January 2015

Securing Virtualized System via Active Protection

Zhongshu Gu
Purdue University

Graduate School Form 30
Updated 1/15/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By GU,ZHONGSHU

Entitled
Securing Virtualized System via Active Protection

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Dr. Dongyan Xu
Chair

Dr. Xiangyu Zhang

Dr. Ninghui Li

Dr. Sonia Fahmy

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Dr. Dongyan Xu

Approved by: Dr. Sunil Prabhakar/Dr. William J Gorman 7/27/2015

Head of the Departmental Graduate Program Date

SECURING VIRTUALIZED SYSTEM VIA ACTIVE PROTECTION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Zhongshu Gu

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2015

Purdue University

West Lafayette, Indiana

To my parents, my wife, and my son.

ACKNOWLEDGMENTS

First of all, I would like to express my deepest gratitude to my advisor, Professor Dongyan Xu. Thank him for the invaluable guidance, caring, encouragement, and support for the last five and half years. He is the most influential and most important person who introduced me into computer science research and teach me how to become a good researcher in all aspects. He always encourage me to explore new research areas and give me the freedom to propose new ideas. From him, I learned how to initiate an idea, how to differentiate it from existing research efforts, how to evaluate the merits of other researchers' work, how to structure a paper, and how to deliver your research ideas to the audience in the most efficient way.

Secondly, I would like to express my sincere thanks to Professor Xiangyu Zhang. We have been working closely since my second year at Purdue. He greatly broaden my view on the system security with his expertise on program analysis. I enjoy the time discussing research ideas with him. He can always give innovative inspirations and provide constructive suggestions.

I would like to extend my thanks to Professor Ninghui Li and Professor Sonia Fahmy for serving in my final examining committee, and Professor Kihong Park for serving in my prelim committee. Their suggestions on my dissertation are very valuable and important to me. I would also like to thank Dr. Peng Xie of Intelligent Automation Inc. I really enjoyed the time of being an intern at Rockville in the summer of 2011.

I have been fortunate to work with many brilliant people from our research group. First I would like to thank our Lab FRIENDS alumni Junghwan Rhee, Zhiqiang Lin, Ardalan Kangarlou, and Sahan Gamage for their invaluable advice and suggestions on my research. I would also like to thank Zhui Deng, Brendan Saltaformaggio, Yonghwi Kwon, Chunghwan Kim, Kexin Pei, Qifan Wang, Cheng Cheng, Cong Xu, Hui Lu, Rohit Bhatia, and Kyriakos Ispoglou. I will never forget the time we work together at Purdue.

Finally, I would like to thank my parents, Tong Gu and Lihui Shao, for their understanding, sacrifice, and support for me. I feel owing more gratitude to them this year when I become a Father myself. I would also like to express my deepest love to my wife, Jiaxi Wu. I cannot imagine I could finish this dissertation without her standing behind me. In addition, I would also like to thank my son, Christopher Xinyuan Gu (a.k.a. DanDan), who is taking an afternoon nap right now when I am writing this down.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Gu, Zhongshu PhD, Purdue University, August 2015. Securing Virtualized System via Active Protection. Major Professor: Dongyan Xu.

Virtualization is the predominant enabling technology of current cloud infrastructures and brings unique security benefits. Traditionally, researchers strive to include security components, such as intrusion detection, malware analysis, and integrity check, into underlying hypervisors. These hypervisor-based security approaches conduct only passive monitoring on the guest systems, but lack active protection mechanisms, i.e., patching the system vulnerabilities, eliminating the malicious logic, and shrinking the kernel attack surface, etc.

In order to achieve the security goals that are missing in existing hypervisor-based research efforts, we aim to expand the reach of the hypervisor to support active protection mechanisms. In this dissertation, we present a hypervisor-based security framework that consists of three key components, PROCESS-IMPLANTING, DRIP, and FACE-CHANGE to provide active protection at the level of user processes, kernel drivers, and OS kernels respectively, within guest virtual machines (VM). In particular, PROCESS-IMPLANTING enables on-demand implantation of general-purpose security tools directly from a hypervisor into a guest VM. The dynamic and stealthy nature of such security tools makes them harder to be predicted and detected by malicious adversaries. DRIP targets in-VM trojaned kernel drivers, which carry both benign and malicious logic. We conduct purification on such trojaned drivers to systematically deactivate the malicious logic and keep the benign logic intact. FACE-CHANGE minimizes the kernel attack surface within guest VMs at fine time-granularity. We achieve such kernel minimalism through dynamic switching of multiple application-specific minimized kernels at runtime.

From our evaluation results on both security and performance metrics, we demonstrate that PROCESS-IMPLANTING, DRIP, and FACE-CHANGE, can effectively provide active protection for the guest VM with minimum negative impact on the guest system execution. Furthermore, it is practical to deploy our security framework in the real-world cloud infrastructures considering its reasonable performance overhead.

1 INTRODUCTION

1.1 Dissertation Statement

Traditional computing systems are under increasing threats from advanced malware attacks. Emerging malware against operating systems (OS) exhibits more sophisticated strategies and behaviors. For example, recently advanced malware is able to actively detect, disable, or bypass security checks embedded in the operating system; kernel-level rootkits of Advanced Persistent Threats (APT) help hiding the presence of user-level malicious "accomplice" in the victim machines; trojaned kernel drivers can stealthily execute malicious functionalities under the cover of a benign cloak.

Virtualization has been widely adopted in the cloud computing infrastructures. It improves the utilization of limited hardware resources and facilitates central administrative tasks for managing guest virtual machines (VM). In addition, virtualization techniques hold unique advantages from security perspectives. The hypervisor (a.k.a. virtual machine monitor) provides an intermediate and confined computing environment to isolate the guest VM execution from underlying hardware. Assuming the guest system is compromised by some user-level malware or kernel-level rootkits, the trusted hypervisor is still able to enclose the malicious tampering within the virtual machine and avoid the harm to the host system.

