security, upon receiving a reference from caller, a kernel function should distinguish the security principal that provides the reference, as well as determine whether that principal has the permission to access the data object associated with the reference.

5.2.2 Threat Model

In SILVER, kernel developers leverage protection domain primitives to protect the integrity of OS kernel in case that untrusted extensions are compromised by attacker. A compromised extension may attempt to subvert a protection domain in many different ways, which may include: (1) directly modifying code/data via write instruction or DMA; (2) control flow attacks that call/jump to unauthorized code in kernel; (3) memory exploits such as stack smashing or buffer overflows;

(4) confused deputy attack via reference forgery; (5) tampering architectural state such as crucial registers. We discuss how SILVER is designed to defend against or mitigate these attacks throughout the rest of the chapter.

In this chapter, we primarily focus on the protection of *integrity*. Although we are not seeking for a comprehensive secrecy protection against private information leakage, SILVER could indeed prevent untrusted principals directly read crucial data (e.g., crypto keys) from a protection domain.

SILVER employs a VMM for reference monitoring and protecting the integrity of its components in the OS subsystem. Hence we assume that the VMM is trusted and cannot be compromised by the attacker.

5.2.3 Protection Domain in SILVER

In SILVER, protection domains can help the OS kernel and other trusted entities collaborating with untrusted code without worrying about the compromise of integrity so that they can exchange information, delegate privilege and export services in a more explicit, secure, and controlled manner. In the following paragraphs, we give an overview of the design goals of our approach.

Data management based on security properties. SILVER maintains security metadata for dynamic data objects in the kernel to keep track of their security properties. For example, for each dynamic objects allocated, besides basic information such as address and size, SILVER also maintain records of its *owner principal* and *integrity level*. Moreover, kernel data objects are managed based on these security properties, and the organization scheme takes advantage of labeling and memory protection primitives provided by SILVER's hypervisor. Such organization guarantees that security-sensitive events will be completely mediated by the reference monitor, which would make security decisions based on security properties of principal and data objects. In this way, SILVER achieves data object granularity in protection domain construction and security enforcement, and addresses challenges stated in Section 5.1.

For example, with SILVER, the kernel buffer overflow attack in Section 5.2.1 would no longer succeed, since data object `shmid_kernel` with security property <OS kernel, high integrity> would never be placed adjacent to data object

`can_frame` with security property <CAN module, low integrity>. Thus, through compromising the vulnerable CAN module, the attacker can only overflow low integrity data objects that only CAN module has write access to, but never hurt the integrity of security-sensitive kernel data. Moreover, the kernel API attack in Section 5.2.1 could also be prevented by SILVER, since SILVER is able to determine whether the caller principal has the access permission to the data object referred by the pointer parameter passed.

Security controlled by developers. Many run-time protection systems ([67, 62, 26, 80, 44]) rely on mandatory access control mechanisms in which the access control rules are completely decided by the protection system or the system administrator. However, mandatory protection itself generally has difficulties in achieving fine-grained policies that closely express the application semantics. In specific, constructing dynamic MAC policies for multi-principal interaction requires complex effort such as role assigning, state definition and type enforcement, which may be too difficult for system administrators to configure correctly. To address this shortcoming, SILVER allows kernel developers to control security properties of its own code and data in a fine granularity to achieve flexibility.

We illustrate security decisions controlled by program developers as follows: (1) by leveraging extended allocation APIs, developers can specify which data objects are security-sensitive while others can be globally shared with untrusted principals by assigning integrity labels to its data objects; (2) developers could control the delegation of data object ownership and access permissions with other principals by relying on SILVER's transfer-based communication primitive; (3) developers could ensure data integrity when providing service to or requesting service from other principals by using the service-based communication primitive; (4) developers can control which services (functions) to be exported to which principals by creating entry points both statically and at run-time; (5) developers could use endorsement functions and reference checking primitives to validate received data and reference; (6) developers (and system administrators) could accommodate trust relationships with protection domain hierarchy.

Noted that although SILVER's primitive could help both participating security principals to achieve secure communication, the security of a protection domain

does not rely on other domain's configuration or security status. For example, as long as the OS kernel programmer properly use SILVER's primitives to enforce isolation and secure communication, the integrity of OS kernel would not be compromised by any untrusted extension which may either fail to use SIVER's primitives correctly or be totally compromised by attacker.

Practical deployment. SILVER requires existing kernel programs to modify their code to leverage the security benefits of protection domains, but its deployment efforts are still practical. Firstly, compared with language-based approaches, SILVER does not require the program to be rewritten entirely. Instead, it only requires the adaptation of a few extended kernel APIs. Moreover, the deployment procedure of SILVER can be *selective* and *incremental*. For instance, one can leverage SILVER's extended API `kmalloc_pd` to declare security-sensitive data for extra security guarantee. However, it can also keep using the original `kmalloc` to leave that allocated object unprotected. Secondly, in contrast to approaches (e.g., micro-kernels) that change the programming paradigm completely, transparent protection domains in SILVER preserve programming conventions in the commodity OS kernel as much as possible. For example, kernel entities still rely on function calls and pointer passing for data communication, and we do not want to replace them with multi-servers and message-passing mechanisms.

5.2.4 Abstract Model

In this section we present an abstract model, describing our approach in a few formal notations. The basic access control rules of our model follow existing integrity protection and information flow models [15, 16, 32] with a few adaptations. In SILVER, a kernel protection domain represents an execution entity in the kernel space. Examples of kernel protection domains include the OS kernel, device drivers and other kernel extensions. In SILVER, kernel protection domains are declared by the developer who would like to protect the integrity of its program from being tampered by other programs in kernel.

In our model, a kernel protection domain is defined as a three-tuple: $S =< p, D, G >$, where: (1) p is the principal associated with the domain. For each protection domain S in kernel, p is unique and immutable so that it can be used as

the identifier of the protection domain. Thus, we denote a protection domain with principal p as S_p. (2) D is the set of data object owned by the principal. Every data object is associated with an integrity level, which can be either high, low or global shared. We denote the subset of high integrity data objects as D^+ and the subset of low integrity data objects as D^- so that $D = \{D^+, D^-\}$. For programmers, high integrity label usually means that the labeled data are private or security-sensitive, and thus not meant to be manipulated by others. Low integrity labels, on the other hand, are often applied to data of low importance, shared with untrusted principal or received but not yet validated and endorsed. (3) G is the set of entry point objects, which are essentially entrance addresses through which a principal could transfer its control to another principal. Entry points are specified by the developer on a per-principal basis, yet some of them can also be declared as global shared. For the global shared data objects and entry points, SILVER virtually organizes them in to a global low-integrity protection domain denoted as S_-. We define the set of rules that govern protection domain activities as follows:

- **Data creation.** A principal p can create data objects of either integrity level in its own protection domain. p can also degrade any high integrity data object $d \in {D_p}^+$ to low integrity level so that $d \in {D_p}^-$.

- **Integrity protection.** A data object can only be possessed by only one principal at any time. A principal p can write to a data object d iff $d \in D_p$. p can read from d iff $d \in {D_p}^+$. While p cannot read $d \in {D_p}^-$ directly, p has the capability to increase the integrity level of d via an endorsement API provided by SILVER.

- **Data communication.** In SILVER, data communications are achieved by moving data objects from one protection domain to another. In order to send data to another principal q, p can move its data object $d \in D_p$ to low integrity part of domain S_q so that $d \in {D_q}^-$. However, to ensure that d is safe in regard to the integrity of q, d is kept to be in low integrity and cannot be read by q until q sanitizes and endorses the input data and render d high integrity ($d \in {D_q}^+$).

- **Cross-domain calls.** Another important method for inter-domain commu-

nication is through calling remote functions exported by other principals. Exporting functions to a principal q is achieved by creating entry point objects in q's domain. To prevent the abuse of code of a protection domain principal, SILVER guarantees that calling through entry points granted by p is the *only* way to transfer control to principal p. Data transfers through cross-domain calls must obey the previous data communication rules.

Protection domain hierarchy. Besides mutually untrusted principals, SILVER introduces protection domain hierarchy to accurately express one-way trust, which is more common in practice (e.g., OS kernel and untrusted extensions). In specific, SILVER allows one principal q or sysadmin to designate another domain S_p as the *parent protection domain* of domain S_q. The restriction of control/data flow rules for parent-child domains are relaxed in the following ways: (1) p can directly create high integrity data d within its child domain S_q so that $d \in {D_q}^+$; (2) p have the full write access permission to all the data object in D_q. High-integrity data objects of p are also considered as high-integrity data for q (${D_p}^+ \subset {D_q}^+$), thus can be read by q directly; (3) p can call arbitrary functions owned by principal q. Noted that the global shared virtual protection domain S_- is the child domain of all other protection domains in the kernel.