Traditionally, security researchers retrofit underlying hypervisors to enable intrusion detection, malware analysis, and enforcement of kernel integrity within guest VMs, but leaving the guest systems "untouched" — the security tools resident at the hypervisor level only monitor the guest VM execution passively and trigger the alarm in the event of anomalous execution. But we expect that the hypervisor can achieve more than only passively monitoring the guest system. For example, if we have identified footprints of some malicious logic, is it possible to actively track its execution and eliminate the influence of

its malicious behavior? If we find that a guest system has some disclosed vulnerabilities that may be exploited in the future, is it possible to patch the system automatically to fix such loopholes? If we discover that a guest OS kernel expose larger-than-minimum kernel functionalities to applications running within a VM, is it possible to shrink this kernel to minimize its attack surface? These *active protection* mechanisms are missing in existing hypervisor-based research efforts. Compared to passive monitoring approaches, active protection mechanisms pose numerous technical challenges. For passive monitoring techniques, the execution of their hypervisor-based security components could be decoupled from the guest VM's execution. But achieving the active protection mechanisms mentioned above requires precise system interventions and "surgical" manipulations of the system state. We need to guarantee that we could safely and transparently operate on guest systems without breaking the consistency of their execution state.

1.2 Contributions

My research aims to expand the reach of the hypervisor and introduces active protection mechanisms into the hypervisor. At the same time, we guarantee that enabling hypervisor-based active protection components does not incur negative impacts on the execution of guest VMs.

The contributions of this dissertation can be summarized as follows:

- We present an integrated hypervisor-based security framework that consists of three key components, PROCESS-IMPLANTING [1], DRIP [2], and FACE-CHANGE [3]. Each component provides active protection to different system layers within the guest VM.

- PROCESS-IMPLANTING is an active VM introspection technique, which enables on-demand implantation of a general-purpose program directly from a hypervisor into a guest VM. The dynamic and stealthy nature of the *implanted processes* makes them harder to be predicted and detected by malicious adversaries. We also propose

various application scenarios of this technique in the areas of both security and cloud VM management.

- DRIP[1] is the first purification technique to systematically deactivate embedded malicious logic of trojaned kernel drivers. The purified kernel drivers could still preserve the benign logic and function the same as their original versions.

- FACE-CHANGE is a virtualization-based technique to shrink the kernel attack surface for applications running within the VM. Compared to existing system-wide kernel minimization approaches, FACE-CHANGE takes one step further to facilitate dynamic switching of multiple application-specific minimized kernels at runtime, thus enabling minimization of the kernel attack surface at finer time-granularity.

1.3 Overview of the Active Protection Framework

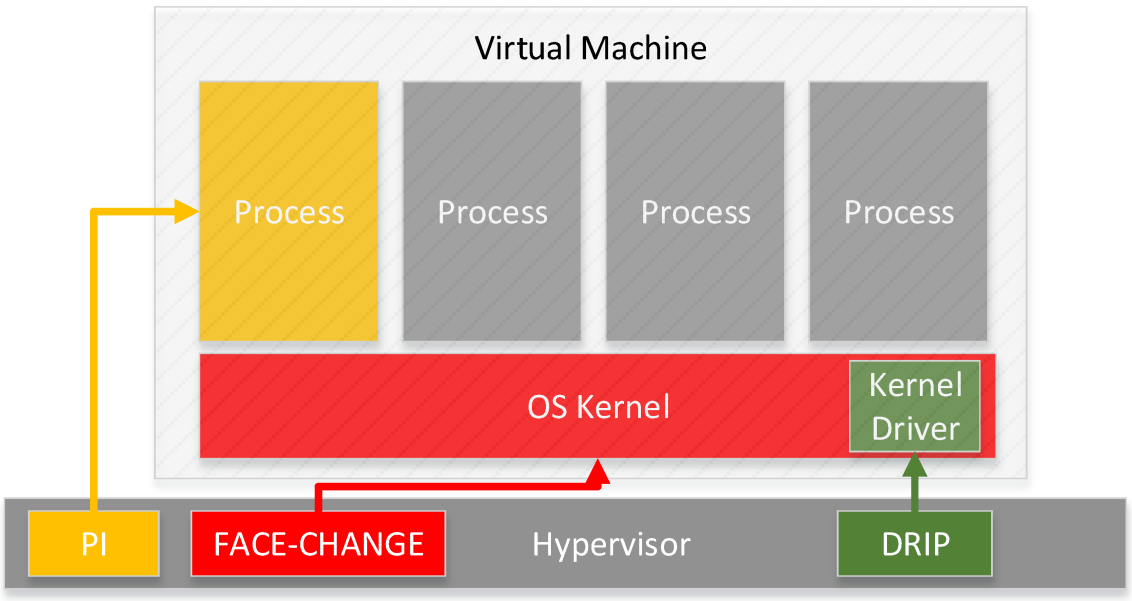

Figure 1.1.: Three key components of our hypervisor-based active protection framework

[1] DRIP stands for "DRIver Purifier"

Figure 1.1 illustrates the architectural overview of our security framework. Each component in our framework provides active protection to system entities at different layers within the guest VM. PROCESS-IMPLANTING (abbreviated as PI in Figure 1.1) manipulates the user process, DRIP targets the kernel drivers loaded in the kernel space, and FACE-CHANGE operates directly on the OS kernel. Below we give a brief introduction for each component.

1.3.1 PROCESS-IMPLANTING

The static nature of security tools in the system makes them exposed explicitly to malicious attacks. Adversaries could have enough time to study a system in advance, identify its vulnerabilities, and choose the right timing to drop their malware. When malware starts to execute, typically the first and most pivotal task is to deactivate security tools installed in the system to make itself under the radar.

In order to prevent security tools from becoming obvious targets of malicious attacks, PROCESS-IMPLANTING intends to make them dynamic and stealthy. Instead of statically installing security tools in the guest system, PROCESS-IMPLANTING randomly select a running process (denoted as a *victim process*) at runtime as camouflage, save its runtime state, and directly implant our security program (denoted as an *implanted process*) into guest VM from the hypervisor to reuse the *victim process* context. As such, the *implanted process* borrow some time slices from the *victim process* and execute the security logic under the cover of its running context. Furthermore, when the *implanted process* is scheduled out at context switch, PROCESS-IMPLANTING temporarily recover the memory mapping for this *victim process* to escape malicious memory inspections from other process. When the implanted security tool finishes its task, PROCESS-IMPLANTING could silently recover the execution of *victim process* from the checkpoint with no negative impact on the whole system.

From the view of an adversary, with PROCESS-IMPLANTING enabled, it becomes extremely difficult to pinpoint the security tool because: 1) The tool is dynamically implanted

at runtime and no installation footprints exist in the guest system. 2) The tool runs under the cover of a randomly-picked normal process without launching any suspicious new process.