5.3 System Design and Implementation

5.3.1 Overall Design

To design a run-time system which enforces our model stated in Section 5.2.4, we have faced several design questions. The first question is on how to achieve a reference monitor for activities in the kernel space, where there is no hooks for mediating kernel object access, and no explicit "context switches" for distinguishing kernel principals. Another major challenge comes from achieving data object granularity for principal security control. In a commodity OS such as Linux, all the dynamic data objects owned by various principals are placed on a single heap without distinction. There is only one global namespace (i.e., virtual address) from which any code can refer to any object in the kernel space. Moreover, there is little meta

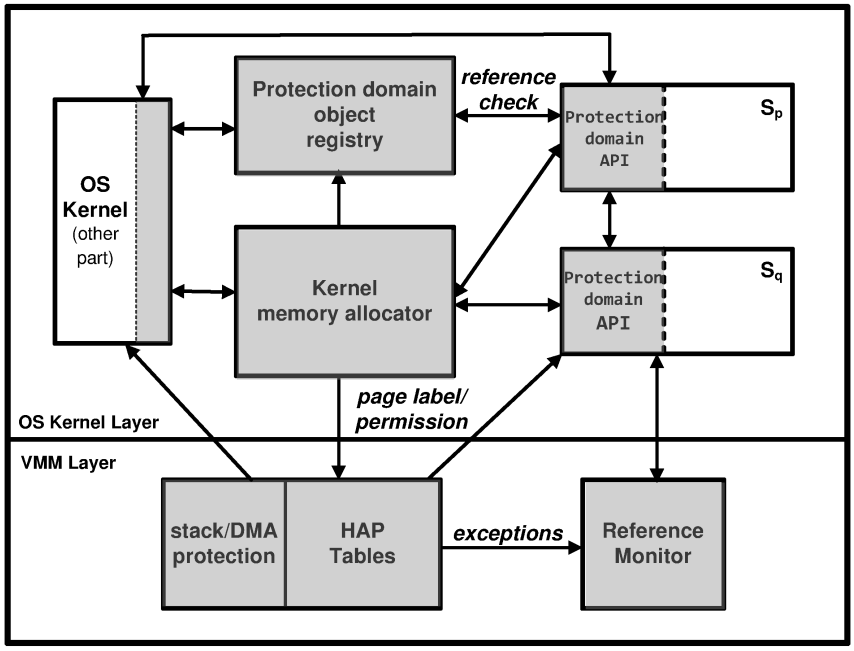

Figure 5.1. The architecture of the SILVER framework.

information for describing security properties of kernel objects. Hence, we need to develop new designs to support fine-grained policies and achieve accountability.

To address the above questions, SILVER exploits several architectural (hardware and virtualization) features to achieve strong isolation and a coarse-grained, OS-agnostic access control mechanism based on page permissions. On top of these facilities, we design a subsystem for Linux kernel to achieve accountability and fine-grained security control. The kernel subsystem includes a specifically designed kernel memory allocator implementing the core functionality of protection domain primitives, a kernel object registry for accounting kernel objects and supporting reference check, and a set of kernel APIs exported to principals for controlling security properties of their data, performing secure communication and granting capability to other principals. Figure 5.1 illustrates the overall design of SILVER's architecture, with the components of SILVER in gray. The entire framework is divided into two layers: the VMM layer and OS subsystem layer, respectively. The reference monitor and architectural-related mechanisms are placed in the VMM layer to achieve transparency and tamper-proof.

5.3.2 The VMM Layer Design

The VMM layer components consist the bottom-half of the SILVER architecture. These components are responsible for enforcing hardware protection to establish protection domain boundaries, as well as providing architectural-level primitives (e.g., page permission control, control transfer monitoring) for upper-layer components in the OS-subsystem.

Principal isolation. In SILVER, each principal is confined within a dedicated, hardware-enforced virtual protection domain realized by the hypervisor. The protection domain separation is achieved by creating multiple sets of HAP (hardware-assisted paging)[1] tables for memory virtualization, one table dedicated for each virtual protection domain. Using such layer of indirection, each principal could have its own *restricted view* of the entire kernel address space, while the shared address space paradigm is still preserved (Figure 5.2). Furthermore, by leveraging IOMMU tables, the VMM enables a principal to control DMA activities within its protection domain by explicitly exporting DMA-write permission to other principals and designating DMA-writable pages in its address space. The VMM prohibits any other DMA writes to the protection domain. Finally, to prevent untrusted code tampering with the architectural state (e.g., control registers, segment selectors, and page table pointer) of other protection domains or the OS kernel, the hypervisor saves all the corresponding hardware state of one protection domain before the control transfers to another subject, and restores the saved invariant values once the control is switching back.

Mapping security labels to page permissions. The hypervisor in SILVER also provides a page-based access control mechanism using hardware virtualization. In specific, it exports a small hypercall interface to the OS subsystem of SILVER, allowing it to associate security labels to kernel physical pages. The low-level access control primitives are implemented by mapping security labels to page permissions (i.e., read, write, execute) in each principal's HAP table, which defines whether certain pages can be accessed by the principal via which permissions. In section 5.3.3, we further describe how SILVER achieves fine-grained data access control

[1] A contemporary processor feature which adds another layer of hardware page tables for guest-physical-to-machine-physical translation in memory virtualization.

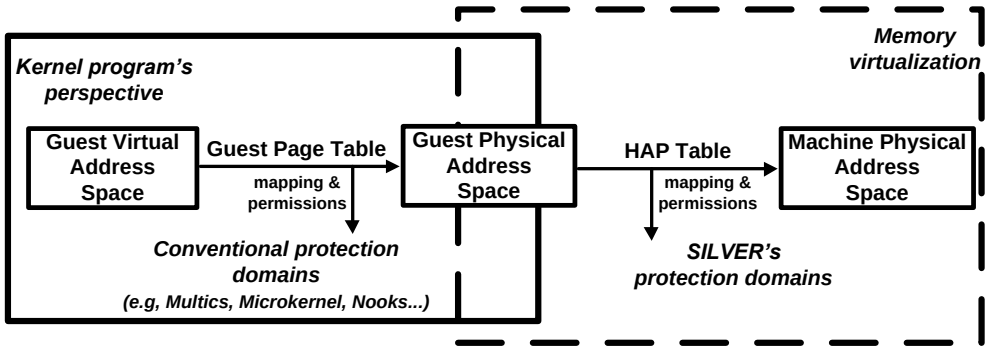

Figure 5.2. SILVER leverages memory virtualization to make protection domains transparent to the kernel space.

on top of these page-based mechanisms.

Securing control flow transfer. By setting up NX (execution disable) bits on corresponding HAP table entries representing pages owned by other principals, the hypervisor is able to intercept all control transfers from/to a protection domain through execution exceptions. Therefore, the reference monitor is fully aware which principal is currently being executed by the processor. The reference monitor then validates the <initiating principal, exception address> against the control transfer capability and the set of entry points designated by the owner principal of the protection domain, and denies all the illegal control transfers. To ensure the stack isolation and data safety during cross-domain calls, whenever a call is made by the protected code to an untrusted principal, the hypervisor forks a *private* kernel stack from the current kernel stack for untrusted execution, and it changes the untrusted principal's HAP table mapping of the stack pages to point to the new machine frames of the private stack. Since both virtual address and (guest) physical address of the stack are kept the same, untrusted code will have the illusion that it operates on the real kernel stack so that the original kernel stack semantics are preserved. After the call finishes, the hypervisor joins the two stacks by propagating legit changes from the private stack to the real kernel stack frames, guaranteeing that only modifications to its own stack frames are committed. In this way, SILVER enforces that all principals have read permission to the entire kernel stack, but only have write permission to their own stack frames.

5.3.3 OS Subsystem Design

The OS subsystem is responsible for achieving fine-grained protection domain mechanism and providing APIs to kernel programs. It leverages the architectural primitives provided by the VMM layer by issuing hypercalls to the VMM.

5.3.3.1 Kernel memory allocator

The kernel memory allocator in SILVER is responsible for managing dynamic kernel objects according to the rules defined in Section 5.2.4, as well as providing primitives to kernel principals for controlling security properties of their data objects. It leverages the hypercall interface provided by the VMM layer for labeling physical page frames and manipulating page permissions for different principals. Based on these mechanisms, the allocator achieves the following key functionality: (1) it allows principals to dynamically create objects within specified protection domain and integrity levels. For example, a principal could create a high integrity object within its protection domain for holding crucial data, or it could create an object in its child domains; (2) It enables a principal to endorse or decrease the integrity level of its objects at run time; (3) It allows a principal to transfer its data objects to be a low-integrity data object in a contracted protection domain for passing data; (4) It restricts principals from accessing the global name space (i.e., kernel virtual address) to refer objects outside of its domain and provide access control according to the rules.