1.3.2 DRIP

DRIP is an offline "purifier" targeting trojaned kernel drivers. Through investigation on the typical trojaned kernel drivers, we gain the observation: within trojaned kernel drivers, the malicious logic embedded inside them is typically orthogonal to its benign counterpart. In addition, malicious code need to invoke kernel APIs to perform critical system operations and access kernel data. Removing those invocations from the malicious code could neutralize the malicious behavior without affecting the driver's original functions.

Based on this key observation, we develop a testing-driven approach called DRIP to perform purification on trojaned kernel drivers. More specifically, we leverage application-level test suites to cover the benign functionalities of the trojaned kernel drivers. DRIP systematically derive a set of unneeded (i.e., the superset of malicious) kernel API invocations in the subject kernel driver, while ensuring the correct execution of all test cases. We could neutralize the hidden malicious logic by removing unnecessary kernel function invocations.

1.3.3 FACE-CHANGE

Kernel minimization has already been established as an effective approach to reducing the trusted computing base (TCB) of a system. In practice, even a highly specialized system usually involves multiple applications. Correspondingly, the minimized kernel for the system includes kernel code that is required by all these applications. However, we argue that such a system-wide minimized kernel is not good enough because it creates a larger-than-minimum attack surface. We observe that the kernel code required by different applications varies significantly. To be more specific, our experiments show that two distinct applications may share as little as 33.6% of their required kernel code, thus system-wide kernel minimization would over-approximate both applications' kernel requirements.

To address the problem, we develop the FACE-CHANGE component to support dynamic switching of multiple minimized kernels in the same VM, with each kernel customized for a specific application. We use the term *kernel view* to refer to the in-memory kernel code needed by an individual application. FACE-CHANGE presents each application process with a different, minimized *kernel view*, which is prepared individually in advance by profiling the application's kernel service needs. At runtime, upon a context switch, FACE-CHANGE dynamically switches to the kernel view of the process for the next time slice, achieving kernel minimalism with fine time-granularity. Compared with having only one system-wide minimized kernel, FACE-CHANGE could prevent more malware attacks because of the smaller attack surface it creates.

1.4 Dissertation Organization

This dissertation presents an integrated hypervisor-based security framework that consists of three key components: PROCESS-IMPLANTING, DRIP, and FACE-CHANGE. Each component address different research problems, but with the same research goal to provide active protection from the hypervisor to the guest VM.

Here we give an outline of this dissertation:

- Chapter 1 explains the unique advantages that can be brought by introducing active protection mechanism at the hypervisor level and research challenges that need to be addressed. Then we present the overview of our hypervisor-based active protection security framework. For each component within this framework, we demonstrate the research problems it targets and the fundamental principles behind the techniques respectively.

- Chapter 2 explains in more details about the motivation, design, implementation, and evaluation of PROCESS-IMPLANTING. We also propose the application scenarios of PROCESS-IMPLANTING in different areas.

- Chapter 3 focuses on the offline purification component DRIP. We investigate the unique properties of trojaned kernel drivers and explain the procedure of DRIP to deactivate the malicious logic within such drivers.

- Chapter 4 presents our investigations on the limitations of existing system-wide kernel minimization techniques and explains in detail how FACE-CHANGE can shrink the kernel attack surface further at finer time-granularity for each individual application.

- Chapter 5 discusses limitations of our current work and the future work we want to pursue.

- Chapter 6 describes representative research efforts that are closely related to this dissertation and compares our work with them.

- Chapter 7 concludes this dissertation.

2 PROCESS-IMPLANTING: PROCESS IMPLANTATION COMPONENT

2.1 Problem Statement

Security tools installed in the system to detect malware have become evident targets of cyber-attacks. Malicious adversaries can study the security tools for a long time before intruding into the system. To evade the detection, typically the first post-intrusion attack launched by malware is to deactivate security tools, such as the anti-malware engine. In order to be tamper-resistant to such attacks, existing research efforts leverage virtualization technology to relocate in-VM security tools to the hypervisor. But viewing from the hypervisor, the whole VM is a blackbox, i.e., only byte-level execution information is exposed. To monitor the execution of the guest system from the hypervisor, we need to reconstruct the guest VM's semantic view by using the virtual machine introspection (VMI) technique. But the semantic gap [4] between the guest VM and hypervisor hinders us from obtaining a complete semantic-rich view. Furthermore, currently hardware virtualization has become the mainstream technique adopted in cloud infrastructures. Hardware virtualization is designed to run most native instructions directly on the CPU and expose as few details as possible to the hypervisor to gain higher performance. This makes the semantic gap problem more challenging. Previous VMI approaches can only passively detect [5–8] or monitor attacks [9–11], but we expect to provide a general-purpose active protection solution, for example, to deactivate malicious logic, track anomalous execution, or patch disclosed vulnerabilities.

In this chapter, we present PROCESS-IMPLANTING, an active introspection component in our hypervisor-based security framework. The key idea is to implant a process directly from the hypervisor into the guest VM to bridge the semantic gap. The *implanted process* runs under the cover of a running in-VM *victim process* and could gain in-context execution environment of the running VM. We also design a series of coordination and protection

mechanisms supported by the higher-privileged hypervisor to exempt the *implanted process* from being tampered by the malware. Furthermore, after the *implanted process* exits, it leaves no negative impact on the normal execution of the guest OS and the applications.

The rest of this chapter is organized as follows. Section 2.2 proposes application scenarios for the PROCESS-IMPLANTING. Section 2.3 provides the detailed design of this component and how it satisfies the security requirements. Section 2.4 describes the implementation of our prototype. Section 2.5 evaluates performance and gives some representative application cases in the areas of both security and cloud VM management. We summarize PROCESS-IMPLANTING in Section 2.6.

2.2 Application Scenarios

There are several security implications of our PROCESS-IMPLANTING technique, as illustrated in the following application scenarios:

2.2.1 Stealthy Tracking

We can leverage PROCESS-IMPLANTING to reveal the evidence of attack provenances in a stealthy way. For example, tracing is a specialized use of logging to record the execution of a program for the purpose of monitoring and debugging. Tools such as *ltrace* and *strace* are widely used to monitor signals and library/system calls issued by a specific process during runtime. These tools get the in-context semantic-rich tracing information by executing inside the guest VM. But they are also vulnerable to attacks from malicious adversaries. If a process running inside the virtual machine presents some suspicious behaviors, we can implant the tracing tool into the guest VM and attach it to this process to gain more detailed evidence of its malicious operations. The result of tracing can be sent from guest VM to the hypervisor directly through hypercall. The host-based auditing system analyzing the logs sent from the implanted tracer can identify malwares more accurately.

2.2.2 System Recovery/Patching

If the system has already been compromised by the malware, PROCESS-IMPLANTING can be utilized to recover the system to its normal state by removing affected files, quarantining suspicious malware executables, and restarting security services that have been disabled. If any critical security vulnerability of guest VM is disclosed, the cloud provider should be responsible to patch applications or a kernel of guest VM to enforce security policies.

2.2.3 Performance Monitoring/Tuning

Performance monitoring of virtual machine is not that intuitive because there is an extra layer between the guest VM and the hardware. The hypervisor only has the system-wide performance view. When fine-grained monitoring is needed, performance data can be collected by using PROCESS-IMPLANTING to inject an agent into the guest VM to collect the performance data. The cloud administrator can conduct central management of the implanting procedure to perform large-scale performance analysis.

2.3 System Overview

2.3.1 Security Requirements

For in-box approaches to detecting and neutralizing malware, the anti-malware security tools are explicitly visible to the attacker. It is not difficult for the malware to identify the process that belongs to the security tools. After identification, the most common attacking technique of malware is to deactivate its opponent to prevent it from conducting scanning and detecting.

In-box security tools typically run at the same privilege level as the malware. Thus they have no advantage over the malware running within the same system. Even if they can elevate to root, some malware can achieve the same privilege escalation by exploiting some system vulnerability.

The out-of-box approaches address the problem by relocating the security tools out to the hypervisor to gain a higher-than-root privilege, but unfortunately at the same time they lose the in-context semantic-rich view as a trade-off and need to rebuild the status of the guest VM from byte-level information.

If we intend to implant the process back into the guest operating system, we need to fulfill the security requirements to make sure that this *implanted process* is protected, hard to be detected, and tamper-resistant to in-VM attacks. Otherwise, it has no advantage over the traditional in-box approaches.

Considering the *implanted process* is still running upon the guest OS, nevertheless it has some interactions with other components and relies on some services offered by the guest OS. We do not want to add too many constraints on the coding standard for the *implanted process* to make it totally isolated from the environment. The reason is that we want to reuse existing security tools as *implanted process* with only minor modifications. Thus we make a security assumption that the integrity of guest kernel is not in a compromised state during implanting. The techniques like NICKLE [12] and HookSafe [13] can be leveraged to maintain the kernel integrity when *implanted process* is running.

We state the security requirements from four aspects, stealthiness, isolation, robustness, and completeness.

Stealthiness: The *implanted process* should be hard to be predicted and detected by other processes in the guest VM.

Isolation: The *implanted process* should rely on as few services of guest OS as possible. Also it should have as few interactions with other process as possible. This can reduce the level of trustworthiness we demand on the guest OS and applications.

Robustness: The *implanted process* should not be terminated by other processes in the guest VM when it is running.

Completeness: When the *implanted process* finishes running or the hypervisor needs to call it back, it should exit gracefully without any impact on the stability of the guest VM.

2.3.2 Design

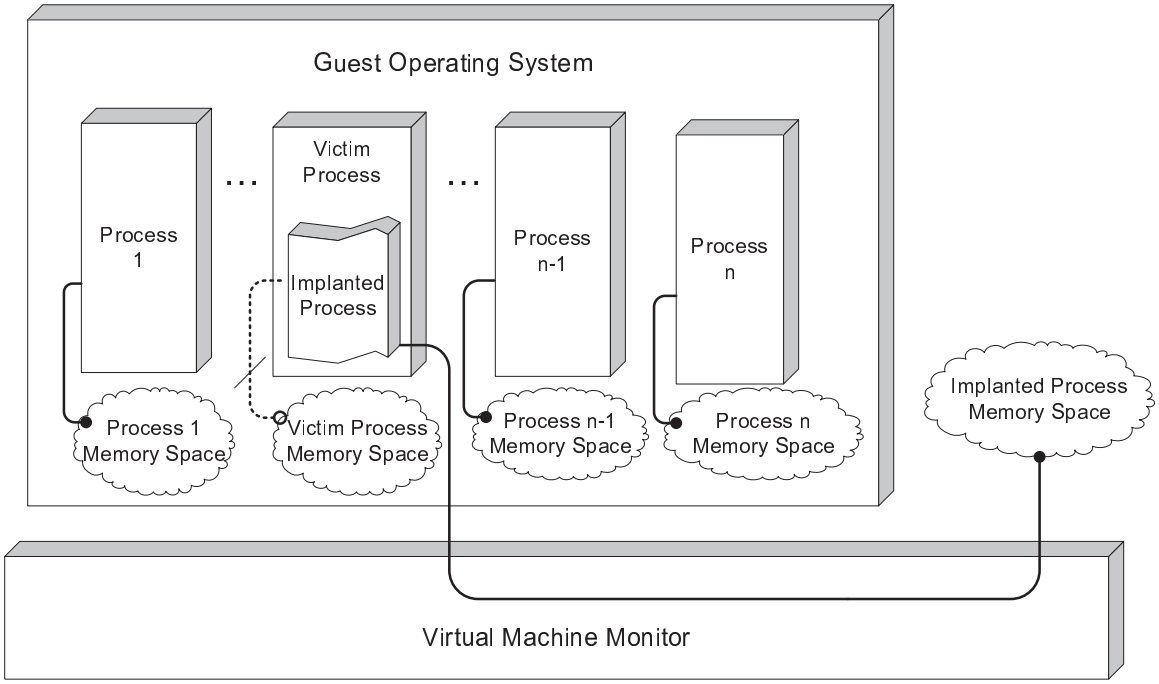

Figure 2.1.: Overall design of PROCESS-IMPLANTING framework

The key idea of PROCESS-IMPLANTING is to load the program from the hypervisor into the guest VM and run it under the camouflage of a running in-VM *victim process*. The administrator of the hypervisor can pick *implanted process* and *victim process* at runtime. We allocate memory regions at the hypervisor level separately for different segments of the *implanted process*. When the *victim process* is scheduled, we intercept the process context switch and trap to the hypervisor. We save current CPU context and replace it with the initial context of the *implanted process*, e.g., replace the instruction pointer with the entry address of the *implanted process*' binary and the stack pointer with the starting address of the stack. Then we modify the page table entries of the *victim process* to redirect it to the memory space where the *implanted process* is loaded. After re-entering the VM to continue the process context switch, the program counter returns back to the user space and begins to execute the code of the *implanted process*. When the *implanted process* finishes its task or the hypervisor needs to enforce mandatory restoration for the *victim process*, we

recover the *victim process* by restoring its saved context. Then the *victim process* could restart its execution from the checkpoint. From the view of the *victim process*, it freezes for a specific time window and some of its time slices are "borrowed" by the *implanted process*. Figure 2.1 gives an illustration of the overall design.