Our design is an extension to the SLUB allocator [113] of Linux, which manages the dynamic allocation and deallocation of kernel objects. The SLUB allocator maintains a number of SLUB caches, distinguished by size for allocation efficiency. There are two kinds of SLUB caches in the system: general purpose SLUB caches (e.g., `kmalloc-32`) and caches which are explicitly defined for frequently allocated data structures (e.g., `task_struct`). A SLUB cache allocates kernel objects from organized physical pages named *slabs*, which are initialized to have multiple instances of a specific type of objects. Each slab has a `freelist` pointer for maintaining a list of available objects. A slub can have four allocation states: `cpu_slab` (the current active slab for a given cpu), `partial_slab` (portion of the objects are used), `full_slab` (slab objects fully used) and `new_slab` (all

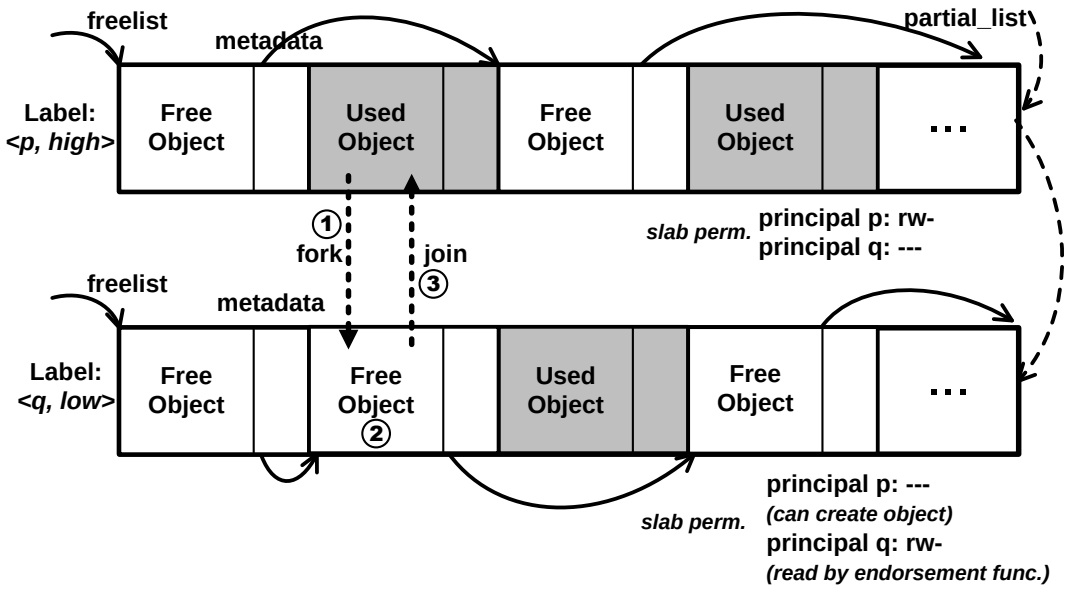

Figure 5.3. The layout of two slabs of the same slub cache involved in a service-based communication.

objects are available).

Organization. SILVER enhanced the Linux SLUB allocator by introducing heterogeneity to slabs for SLUB caches. In SILVER, each slab is associated with an extra label <*principal, integrity*>, and according to the label, it is restricted to contain kernel objects of the specified integrity level owned by the principal. The memory allocator achieves the slab access control by issuing hypercalls to the VMM layer, labeling and setting up page permissions. Figure 5.3 illustrates the organization of two `partial_slabs` from the same SLUB cache but with different owner principal and integrity levels. Their heterogeneous labels will eventually result in different page permissions in principals' HAP table, preventing principals from accessing objects that are disallowed by the access control rules. In general, for one principal p, there could be two kinds of slabs in each SLUB cache: domain high (for storing data objects that belongs to ${D_p}^+$), and domain low (for storing data objects that belongs to ${D_p}^-$). Moreover, in SILVER, there is a special global low slab for containing kernel objects of protection domain S_- defined in Section 5.2.4.

Allocation and Deallocation. The kernel memory allocator in SILVER provides a family of secure allocation APIs (e.g., `kmalloc_pd()`) for protection domain principals. These APIs follow the similar semantics of `kmalloc` family functions in Linux, except for having two extra parameters to designate the principal ID and integrity level of the object allocation. The work flow of the allocation procedure is described in Algorithm 1. One major difference between our allocation scheme and the original SLUB allocation algorithm is on the slab selection strategy, since SILVER must guarantee to pick the slab that matches the security criteria rather than to choose the first available objects from `cpu_slab` or `partial_slabs`. Once a new slab is created, SILVER must register the label to the VMM to establish principal access control before using it. On the other hand, the deallocation procedure is similar as the SLUB allocator, but it needs to check whether the requesting principal has the permission for freeing objects on the requested slab. If not, the free operation will be denied. The memory allocator also provides APIs to principals for changing the integrity level of their domain objects so that they have the ability to endorse their received data by implementing their endorsement functions.

Aside from slab objects, there is another major kind of dynamic data used by kernel programs: pages directly allocated by the free page allocator. Since this kind of data is already page-aligned, SILVER treats each of them as an object that occupies an entire slab (or multiple slabs), and labels the corresponding pages in the same way as the slab labeling. The access control strategies for allocated pages is also unified with the slab mechanism. Accordingly, we extend the APIs of the free page allocator to enable secure allocation and data communication.

5.3.3.2 Support for secure communication

As a major task, the OS subsystem in SILVER is responsible for offering secure primitives to principals for exchanging data, with the strong guarantee of integrity. The data communication is governed by the rules defined in Section 5.2.4. According to the model, using direct memory sharing to pass high-integrity data is prohibited in SILVER [2]. Instead, SILVER provides primitives for two primary types of data communication: *transfer-based* communication and *service-based* commu-

[2] Although principals can still declare unprotected data sharing via the special $S_{\_}$ domain.

Algorithm 1 The procedure for handling allocation requests from a protection domain principal

1: **if** label $< principal, integrity >$ of current `cpu_slab` matches $< requesting\_principal, integrity >$ of the requested object **and** `freelist` is not empty **then**
2: **return** the first available object in the `freelist`
3: **end if**
4: Try to find a `partial_slab` with the matching label
5: **if** `partial_slab` found **then**
6: Activate this `partial_slab` as the current `cpu_slab`
7: **return** the first available object in the `freelist`
8: **else**
9: Allocate and initialize a `new_slab` from the page frame allocator
10: Associate label <requesting_principal, integrity> to the slab's `page` struct
11: Issue a `hypercall` to SILVER's hypervisor to label the corresponding physical pages and set up permissions in principals' HAP tables
12: Activate this `new_slab` and **return** object as of Line 6-7
13: **end if**

nication. In transfer-based communication, a principal p sends one of its own data object d to another principal q. After that, d will become a (low-integrity) data object of S_q, and can no longer be accessed by p.

In SILVER's implementation, The data object transfer is conducted by the memory allocator by moving data object from one slab to another. In this case, principal p will invoke the API call `pd_transfer_object`, providing its object and q's principal id as input. The memory allocator locates the particular slab (label: $< p, high/low >$) that contains d, removing d from that slab, and copying d to a slab with the label $< q, low >$ of the same SLUB cache. The API call will return a new object reference which p could pass to q (but p can no longer dereference to d due to slab access control). Upon receiving the reference, q will leverage SILVER's reference validation primitives (described in Section 5.3.3.3) to ensure that the reference is legal, and finally endorse d to complete the transfer. Noted that in transfer-based communication, since the object ownership is surrendered, the sending principal must release all the references to the object before calling the `pd_transfer_object`, the same way as it is calling the `kfree` function.

Service-based communication represents the semantic that a principal requests another principal to process its data object, rather than giving up the ownership

permanently. This kind of communication is mostly carried out by cross-domain calls, and it needs to be achieved in a different way other than two back-and-forth transfer-based communication. The primary difference is that in service-based communication, the original stored location of the data object is not released during the transfer process, instead, a shadow copy of the object is created to be used by the domain that provides the service. After the service call is completed, the updated value of the object is copied back to the original location.

SILVER also implements service-based communication based on the SLUB allocator: when a principal p is requesting another principal q to process its own object d, SILVER will first *fork* object d from its current slab to a new object d^* in a $< q, low >$ slab in the same SLUB cache, and then use the reference of the forked object as the parameter of the cross-domain call. Before the call returns, all the references of d in S_p would dereference to the original d in p's slab. Once the call returns, SILVER will *join* the d^* with d if d^* can be endorsed, committing changes made by q, and free d^* from q's slab. Figure 5.3 shows the procedure of the corresponding slab operations. By proxying data during cross-domain calls, service-based communication not only improves security of data exchange, but also provides guarantee of consistency, since changes would not committed to the protection domain until the call is finished and updated data objects are endorsed. We describe how to adapt existing programs to leverage the two communication primitives in Section 5.4.2.

Noted that in most cases there is no extra hypervisor operation involved during the communication procedure, since both two slabs are pre-allocated so that no labeling/relabeling is required.

5.3.3.3 Reference validation and object accounting

In commodity OS kernel like Linux, fetching data from another principal is usually achieved by obtaining a reference (i.e., pointer of virtual address) to the particular data object. Object references can be passed between principals through function call parameters, function call return values, and reading exported symbols.