The challenges of this approach come from the needs to satisfy the security requirements proposed by us. We demonstrate the detailed design choices to fulfill these security requirements.

Random Selection of *Victim Process*

The static nature of traditional anti-malware software makes it easy to be targeted. The attacker can identify its existence by reading the software configuration from the OS. In order to be more stealthy, the administrator of hypervisor can randomly select the *victim process* and restart the *implanted process* by choosing another *victim process* at runtime. Such randomness eliminates the possibility that malware can locate its opponent only by querying the system.

Single Virtual CPU When *Implanted Process* is Running

If a guest VM owns multiple virtual CPUs, the *implanted process* running on one VCPU could be detected by a process running on another VCPU in parallel. In order to minimize the chance of being detected as an *implanted process*, we could disable other VCPUs temporarily when *implanted process* is running.

Camouflage of *Implanted Process*

The *implanted process* reuses the *victim process*' kernel-level data structures, e.g., process descriptor, page table, and heap, within the guest OS. From the view of the guest OS, it cannot identify the difference from the normal execution of the *victim process*. The *im-*

planted process just "borrow" a number of time slices from the *victim process* to run its own program and restore the *victim process*' execution state later.

In addition, *implanted process* is not backed by any file-based binary in the guest VM. If the malware tends to perform binary analysis on the executable, it can only find the binary of the *victim process* rather than the *implanted process*.

It can satisfy the security requirements of stealthiness with the camouflage of the *victim process*. Forking a new process is not permitted during the execution of the *implanted process* as it may leave obvious fingerprints, e.g., one more running process, in the guest VM. This could violate the stealthiness requirement.

Self-contained Executable of the *Implanted Process*

When compiling the *implanted process* executable, we choose to link the library routines statically to make it self-contained. Although it increases the size of the binary image, the *implanted process* does not need to rely on the library functions offered by the guest system. Otherwise, if the libraries in the guest VM are compromised, the results generated by the *implanted process* cannot be fully trusted. Then the assumptions of trust level have to be expanded. With such self-contained binary image, we only need to make an assumption that the operating system services used by the *implanted process* are trusted. This satisfies the security requirement of isolation.

Invisible Memory Space

We allocate three memory regions (code, data, and stack segment) for the *implanted process*. These memory regions are located at the hypervisor level and are beyond the physical memory range of the guest VM. The guest VM only checks the physical memory size at its booting time and indexes it into its kernel data structure of memory pages. Adding more memory to the guest VM during its runtime is similar to hot-plugging memory into a memory slot. The guest VM has no knowledge of the newly registered memory regions and

it does not access the memory address beyond its physical memory size. This satisfies the security requirement of stealthiness because of the transparency of these memory regions.

Timing of Implanting

The timing of implanting is critical to our system in order to satisfy the security requirement of completeness. From the view of guest system, the execution of *victim process* freezes when the implantation happens. Remember we choose to implant the process in the event of context switch and the next scheduled process is the *victim process*. However, as a side effect, part of the execution in the kernel space for *victim process* is lost. The reason is that when it enters back to the user space, the *implanted process* substitutes the *victim process* to execute. Considering that, we make three design decisions to address this problem. We first check if this context switch is triggered by the system call from the *victim process*. When we restore the *victim process* after the *implanted process* exits, we set the instruction pointer on the kernel stack backwards to restart this system call. Secondly, If the *victim process* is waiting for the resources and has already set its state as *uninterruptible*, we will enter the virtual machine silently without implanting. Thirdly, if this context switch is caused by a kernel preemption, we choose not to implant at this time.

Frequent Scene Restoration

We only permit the *implanted process* to read, write, and execute on its own memory regions and the other process should not have access to it. Information leakage should be prevented in the situation that the malware is capable of scanning the page table to detect the modification. We design a mechanism called *Frequent Scene Restoration* (FSR), to recover all the states we modified during implanting when the *implanted process* is scheduled out. After the context switch, the *victim process* is not modified from the guest view. Although it may incur some performance overhead, it satisfies the security requirement of stealthiness.

Checkpoint/Restart

Checkpoint/Restart is an optional design for some specific *implanted process*. As the user stack of *implanted process* is allocated independently, we can checkpoint the execution status by recording the register status. The *implanted process* can restart from this checkpoint and continue its execution at a later time when we want to implant it again. This is also designed to fulfill the security requirement of stealthiness.

Coordination Between the *Implanted Process* and the Hypervisor

The coordination mechanism between the *implanted process* and the hypervisor is designed for solving the concrete problem encountered when implanting some specific process. For example, the tracing program like *ltrace* and *strace* could attach to a running process and monitor its behavior. If we plan to execute mandatory restoration of *victim process*, it could make the process being traced behave incorrectly because the tracer has not detached from it. A similar problem arises for implanting a multi-threaded program. If all the child threads spawned by the *implanted process* are still running when the mandatory restoration happens, they could run on the address space of *victim process* after the restoration and this may cause serious errors. We design a coordination mechanism between the *implanted process* and the hypervisor to address these problems. A covert channel is created by setting a control bit on the argument part of the user stack. It can be read by the *implanted process* and written by the hypervisor. The *implanted process* should check it periodically. Instead of restoring the *victim process* immediately, the hypervisor sets this control bit to notify the *implanted process* that it could exit. When the *implanted process* read this bit, it should clean up, e.g., let all the child threads exit or detach from the process being traced, and then exit. The exit operation is intercepted by the hypervisor and we will discuss it in the next paragraph of *Graceful Exiting*.

The other coordination mechanism is that we modify the source code of existing tools to let them send the string pointer through hypercalls to the hypervisor instead of printing

on the console within the guest VM. The hypervisor can read this string by translating the guest virtual address to the host virtual address.

These coordination mechanisms promise the security requirements for completeness and stealthiness.