As stated in Section 5.2.1, the absence of reference validation in function parameters could leave avenues for attackers. In order to support reference valida-

tion, SILVER must be able to track security information of kernel data objects at run-time so that given any reference, SILVER could identify the object that the reference points to. To further support type-enforcement and bound checking, the type and size information of protected objects must also be known at run-time. By extending the SLUB tracking mechanism, we implemented an accountable resource management layer named object registry, for managing protected objects. The object registry maintains additional metadata for each protected object, and updates metadata upon allocation, deallocation, and communication events. The metadata include allocation principal, owner principal, object size, integrity level, object type and the time of allocation. The object type can be obtained because the SLUB allocator follows a type-based organization, and for generic-sized types, we use the allocation request function/location (the function that calls `kmalloc`) as well as the object size to identify the type of the object.

SILVER ensures that references passed through the `pd_transfer_object` API and service-based communication functions through designated parameters must be owned by the sender principal. In addition, the object registry offers basic primitives to principals for implementing their own reference validation schemes.

5.4 Evaluation

In this section, we first describe the implementation of our prototype, then we show how to apply SILVER to existing kernel programs for establishing protection domains. In Section 5.4.4, we demonstrate SILVER's protection effectiveness using security case studies of different kernel threats. We evaluate the performance of SILVER in Section 5.4.5.

5.4.1 Prototype Implementation

We have built a proof-of-concept prototype of SILVER. The VMM layer is an extension of the HUKO hypervisor [111], which is based on Xen 3.4.2 x86-64. The HAP table implementation requires hardware-virtualization features of processors, and our prototype is based on Intel's EPT (Extended Page Table) [39], which is

supported by most Intel processors in recent years. The low-level page labels are stored in unused bites of each EPT entry. We extended HUKO's mandatory protection states and added new labeling and control hypercalls to support VMM layer protection enforcement. We also added data marshaling mechanism in kernel function call mediation.

In SILVER, the operating systems are deployed as a Xen DomU in HVM mode. The OS subsystem of SILVER is developed to use with Linux kernel 2.6.24.6 on x86-64 architecture. Principal programs are loaded via the LKM (Loadable Kernel Module) interface, and we modified the `load_module` routine to register their program layout with the hypervisor and initialize the protection environment. Protection domain metadata are maintained in various locations. For each security principal we maintain a security identifier *prid* in the `module` struct, and we encode the slab label <principal, integrity> as additional flags in the corresponding `page_struct`. The object registry is organized in a search tree with the object address as the key value. In addition, to facilitate monitoring for the administrator, we export the run-time status of protection domains in the kernel, including object information and exported functions, to a virtual directory in the `/proc/` file system.

5.4.2 Protection Domain Deployment

In this section we describe how to adapt existing kernel programs to leverage primitives provided by SILVER. The first step is to establish the protection by declaring a specific LKM as a domain principal using the `pd_initialize()` routine, which will return an unique principal id. Entry points of this domain need to be initialized by `pd_ep_create` API.

The second step involves modifying the declaration or creation of security-sensitive program data. There are four kinds of data object associated with a kernel program: global object, stack object, heap object and page object. For static data and stack data, SILVER could automatically recognize them and treat them private to their principal so that modification by other principals must be carried out by calling wrapper functions. For heap and page objects, developers could specify their security property to control how they could be accessed by

other principals through calling `kmalloc_pd` and `__get_free_pages_pd` API with an integrity label. For example, unprotected memory sharing of low integrity data could be declared using the `GB_LOW` flag. Noted that this process could be performed *incrementally* and *selectively*.

The next step is to handle data communication. The major task is to convert functions that handle exchange of high-integrity data to exploit transfer-based and service-based communication primitives. The example code below is a fragment of `alloc_skb` function that returns an allocated network buffer to NIC driver using transfer-based communication. By adding five lines of code at the end of the function, the owner principal of the `sk_buff` object changes accordingly.

```
out:
-  return skb;
+  if(is_protected(prid = get_caller_prid()))
+    transfer_skb = pd_transfer_object(skb, prid, PD_HIGH, sizeof(struct
sk_buff));
+  else
+    transfer_skb = pd_degrade_object(skb, GB_LOW);
+  return transfer_skb;
```

Service-based communication is used in a similar manner, the data proxying is accomplished by SILVER automatically, but the developer needs to register the function signature and mark the transferring parameter at both the beginning and the end of function using SILVER's APIs. To support reference validation, SILVER provides routine that automatically checks whether a designated parameter reference belongs to the caller principal.

We have converted a number of Linux kernel functions and extensions using SILVER's primitive to secure their interactions. The extensions include the Realtek RTL-8139 NIC driver, the CAN BCM module, Media Independent Interface module, the emulated sound card driver, and two kernel modules written by us for attacking experiments. For all cases, the total amount of modification incurs changing less than 10% lines of original code.

5.4.3 Security Analysis

In this section, we evaluate the security of SILVER by performing security analysis to show how SILVER helps a properly-configured principal defend against various kinds of security threats from other untrusted principals in the kernel. Here we assume that q is the attacker principal, and q has the capability to communicate with the victim principal p.

We discuss each of the attacks as follows.

- **Directly modifying code/data.** q could conduct this attack by using either using store instructions or DMA writes via arbitrary addressing methods (physical or virtual address). The attack cannot be achieved since the code and all data object (except stack data) of p are only placed on physical pages that labeled p as the owner principal, and these pages are set to be non-writable in q's HAP table in the VMM. Also, the DMA writes are restricted into p's DMA zones by the IOMMU.

- **Control flow attacks.** q may attempt to divert the control flow to arbitrary positions in p's text by call/jmp/return instructions. However, according to Section 5.3.2, control transfers between protection domains are mediated by the hypervisor, which guarantees that entry points exported by p are only entrances to invoke p.

- **Stack manipulation.** As all kernel programs share the same kernel stack in the context of each user process, q may attempt to manipulate stack frames of other principals in the kernel. SILVER enforces that a principal's stack frame is read-only to other principals by employing a VMM-level private stack during the cross-domain calls (refer to Chapter 4 Section 4.3.4 for the detailed mechanism). Hence, any changes q made to p's stack frames are discarded during the protection domain switch so that the stack integrity of p is preserved.

- **Tampering architectural state.** As described in Section 5.3.2, the architectural state is preserved by the VMM during protection domain switches. As a result, q's attempt to manipulate p's architectural state (e.g., installing a

new page table) will be invalid at the time that protection domain is switched back to p.

- **Memory related exploits.** A typical attack path to commodity kernels is first exploiting a buffer/integer overflow vulnerability of a device driver function which accesses one of its objects on the slab. Then by overflowing that object, the attacker can also corrupt objects (e.g., function pointers) owned by the OS kernel or other kernel programs adjacently placed on the same slab.

  Although SILVER cannot eliminate the vulnerabilities caused by lack of bound/type checking, it can still mitigate the effect of these attacks. First, even if principal p's data are corrupted, it is almost infeasible to corrupt data object of any other principal q since in SILVER, one slab can only contain objects owned by one principal. As a result, the attacker can only compromise data objects owned by the kernel program that has the vulnerability. Moreover, damage of data corruption or code injection is still contained in the exact protection domain since the attacker can only exercise the privilege of the victim principal in terms of data access and control transfers.

  However, in the rare case that the vulnerability is in the OS kernel itself and exposed to attackers directly, SILVER cannot stop the kernel from being compromised since such compromised OS kernel could exercise its full privilege without violating access control policy of SILVER.

- **Exploit Communication.** A malicious principal q could exploit communication activities with principal p for privilege escalation. The attack leverages the fact that p and q may exchange data in certain legal channels (e.g., function parameters and shared memory) but q failed to validate its input. For example, p may need q to provide a callback function pointer or data reference but does not validate that the function or data is actually owned by q, which would cause confused deputy vulnerability as described in 5.2.1.

  SILVER does not offer direct input validation by itself, as program input depends too heavily on semantics of the program. Instead, SILVER helps programmers to achieve input validation and communication safety by pro-

viding them with two security features: 1) enforcing protection domain isolation: data object can only be owned by one principal at a time; 2) reference validation: given any reference, SILVER returns the owner principal and integrity level associated with the object that the reference actually referring to. These two features enables programs to detect forged references more conveniently and we demonstrated their applications for securing function parameters in cross-domain calls.

However, just enforcing reference validation on function parameters alone does not guarantee communication safety completely. Consider that q passes p a reference of its own struct-type data in function parameters, which is legitimate for function parameter validation. However, the struct contains a callback function pointer that was forged (not owned) by q, which would cause privilege escalation once got invoked by p. Besides the `struct` type, there are even more complicated types of data involved in communication, such as dynamic arrays, linked list and trees, and program semantics can be more complicated as well. In such cases, programmers would be required to provide validation specifications case by case, yet SILVER's primitives can still be served as basic building blocks for implementing complex input validation specifications.

A malicious principal could also attempt to subvert the OS subsystem of SILVER to disable or corrupt the protection domain mechanism itself. However, since the reference monitor is achieved at the VMM layer and the code and data of OS subsystem is also labeled by the VMM, any interactions between the OS subsystem and untrusted principals would be mediated by the hypervisor. The hypervisor enforces the data and control integrity requirements [111] to ensure that the OS subsystem can not be tampered.