Graceful Exiting

When the *implanted process* is exiting, the guest OS will free the memory space of the *implanted process* we allocated at the hypervisor, and this will lead to a crash since the physical address of that memory space exceeds the maximum physical memory that the guest VM could access. This will break our security requirements of stealthiness and completeness and is undesirable. So instead of letting the process complete its exiting, we pause the exiting attempt and restore the *victim process* to maintain stealthiness and completeness. To perform the interception and restoration, the hypervisor needs to know exactly when the *implanted process* is going to exit. Although it is possible to modify the source code of the implanted program to let it inform the hypervisor actively, that would be inconvenient and not applicable to closed-source programs. Instead, we choose to set a trap through debug register for the exiting event of the *implanted process*, and the hypervisor is notified as soon as the trap is triggered.

Protection from the Hypervisor

The *implanted process* is not alone in the guest VM and is backed by the hypervisor. We can add protection to the *implanted process* from the hypervisor to satisfy the security requirement of robustness. Two mechanisms are designed in PROCESS-IMPLANTING to achieve this goal. First we elevate the privilege level to root by modifying the credential entry in the process descriptor. This has the same effect of switching user to root in the guest VM. With root privilege, the user-level malware is not capable to kill it by merely sending the terminating signal. It is also useful for some application scenarios because

monitoring or patching operations can only be done with the highest privilege in the guest VM.

If the malware also possess the same root privilege, it is still able to kill the *implanted process*. In order to strengthen its robustness, we design the second mechanism for more protection. We set the *unkillable* flag for this process and check its status for every context switch. The *unkillable* flag is used only by the *init* process in Linux to prevent it from being killed in any situation. We also utilize it to make our *implanted process* unkillable. If this bit is cleared by other process, we check it during every context switch to make sure that it has been set before running the *implanted process*.

Special Case: Multi-thread Program Implanting

Multi-threading is a widely-used programming paradigm nowadays and PROCESS-IMPLANTING supports multi-threaded applications to be practical in real-world scenarios. However, it needs special care for both selecting a multi-thread *victim process* and implanting a multi-threaded program.

To illustrate this problem, let us first take a close look at the scene of selecting a multi-thread *victim process*. When implanting happens, we choose a thread of the *victim process* to execute the *implanted process*. We denote this specific thread as *victim thread*, and other threads of the *victim process* as *innocent threads*. We modify the address space and the execution context of the *victim thread* to provide an execution environment for the *implanted process*. Note that such modification to the address space is shared among all threads of the *victim process*, but the modification to the execution context is only done to the *victim thread*. When those *innocent threads* begin to execute, inconsistency between the address space and their execution contexts may lead to a crash. There are two ways to address this problem, either by freezing all *innocent threads*, or by restoring the address space when there is a context switch to a *innocent thread*. We choose the latter one because it is more stealthy and easier to implement.

Algorithm 1 Multi-thread program implanting handling on context switch

```
1: procedure MULTITHREADIMPLANT(next)
2:     if next = victim and next.pid = next.tgid and imp = FALSE then
3:         IMPLANT()
4:         imp ← TRUE
5:         maxpid ← GET_MAXPID_IN_GROUP(next)
6:         vicpid ← next.pid
7:     else if imp = TRUE then
8:         ptype ← OTHER
9:         ntype ← OTHER
10:        if prev.tgid = vicpid then
11:            if prev.pid = prev.tgid then
12:                ptype ← VICTIM
13:            else if prev.pid ≤ maxpid then
14:                ptype ← INNOCENT
15:            else
16:                ptype ← IMPNEW
17:        if next.tgid = vicpid then
18:            if next.pid = next.tgid then
19:                ntype ← VICTIM
20:            else if next.pid ≤ maxpid then
21:                ntype ← INNOCENT
22:            else
23:                ntype ← IMPNEW
24:        if (ptype = VICTIM or ptype = IMPNEW) and (ntype = INNOCENT or ntype = OTHER) then
25:            RESTORE_SCENE()
26:        else if (ptype = INNOCENT or ptype = OTHER) and (ntype = VICTIM or ntype = IMPNEW) then
27:            LOAD_SCENE()
```

Implanting a multi-threaded program makes the scene more complex. The threads created by the *implanted process* require the address space for the *implanted process* while *innocent threads* of the *victim process* require the original address space of the *victim process*, so we have to switch address space when context switch happens between these two kinds of threads. However, there is no simple way to differentiate between these two kinds of threads because they belong to the same thread group. We use a technique here by leveraging the fact that if no *innocent thread* is created after implanting, then any thread created by the *implanted process* should have greater *pid* than any of the *innocent threads* (assuming the pid is in the same order of process creation time). Note that most programs only create threads in their main threads, so if we choose the main thread as the *victim thread*, the condition of the above fact is naturally fulfilled. In this way we could find the maximum *pid* of *innocent threads* before implanting and use it as a boundary between the two kinds of threads. Algorithm 1 demonstrates the detailed procedures we develop to handle multi-threaded program in PROCESS-IMPLANTING. We denote the previous task as $prev$, next task as $next$, type of previous task as $ptype$, type of next task type as $ntype$, *victim thread*'s *pid* as $vicpid$, process implanted flag imp and maximum *pid* in *victim thread group* as $maxpid$.

2.4 Implementation

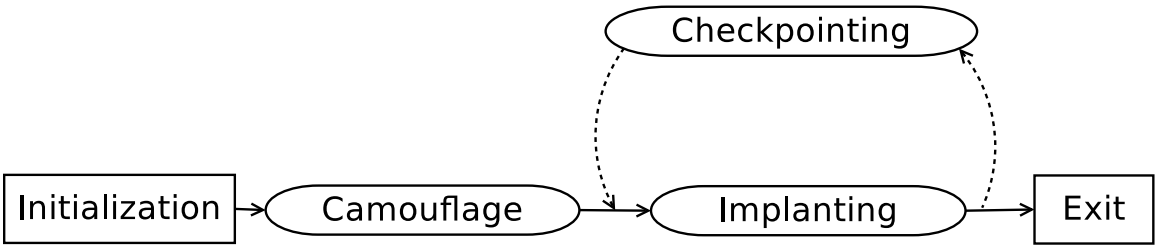

Figure 2.2.: Workflow of PROCESS-IMPLANTING

We have implemented a proof-of-concept PROCESS-IMPLANTING component as an extension of Kernel-based Virtual Machine [14] (KVM) hypervisor leveraging the Intel

virtualization technology [15]. The host OS is Ubuntu 10.04 32bit (Linux Kernel 2.6.32-23) distribution and the guest OS is Ubuntu 9.10 32bit (Linux Kernel 2.6.31-14) distribution.

The workflow can be divided into five phases: initialization, camouflage, implanting, checkpointing, and exit, as illustrated in Figure 2.2. The dotted line for the checkpointing phase means that it is optional. We explain each phase in detail below:

2.4.1 Initialization Phase

We choose Executable and Linkable Format (ELF) as the file format for the *implanted process* image. The binary image is compiled beforehand by statically linking all the library routines to make it self-contained. A program loader is implemented to load the code/data/stack segments into the memory allocated at the hypervisor level.

Then we register the memory slots in KVM for these three memory segments and assign them the guest physical addresses which are beyond the boundary of existing memory size of the guest VM. These memory segments are transparent to the guest OS as it has no knowledge that they are "hot-plugged" into the slots during the runtime and the guest VM only calculates the memory pages at its booting time. Only the *implanted process* can access these parts of memory in the following phases.

2.4.2 Camouflage Phase

In the camouflage phase, the *victim process* name can be determined at runtime. The name is written in the *victim process configuration* file, which is read by the hypervisor periodically. The guest virtual addresses of all exported kernel functions can be read from the *system map* of the guest kernel. *__switch_to* is the function responsible for process context switch in the Linux Kernel. After finding its entry address by searching the *system map*, we set debug register at this address in the guest kernel. Thus every context switch causes debug exception and can be captured by the hypervisor.

In our previous design, we considered the setting of a new *cr3* register as the symbol of context switch. This cannot fulfill the security requirements because if the thread scheduled

after the *implanted process* is a *kernel thread* or a *user thread* in the same thread group, it may reuse the previous *cr3* and no VM exit happens. If the subsequent thread is a *user thread*, it may crash because it would run on a wrong address space. If this thread is a *kernel thread* and is malicious, it could scan the page table of the previous thread to dump the code and data segment of the *implanted process*.

With VM exits intercepted at every context switch, if the previous thread is *implanted process*, we can restore both the page table and the modified entries in process descriptor before the execution of the next thread. Before implanting, we need to fill the upper part of the user stack by copying the content from the *victim process*' user stack. These are arguments, environments, and the auxiliary array, which are read by the *implanted process* during its loading time.

2.4.3 Implanting Phase

If the context switch happens and *victim process* is the next thread to be scheduled, VM traps to the hypervisor. All the user registers for the *victim process* are stored on the kernel stack. The steps of implanting are:

(i) Save user registers

(ii) Save the memory region descriptor's list

(iii) Save the original affected address mapping

(iv) Adjust physical page table of the *victim process* to point to implant process' memory space

(v) Update related entries in the shadow page table

(vi) Adjust the memory region descriptor's list to adapt to the new address space

(vii) Set user registers with the value of the *implanted process*

After completing seven steps above, when the guest VM resumes to guest mode, the *victim process* is completely replaced with the *implanted process*. The procedure of switching to address space of the *implanted process* is illustrate in Figure 2.3.

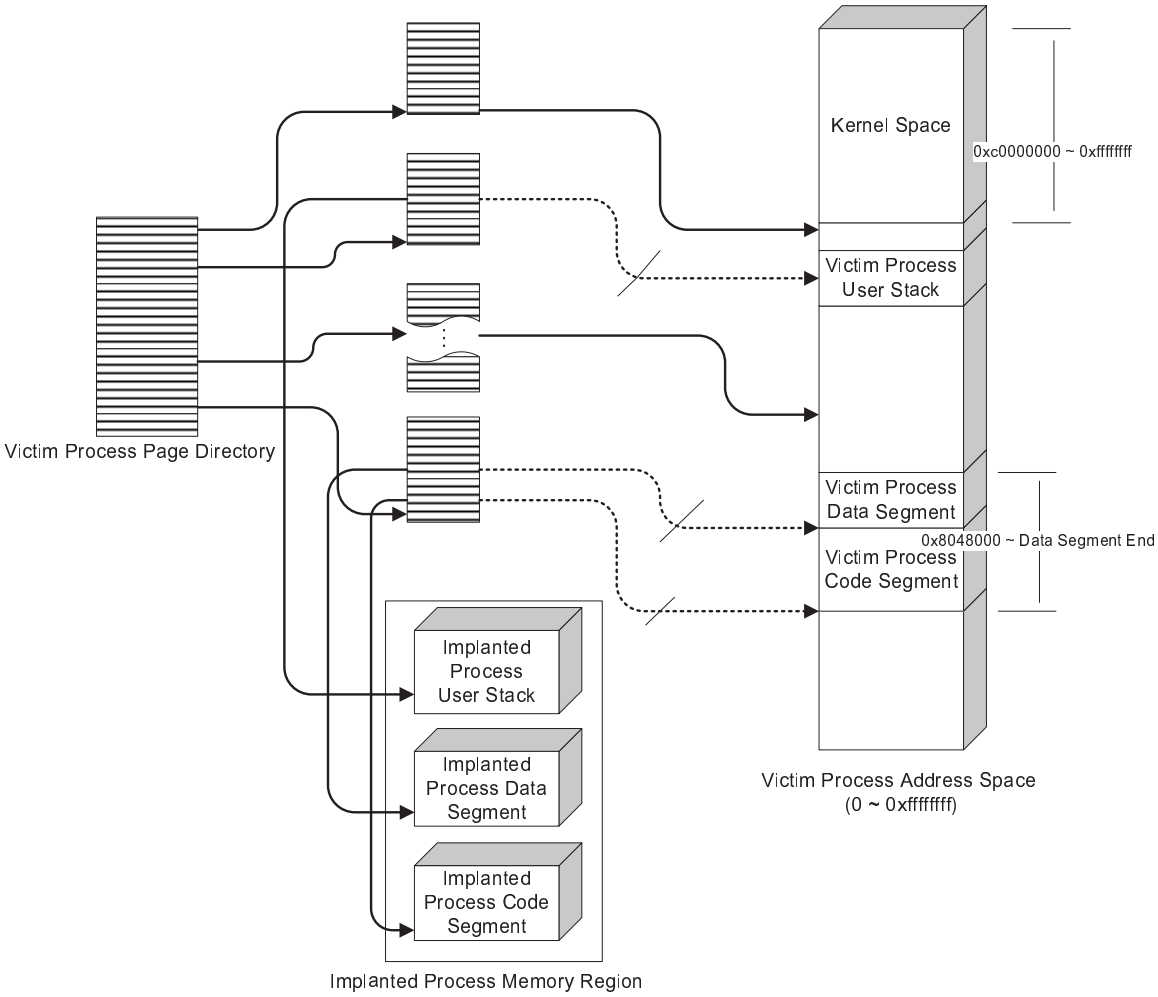

Figure 2.3.: Address space of *implanted process*

After implanting for the first time, when the *implanted process* is scheduled out, we restore *victim process*' physical page table and memory region descriptor list as mentioned in the design section about FSR. If the *implanted process* is scheduled again, we load the *implanted process*' physical page table and the memory region descriptor list correspondingly.