5.4.4 Security Experiments

In this section we evaluate the effectiveness of security protection provided by SILVER mechanism with both real-world and synthetic attacks.

Kernel SLUB overflow. In Section 5.2.1, we mention an exploit described by

Jon Oberheide (CVE-2010-2959) to the vulnerable CAN Linux kernel module that achieves privilege escalation through overflowing dynamic data in the SLUB cache and corrupting crucial kernel control data in the same SLUB cache. We ported the vulnerable module to our Linux system, implemented and tested our exploit based on the attack code provided by Jon Oberheide. We then tested our attack in case the module is secured by SILVER's primitives. We designated the vulnerable CAN BCM module to be an untrusted principal and placed it into an untrusted protection domain. As result, dynamic data (e.g., `op->frames`) allocated by the CAN module are labeled with untrusted principal. According to SILVER's SLUB memory allocation scheme, these data object are placed on dedicated slabs for the untrusted CAN module principal, and they could never be adjacent to a high integrity kernel object `shmid_kernel` in the SLUB cache, despite any allocation pattern carried out by the attacker. For this reason, the attack can never succeed in our experiment.

Moreover, in case the attacker successfully compromise the vulnerable kernel module (e.g., be able to execute injected code), it still cannot tamper the integrity of OS kernel since the entire kernel module can only exercise permissions of an untrusted principal. In our experiment, we deliberately injected malicious code into the OS kernel, performing malicious activities such as modifying security-sensitive kernel data and calling functions that are not exported to the protection domain. All of these attack attempts raised security violation in SILVER's reference monitor and were therefore denied.

Kernel NULL pointer dereference. The key idea of NULL pointer dereference is to leverage the vulnerability that a kernel module does not check whether a function pointer is valid before invoking that function pointer. As the result, the control will jump to the page at address zero, where the attacker maps a payload page containing the malicious code from user space before hand. Once get executed, the payload code could modify crucial kernel data or invoke kernel functions to achieve malicious goals such as privilege escalation. Such vulnerabilities are quite common in buggy extensions and even the core kernel code (CVE-2009-2692, CVE-2010-3849, CVE-2010-4258).

In our experiment, dereferencing a NULL pointer in a buggy untrusted module

could not succeed in SILVER, primarily for two reasons. Firstly, in SILVER, executing user-level code by an untrusted principal is prohibited according to access control rules. This is because NX bits are set for user pages in the untrusted principal HAP table. Hence, even if the zero page is successfully mapped in user space, executing the payload code would still be blocked by the reference monitor with an execution exception. Secondly, even if the attack code got executed, it is still executed on behalf of untrusted principal with restricted permissions. As a result, attacking efforts such as privilege escalation (e.g., setting the `task->uid`, calling the `commit_creds` function) would be intercepted by the reference monitor and the integrity of core OS kernel is preserved.

Attacks through Kernel API. In Section 5.2.1, we show that even with protection schemes like memory isolation or SFI, attackers can still compromise kernel integrity by launching confuse-of-deputy attacks over legitimated kernel APIs. Noted that this kind of attacks is very rare in practice, for the reason that currently few Linux systems employ protection/sandboxing approaches inside OS kernel so that kernel attackers do not need to resort to this approach at all. To demonstrate SILVER's protection effectiveness against kernel API attacks, we implemented a kernel API attack module based on the RTL-8139 NIC driver. The attacking module provides a crafted reference of `struct pci_dev *` and uses it as input to the exported routine `pci_enable_device`. The reference is actually pointing to a calculated offset of the current process descriptor. By calling legitimate kernel API with such reference, the uid to current process will be set to 0 (root). SILVER prevents such attack by looking up the security property of the object referred by the actual pointer value. The reference monitor then detected that the caller principal actually does not owned the data object provided, and it raised an exception denying the attack attempt.

5.4.5 Performance Evaluation

In this section, we measure the performance overhead introduced by using SILVER's protection domain primitives. Our evaluation has three purposes: first, we would like to measure the time overhead of calling the extended or new APIs of SILVER by relying on a set of micro-benchmarks. Second, we would like to use

macro-benchmarks to measure the overall performance impact on throughput when a kernel NIC driver is protected. Finally, we would like to measure the memory overhead to see how much extra memory does SILVER consumed for protecting a typical kernel program. All experiments have been performed on a HP laptop computer with a 2.4GHz Intel i5-520M processor and 4GB of memory. The VMM layer is based on Xen 3.4.2 with a Linux 2.6.31 Dom0 kernel. The OS kernel environment was configured as a HVM guest running Ubuntu 8.04.4 (kernel version 2.6.24.6) with single core and 512MB memory.

Run-time performance. Table 5.1 reports the microbenchmark results of selected APIs of SILVER. The first four rows denote the performance of the native Linux kernel SLUB memory allocator running on unmodified Xen. The fast path happens when the object requested is exactly available at the current `cpu_slab`. The rest of rows shows the performance of SILVER's dynamic data management primitives. There are three major sources of overhead added by SILVER's run-time system: (1) "context switch" between protection domains, (2) labeling a physical page through hypercalls, and (3) updating the object registry and data marshaling. Row 5 and 6 show the overhead of allocation and free when the caller is kernel itself, which only incurs overhead caused by (3). Row 7-8 show the overhead of calling `kmalloc_pd` and `kfree` by protection domains other than kernel. In this case, besides overhead (3), a protection domain switch (1) is also involved, and page labeling (2) happens occasionally when a new slab is required. The relatively expensive guest-VMM switches in (1) and (2) make allocations/free operations by untrusted principals much more expensive. In future work, we plan to optimize the untrusted allocation performance by maintaining cached pool of labeled pages.

Table 5.2 demonstrates the performance cost of cross-domain activities in SILVER. These penalties do not exist in the native Linux kernel, as Linux kernel does not distinguish protection domains and it uses function calls and pointers for communication. Although it is difficult to do a fair comparison, in general the performance of communication primitives in SILVER prototype is also much slower than optimized IPC in micro-kernel systems (e.g., IDL [114] for stub code generating, L4Ka::Pistachio [65]) because of the cost of an additional hypervisor layer. We consider part of the extra performance penalties of SILVER as trade-

off for achieving compatibility with existing monolithic kernel, transparency to programmers and finer-grained control.

We summarize the decomposition of SILVER's overhead as follows:

- "Context switch" between protection domains. A protection domain switch is usually caused by an exception (e.g., cross-domain calls will trigger execution violation) that traps to the hypervisor. A guest-to-VMM switch involves costly VMEXIT operations, which save architecture context of the guest domain and prepare the context for the hypervisor. The hypervisor will go through its exception handling paths, and eventually determine that the exception is caused by a protection domain switch. Then it will load the HAP table for the other protection domain with necessary invalidation of TLB entries for the previous HAP table. Finally the hypervisor will return from the exception, invoking a VMM-to-guest switch with costly VMENTER operations. Protection domain switch happens every time when control transfers from one kernel program to another, and it could be very frequent in many cases. For example, OS kernel will invoke the NIC driver once it receives an interrupt from the NIC device or needs to send a packet.

- Labeling a physical page through hypercalls. This operation also requires hypervisor intervention by invoking VMEXIT and VMENTER instructions. However, labeling a physical page does not need to switch the HAP table or invalidate TLB entries, instead, it requires a page table walk and a HAP entry update. Labeling/relabeling a physical page happens less frequently during execution, usually only when a new slab is created or an old slab is destroyed. Hence, the actual number of labeling operations are far less than the number of allocation/free requests due to the slab cache mechanism.

- Data copying and metadata keeping. Data copy happens when relabeling of a data object occurs, and this is often associated with communication such as cross-domain calls. To relabel a data object, SILVER first needs to allocate a shadow object on the target slab matching the new label, then it copies the data object to the new slab, and finally releasing the old data object. In

the meanwhile, SILVER needs to keep track of the changes by maintaining meta on the corresponding slab as well as the object registry.

To perform evaluation on application performance, we use SILVER to contain a `8139too` NIC driver, and leverage secure communication primitives to protect *all* of its object creation and data exchanges with the Linux kernel. For each packet to process, `8139too` invokes the kernel routine `__alloc_skb` to create socket buffers and deliver to the networking stack, which happens very frequently. We use the following macro-benchmarks to evaluate performance impact of SILVER towards different applications: (a) Dhrystone 2 integer performance; (b) building a Linux 2.6.30 kernel with defconfig; (c) apache ab (5 concurrent client, 2000 requests of 8KB web page) and (d) `netperf` benchmark (`TCP_STREAM`, 32KB message size, transmit). Figure 5.4 illustrates the normalized performance results compared to native Linux on unmodified Xen. In our experiments, for CPU-intensive applications and I/O-intensive applications that do not use network, the overall performance is not much affected. However, in test cases that the network I/O is saturated, SILVER brings larger performance overhead in terms of throughput. This is primarily caused by very frequent protection domain switches and transfer-based communication. We measured protection domain switch rate of the `apache` test to be around $32,000$ per second. The overall performance also depends on how much data are specified as security-sensitive, how often security-sensitive data are created and the frequency of protected communication with untrusted principals. Fortunately, with SILVER, many of these security properties are controlled by the programmer so that she can manage the balance between security and performance. Hence, we expect SILVER to have better run-time performance in case of protecting only crucial data rather than the entire program. We also believe that our prototype can be greatly improved by optimizing Xen's VMEXIT and page fault exception handling to create a specialized path for SILVER's protection domain switch to avoid the unnecessary cost of VM switches.

Memory consumption. Compared with the original SLUB allocator, SILVER consumes more slabs (pages) for separating data objects of different principals and integrity levels. To measure the extra memory overhead, during our Apache

Linux (Xen)	`kmalloc` SLUB fast path	1.4μs
	`kmalloc` SLUB slow path	7.7μs
	`kfree` SLUB fast path	0.7μs
	`kfree` SLUB slow path	6.2μs
SILVER (called by kernel)	`kmalloc`	16.2μs
	`kfree`	14.4μs
SILVER (called by other principal)	`kmalloc_pd` average	56.7μs
	`kfree` average	64.1μs

Table 5.1. Micro-benchmarks results for dynamic data management APIs of SILVER, average of 1000 runs. The data object size of allocation is 192 bytes.

`pd_transfer_object` size: 1KB	51.9μs
service-based function forking data size:1KB	75.7μs
general protection domain switch	25.5μs

Table 5.2. Micro-benchmarks results for control transfer events in SILVER, average of 1000 runs.

`ab` test, we sampled the total number of slabs in use by the memory allocator for 20 times, and compared the average number of active slubs with the same measurement using the original SLUB allocator. It turned out to be that fully protecting `8139too` driver only takes 12 extra slabs (48KB of kernel memory) on average in our web server test. This is due to the short lifetime of dynamic objects and the on-demand allocation scheme which only creates a specific slab upon the first request of a corresponding object.

5.5 Limitations and Future Work

Our current prototype has several limitations. First, for a few functions, we found difficulties in directly applying service-based communication on them, as they move complex data structures across function calls instead of transferring a single data object. Dealing with these functions may require us to manually write data marshalling routines. Fortunately, most of these functions are provided by the OS kernel, which usually configures as the parent domain of the caller principal and

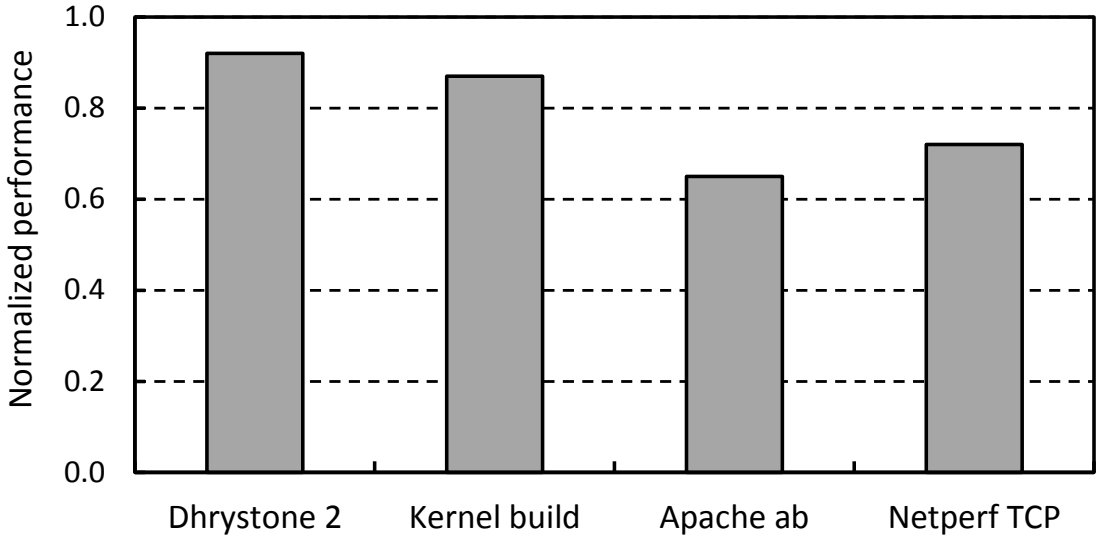

Figure 5.4. Application benchmark performance, normalized to native Linux/Xen.

can directly operates on these data structures without data marshalling.

Compared with language-based and other static isolation approaches, SILVER's run-time mechanism is more accurate in resource tracking than static inference. However, our approach also has shortcomings for not providing verification and automatic error detection to programmers. For example, programmers must pay extra attention for not creating dangling pointers when using object transfer and endorsement primitives of SILVER, since these operations will release the original object in the same way as `kfree` function. We plan to incorporate kernel reference counting [115] to help programmers manage their references of protection domain data objects. Moreover, adapting kernel programs to use SILVER requires certain understanding of security properties of their data and functions, and the entire procedure might be complex for converting very large programs. Hence, we also would like to explore automatic ways to transform an existing program to use SILVER given a security specification.

5.6 Summary

In this chapter, we have described the design, implementation and evaluation of SILVER, a framework to achieve transparent protection primitives that provide

fine-grained access control and secure interactions between OS kernel and untrusted extensions. We believe that SILVER is an effective approach towards controlled privilege separation, by which developers could protect their programs and mitigate the damage to OS kernel caused by attacks exploiting a vulnerability in untrusted extensions.

Chapter 6

Conclusion

In this thesis, we demonstrate technical approaches that help commodity computers preserve system integrity both proactively and reactively.

We have demonstrated that a computer system could leverage intrusion recovery techniques to preserve its system integrity in a reactive manner. We develop SHELF, a system that restores clean state for system objects after detected compromise. SHELF uses taint tracking to record object dependencies so that it could precisely restore benign state of infected objects to preserve business continuity and achieve recovery accuracy. Moreover, at the recovery stage, SHELF uses quarantine techniques to contain the infection so that uninfected objects can maintain their availability.

Motivated by the need of protecting the integrity of operating system kernels, we design and implement HUKO architecture to secure the execution of untrusted kernel extensions. With HUKO, untrusted extensions are transparently isolated from the OS kernel using memory virtualization techniques. Their interactions with the OS kernel are completely mediated and enforced by mandatory access control policies.

To help the OS kernel achieve better privilege separation and controlled communication, we design and implement SILVER, an architecture and a set of kernel-level primitives which offer a more general and fine-grained protection domain mechanism for principals in commodity kernel environment. In SILVER, security principals in the OS kernel can specify security properties of their data and communication with other principals. Compared to other approaches, SILVER does

not require shifting the programming paradigm or fundamental changes to the program structure.

In conclusion, these techniques provide commodity operating systems a better integrity guarantee on security-sensitive data of both users and the system, even in the presence of untrusted code and intrusions.

Bibliography

[1] Witchel, E., J. Rhee, and K. Asanović (2005) "Mondrix: memory isolation for linux using mondriaan memory protection," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, ACM, New York, NY, USA, pp. 31–44.

[2] Zeldovich, N., H. Kannan, M. Dalton, and C. Kozyrakis (2008) "Hardware Enforcement of Application Security Policies Using Tagged Memory." in *OSDI 2008*, USENIX Association, pp. 225–240.

[3] Efstathopoulos, P., M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris (2005) "Labels and Event Processes in the Asbestos Operating System," in *SOSP '05*.

[4] Zeldovich, N., S. Boyd-Wickizer, E. Kohler, and D. Mazières (2006) "Making Information Flow Explicit in HiStar," in *OSDI '06*.

[5] Fähndrich, M., M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi (2006) "Language support for fast and reliable message-based communication in singularity OS," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, ACM, New York, NY, USA, pp. 177–190.
URL `http://doi.acm.org/10.1145/1217935.1217953`

[6] Myers, A. C. and B. Liskov (1997) "A decentralized model for information flow control," in *Proceedings of the sixteenth ACM symposium on Operating Systems Principles*, SOSP '97, ACM, New York, NY, USA, pp. 129–142.
URL `http://doi.acm.org/10.1145/268998.266669`

[7] "Restoring Files and File Systems: ufsdump and ufsrestore," http://docs.sun.com/app/docs/doc/805-7228/6j6q7uf1k?a=view.

[8] "ThinkVantage Rescue and Recovery: One button recovery," http://www.pc.ibm.com/us/think/thinkvantagetech/rescuerecovery.html.

[9] "HP StorageWorks One-Button Disaster Recovery Solution," http://h18006.www1.hp.com/products/storageworks/drs/summary.html.

[10] Zhu, N. and T. cker Chiueh (2003) "Design, Implementation, and Evaluation of Repairable File Service," in *International Conference on Dependable Systems and Networks (DSN)*.

[11] Goel, A., K. Po, K. Farhadi, Z. Li, and E. de Lara (2005) "The Taser Intrusion Recovery System," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*.

[12] Liedtke, J. (1995) "On Micro-kernel Construction," in *SOSP '95*.

[13] Denning, D. E. (1976) "A lattice model of secure information flow," *Commun. ACM*, **19**(5), pp. 236–243.

[14] Bell, D. E. and L. J. LaPadula (1973) *Secure Computer Systems: Mathematical Foundations*, *Tech. Rep. MTR-2547*, The Mitre Corporation.

[15] Biba, K. J. (1977) *Integrity Considerations for Secure Computer Systems*, *Tech. Rep. MTR-3153*, The Mitre Corporation.

[16] Clark, D. D. and D. R. Wilson (1987) "A Comparison of Commercial and Military Computer Security Policies," *IEEE Symposium on Security and Privacy*.

[17] Fraser, T. (2000) "LOMAC: Low Water-Mark Integrity Protection for COTS Environments," in *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, IEEE Computer Society, Washington, DC, USA, p. 230.

[18] Abadi, M., M. Budiu, U. Erlingsson, and J. Ligatti (2005) "Control-flow Integrity," in *CCS '05*.

[19] Shacham, H. (2007) "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, ACM, New York, NY, USA, pp. 552–561.

[20] Saltzer, J. H. and M. D. Schroeder (1975) "The protection of information in computer systems," *Proceedings of the IEEE*, **63**(9), pp. 1278–1308.

[21] Department of Defense (1985) *Trusted Computer System Evaluation Criteria*, dOD 5200.28-STD (supersedes CSC-STD-001-83).

[22] Anderson, J. P. (1972) *Computer Security technology planning study*, *Tech. rep.*, Deputy for Command and Management System, USA.
URL `http://csrc.nist.gov/publications/history/ande72.pdf`

[23] Irvine, C. E. (1999) "The Reference Monitor Concept as a Unifying Principle in Computer Security Education," in *IN PROCEEDINGS OF THE IFIP TC11 WG 11.8 FIRST WORLD CONFERENCE ON INFORMATION SECURITY EDUCATION*, pp. 27–37.

[24] Jaeger, T. (2008) *Operating System Security*, Morgan and Claypool Publishers.

[25] Fraim, L. J. (1983) "Scomp: A Solution to the Multilevel Security Problem," *Computer*, **16**(7), pp. 26–34.

[26] "NSA. Security enhanced linux," http://www.nsa.gov/selinux/.

[27] Watson, R., W. Morrison, C. Vance, and B. Feldman (2003) "The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 285–296.

[28] Wright, C., C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman (2002) "Linux Security Modules: General Security Support for the Linux Kernel," in *Proceedings of the 11th USENIX Security Symposium*.

[29] Bomberger, A. C., W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro (1992) "The KeyKOS Nanokernel Architecture," in *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*.

[30] Shapiro, J. S., J. M. Smith, and D. J. Farber (1999) "EROS: a fast capability system," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, ACM, New York, NY, USA, pp. 170–185.
URL `http://doi.acm.org/10.1145/319151.319163`

[31] Watson, R. N. M., J. Anderson, B. Laurie, and K. Kennaway (2010) "Capsicum: Practical Capabilities for UNIX," in *USENIX Security'10*.

[32] Krohn, M., A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris (2007) "Information Flow Control for Standard OS Abstractions," in *SOSP '07*.

[33] Corbató, F. J. and V. A. Vyssotsky (1965) "Introduction and overview of the multics system," in *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, AFIPS '65 (Fall, part I), ACM, New York, NY, USA, pp. 185–196.

[34] "ptrace documentation," http://lwn.net/Articles/446593/.

[35] Goldberg, I., D. Wagner, R. Thomas, and E. A. Brewer (1996) "A secure environment for untrusted helper applications confining the Wily Hacker," in *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, USENIX Association, Berkeley, CA, USA, pp. 1–1.
URL `http://dl.acm.org/citation.cfm?id=1267569.1267570`

[36] Provos, N. (2003) "Improving host security with system call policies," in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, USENIX Association, Berkeley, CA, USA, pp. 18–18.
URL `http://dl.acm.org/citation.cfm?id=1251353.1251371`

[37] "Apparmor," http://www.novell.com/linux/security/apparmor/.

[38] "AMD-V Nested Paging," http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf.

[39] "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide," http://www.intel.com/Assets/PDF/manual/253669.pdf.

[40] Witchel, E., J. Cates, and K. Asanović (2002) "Mondrian memory protection," in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ACM, New York, NY, USA, pp. 304–316.

[41] Wahbe, R., S. Lucco, T. E. Anderson, and S. L. Graham (1993) "Efficient Software-based Fault Isolation," in *SOSP '93*.

[42] Erlingsson, U., M. Abadi, M. Vrable, M. Budiu, and G. C. Necula (2006) "XFI: software guards for system address spaces," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, USENIX Association, Berkeley, CA, USA, pp. 75–88.

[43] Castro, M., M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black (2009) "Fast Byte-granularity Software Fault Isolation," in *SOSP '09*.

[44] Yee, B., D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar (2009) "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," *Security and Privacy, IEEE Symposium on*, **0**, pp. 79–93.

[45] Mao, Y., H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek (2011) "Software fault isolation with API integrity and multi-principal modules," in *SOSP '11*.

[46] Myers, A. C. "JFlow: practical mostly-static information flow control," POPL '99.

[47] Mettler, A., D. Wagner, and T. Close "Joe-E: A Security-Oriented Subset of Java," in *NDSS '10*.

[48] Roy, I., D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel "Laminar: practical fine-grained decentralized information flow control," PLDI '09.

[49] Seshadri, A., M. Luk, N. Qu, and A. Perrig (2007) "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ACM, New York, NY, USA, pp. 335–350.

[50] Riley, R., X. Jiang, and D. Xu (2008) "Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing," in *RAID '08: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, Springer-Verlag, Berlin, Heidelberg, pp. 1–20.

[51] Litty, L., H. A. Lagar-Cavilla, and D. Lie (2008) "Hypervisor Support for Identifying Covertly Executing Binaries," in *SS'08: Proceedings of the 17th USENIX Security Symposium*, USENIX Association, Berkeley, CA, USA, pp. 243–258.

[52] Baliga, A., V. Ganapathy, and L. Iftode (2008) "Automatic Inference and Enforcement of Kernel Data Structure Invariants," in *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, IEEE Computer Society, Washington, DC, USA, pp. 77–86.

[53] Srivastava, A., I. Erete, and J. Giffin (2009) *Kernel Data Integrity Protection via Memory Access Control*, *Tech. Rep. GT-CS-09-04*, Georgia Institute of Technology.

[54] Wang, Z., X. Jiang, W. Cui, and P. Ning (2009) "Countering Kernel Rootkits with Lightweight Hook Protection," in *CCS '09: Proceedings of the*

16th ACM Conference on Computer and Communications Security, ACM, New York, NY, USA, pp. 545–554.

[55] Petroni, N. L., Jr. and M. Hicks (2007) "Automated Detection of Persistent Kernel Control-flow Attacks," in *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, ACM, New York, NY, USA, pp. 103–115.

[56] Wei, J., B. D. Payne, J. Giffin, and C. Pu (2008) "Soft-Timer Driven Transient Kernel Control Flow Attacks and Defense," in *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, IEEE Computer Society, Washington, DC, USA, pp. 97–107.

[57] Yin, H., D. Song, M. Egele, C. Kruegel, and E. Kirda (2007) "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, ACM, New York, NY, USA, pp. 116–127.

[58] Lanzi, A., M. Sharif, and W. Lee (2009) "K-Tracer: A System for Extracting Kernel Malware Behavior," in *Network and Distributed System Security Symposium.*

[59] Yin, H., Z. Liang, and D. Song. (2008) "HookFinder: Identifying and Understanding Malware Hooking Behaviors." in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08).*

[60] Wang, Z., X. Jiang, W. Cui, and X. Wang (2008) "Countering Persistent Kernel Rootkits through Systematic Hook Discovery," in *RAID '08: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, Springer-Verlag, Berlin, Heidelberg, pp. 21–38.

[61] Riley, R., X. Jiang, and D. Xu (2009) "Multi-aspect Profiling of Kernel Rootkit Behavior," in *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer systems*, ACM, New York, NY, USA, pp. 47–60.

[62] Swift, M. M., B. N. Bershad, and H. M. Levy (2003) "Improving the Reliability of Commodity Operating Systems," in *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, ACM, New York, NY, USA, pp. 207–222.

[63] Herder, J., H. Bos, B. Gras, P. Homburg, and A. Tanenbaum (2009) "Fault Isolation for Device Drivers," in *IEEE/IFIP International Conference on Dependable Systems and Networks, 2009. DSN '09.*

[64] Zhou, F., J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer (2006) "SafeDrive: safe and recoverable extensions using language-based techniques," in *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, USENIX Association, Berkeley, CA, USA, pp. 45–60.
URL `http://portal.acm.org/citation.cfm?id=1298455.1298461`

[65] "L4Ka::Pistachio," http://www.l4ka.org/65.php.

[66] Klein, G., K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood (2009) "seL4: Formal Verification of an OS Kernel," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ACM, New York, NY, USA, pp. 207–220.

[67] Chase, J. S., H. M. Levy, M. J. Feeley, and E. D. Lazowska (1994) "Sharing and Protection in a Single-Address-Space Operating System," *ACM Trans. Comput. Syst.*, **12**, pp. 271–307.

[68] Witchel, E., J. Rhee, and K. Asanović (2005) "Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection," in *SOSP '05*.

[69] Li, N., Z. Mao, and H. Chen (2007) "Usable Mandatory Integrity Protection for Operating Systems," in *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, IEEE Computer Society, Washington, DC, USA, pp. 164–178.

[70] Sharif, M. I., W. Lee, W. Cui, and A. Lanzi (2009) "Secure In-VM Monitoring using Hardware Virtualization," in *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, ACM, New York, NY, USA, pp. 477–487.

[71] Chen, X., T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports (2008) "Overshadow: a Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems," in *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, New York, NY, USA, pp. 2–13.

[72] Champagne, D. and R. B. Lee (2010) "Scalable Architectural Support for Trusted Software," in *The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Bangalore, India.

[73] McCune, J. M., Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig "TrustVisor: Efficient TCB Reduction and Attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*.

[74] Douceur, J. R., J. Elson, J. Howell, and J. R. Lorch (2008) "Leveraging Legacy Code to Deploy Desktop Applications on the Web," in *OSDI'08*.

[75] "Sandbox in Chrome," http://dev.chromium.org/developers/design-documents/sandbox.

[76] Wang, H. J., C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter (2009) "The Multi-principal OS Construction of the Gazelle Web Browser," in *USENIX Security '09*.

[77] "Android: Security and Permissions," http://developer.android.com/guide/topics/security/sec

[78] Chiueh, T.-c., G. Venkitachalam, and P. Pradhan (1999) "Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions," in *SOSP '99*.

[79] Ford, B. and R. Cox (2008) "Vx32: Lightweight User-level Sandboxing on the x86," in *USENIX ATC*.

[80] Garfinkel, T., B. Pfaff, and M. Rosenblum (2004) "Ostia: A Delegating Architecture for Secure System Call Interposition," in *NDSS'04*.

[81] Siefers, J., G. Tan, and G. Morrisett (2010) "Robusta: Taming the Native Beast of the JVM," in *CCS '10*.

[82] Prabhakaran, V., A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau (2005) "Analysis and evolution of journaling file systems," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 8–8.

[83] Srinivasan, S., C. Andrews, S. Kandula, and Y. Zhou (2004) "Flashback: A Light-weight Extension for Rollback and Deterministic Replay for Software Debugging," in *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, USENIX Association, pp. 3–3.

[84] Qin, F., J. Tucek, J. Sundaresan, and Y. Zhou (2005) "Rx: treating bugs as allergies—a safe method to survive software failures," *SIGOPS Oper. Syst. Rev.*, **39**(5), pp. 235–248.

[85] Osman, S., D. Subhraveti, G. Su, and J. Nieh (2002) "The Design and Implementation of Zap: A System for Migrating Computing Environments," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI).*

[86] Dunlap, G. W., S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen (2002) "ReVirt: enabling intrusion analysis through virtual-machine logging and replay," in *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation.*

[87] Konuru, R., H. Srinivasan, and J.-D. Choi (2000) "Deterministic Replay of Distributed Java Applications," *Parallel and Distributed Processing Symposium, International*, **0**, p. 219.

[88] Liu, P., P. Ammann, and S. Jajodia (2000) "Rewriting Histories: Recovering from Malicious Transactions," *Distrib. Parallel Databases*, **8**(1), pp. 7–40.

[89] Bhatkar, S., A. Chaturvedi, and R. Sekar (2006) "Dataflow Anomaly Detection," *IEEE Symposium on Security and Privacy,*, **0**, pp. 48–62.

[90] Gao, D., M. K. Reiter, and D. Song (2004) "Gray-box extraction of execution graphs for anomaly detection," in *Proc. of the 11th ACM conference on Computer and Communications Security*, pp. 318–329.

[91] Garfinkel, T. and M. Rosenblum (2003) "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. Network and Distributed Systems Security Symposium.*
URL `citeseer.ist.psu.edu/garfinkel03virtual.html`

[92] Jiang, X., X. Wang, and D. Xu (2007) "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *Proc. of the 14th ACM conference on Computer and Communications Security*, pp. 128–138.

[93] Payne, B. D., M. Carbone, M. Sharif, and W. Lee (2008) "Lares: An Architecture for Secure Active Monitoring Using Virtualization," in *Proc. of the 2008 IEEE Symposium on Security and Privacy*, pp. 233–247.

[94] Yin, H., D. Song, M. Egele, C. Kruegel, and E. Kirda (2007) "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proc. of the 14th ACM conference on Computer and Communications Security*, pp. 116–127.

[95] King, S. T. and P. M. Chen (2003) "Backtracking intrusions," in *Proc. of the nineteenth ACM symposium on Operating systems principles*, pp. 223–236.

[96] Newsome, J. and D. Song (2005) "Dynamic Taint Analysis for Automatic Detection and Analysis and Signature Generation of Exploits Commodity Software," in *Proc. of the Twelth Symposium on Network and Distributed Security (NDSS)*.

[97] Soules, C. A. N., G. R. Goodson, J. D. Strunk, and G. R. Ganger (2003) "Metadata Efficiency in Versioning File Systems," in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pp. 43–58.

[98] Hsu, F., H. Chen, T. Ristenpart, J. Li, and Z. Su (2006) "Back to the Future: A Framework for Automatic Malware Removal and System Repair," in *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, IEEE Computer Society, Washington, DC, USA, pp. 257–268.

[99] Jain, S., F. Shafique, V. Djeric, and A. Goel (2008) "Application-level isolation and recovery with solitude," *SIGOPS Oper. Syst. Rev.*, **42**(4), pp. 95–107.

[100] "The User-Mode Linux Kernel Homepage," http://user-mode-linux.sourceforge.net/.

[101] Chen, P. M. and B. D. Noble (2001) "When Virtual Is Better Than Real," in *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, IEEE Computer Society, p. 133.

[102] "Detecting and Crashing UML Honeypots," http://quuxlabs.com/ gerard/pub/hack-lu-uml.pdf.

[103] Hund, R., T. Holz, and F. Freiling (2009) "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms," in *Security'09: Proceedings of the 18th USENIX Security Symposium*.

[104] Carbone, M., W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang (2009) "Mapping Kernel Objects to Enable Systematic Integrity Checking," in *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, ACM, New York, NY, USA, pp. 555–565.

[105] Dolan-Gavitt, B., A. Srivastava, P. Traynor, and J. Giffin (2009) "Robust Signatures for Kernel Data Structures," in *CCS '09: Proceedings*

of the 16th ACM Conference on Computer and Communications Security, ACM, New York, NY, USA, pp. 566–577.

[106] "Intel Virtualization Technology for Directed I/O," ftp://download.intel.com/technology/computing /vptech/Intel(r)_VT_for_Direct_IO.pdf.

[107] "EnyeLKM," http://www.packetstormsecurity.com/UNIX/penetration /rootkits/enyelkm-1.3-no-objs.tar.gz.

[108] "All-root," http://packetstormsecurity.org/UNIX/penetration /rootkits/all-root.c.

[109] "Unixbench," http://ftp.tux.org/pub/benchmarks/System/unixbench/.

[110] "LMbench," http://www.bitmover.com/lmbench/.

[111] Xiong, X., D. Tian, and P. Liu. (2011) "Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions." in *NDSS'11*.

[112] "Linux Kernel CAN SLUB Overflow," http://jon.oberheide.org/blog/2010/09/10/linux-kernel-can-slub-overflow/.

[113] "The SLUB allocator," http://lwn.net/Articles/229984/.

[114] Haeberlen, A., J. Liedtke, Y. Park, L. Reuther, and V. Uhlig (2000) "Stub-code Performance is Becoming Important," in *Proceedings of the 1st conference on Industrial Experiences with Systems Software - Volume 1*, WIESS'00, USENIX Association, Berkeley, CA, USA, pp. 4–4.

[115] McKenney, P. E. (2007) *Overview of Linux-Kernel Reference Counting*, *Tech. Rep. N2167=07-0027*, IBM Beaverton.

Vita

Xi Xiong

Xi Xiong was born in China. He received his BS degree in Computer Science and Engineering from University of Science and Technology of China in 2007. He joined the Ph.D. program of Department of Computer Science and Engineering at Pennsylvania State University in 2007. He is currently a Ph.D. candidate at the Cyber Security Lab under the supervision of Professor Peng Liu. His research interests include various topics in operating system security, sandboxing techniques, intrusion analysis and recovery.

He is a student member of ACM and ACM SIGSAC.