2.4.4 Checkpointing Phase

Checkpointing phase is optional and its main purpose is to raise the bar of stealthiness. The *implanted process* can be paused at a specific time by saving its execution state, i.e., user registers, the memory region descriptor list, and restore the execution of the *victim process*. The *implanted process* can restart at the checkpoint and continue execution after the *victim process* runs for several time slices.

2.4.5 Exit Phase

When the *implanted process* attempts to exit, we need to restore the *victim process*. The hypervisor intercepts the exiting attempt by setting traps on two system calls, *sys_exit* and *sys_exit_group*. All user mode processes in the Linux invoke either of these two system calls when they exit. We set two debug registers to the entry addresses of these two system calls. Such that any call or jump to them triggers a VM exit and can be captured by the hypervisor. When the hypervisor intercepts an exiting event and finds that the exiting process is the *implanted process*, it restores the *victim process*. Note that restoration here is slightly different from what we do in the checkpointing phase. In the checkpointing phase, the user mode registers saved in the kernel stack are restored directly when the kernel returns to the user mode. However, in the exit phase, the user mode *EAX* register is set to the return value of the *sys_exit* or *sys_exit_group* system call by the kernel. This is unexpected since the call was invoked by the *implanted process* but not the *victim process*, and the user mode *EAX* register of the *victim process* should not be tampered. To solve the problem, we set the kernel mode *EAX* register, which is used to store the return value of the system call, to the same value as the user mode *EAX* register of the *victim process*. In this way the user mode *EAX* register of the *victim process* would remain unmodified even if it is set to the return value of the system call. In addition, because the *sys_exit* or *sys_exit_group* function should not be actually executed, we set the instruction pointer and the stack pointer to the frame of the function's caller. From a user's view, the function returns without executing its code.

2.5 Evaluation

In this section, we evaluate our PROCESS-IMPLANTING component in three aspects. First, we show how our system satisfy the security requirements. Then we present some active introspection application cases. Finally we give the performance measurement results.

2.5.1 Security Evaluation

Experiment I: Scan the page table of the *victim processm*

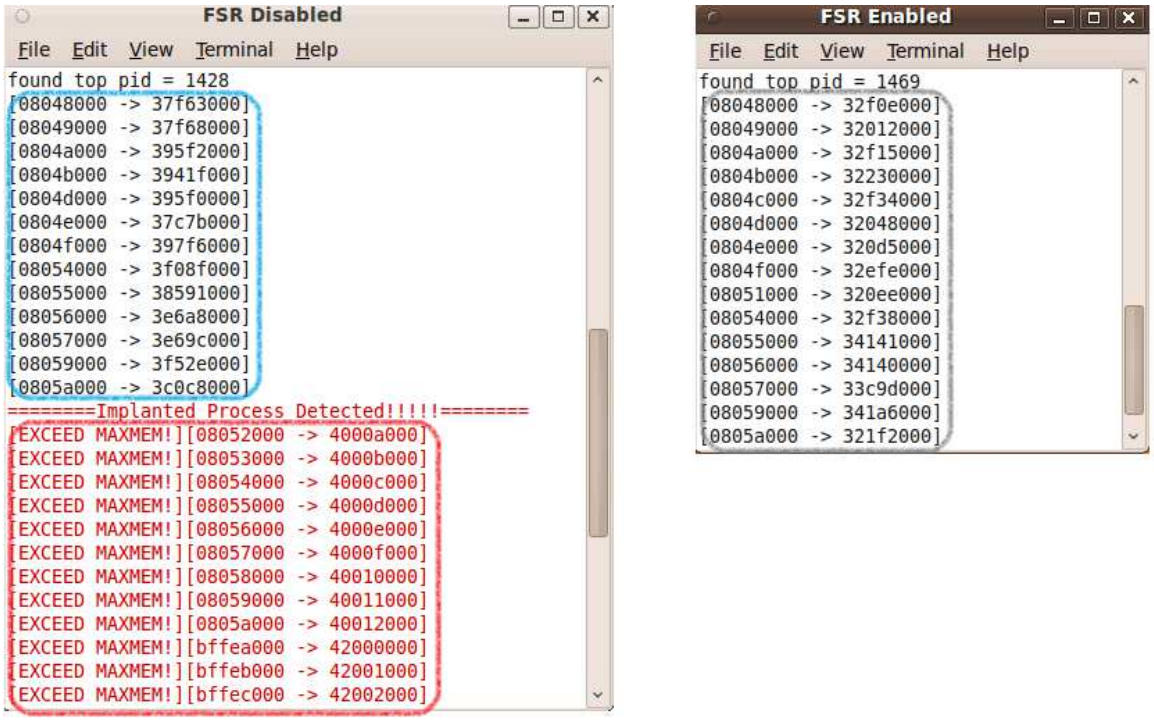

Figure 2.4.: Scan the page table of *victim process*

To demonstrate that even if the guest kernel is trusted, our mechanism of FSR is still crucial and necessary to maintain the stealthiness of the *implanted process*, we implemented a userspace program to simulate the potential attack by scanning the page table of the *victim process* repeatedly. Since Linux Kernel 2.6.25, there is a new feature that allows

userspace programs to read page tables of other processes by scanning */proc/pid/pagemap*. We assume that the attacker knows the design of PROCESS-IMPLANTING, the details of our implementation, and even the exact process chosen as the *victim process*.

The detection follows this observation: if a process is implanted, some of its virtual addresses are mapped to physical pages exceeding the maximum physical memory allocated for the virtual machine; for normal process, no such mapping exists. This is because the memory used to store code, data, and stack segments of the *implanted process* is allocated at the hypervisor level. Its memory address falls out of the border of the guest VM's memory. The scanner can claim that a process is implanted if it finds such suspicious page mappings in the page table of that *victim process*.

We perform the experiment with FSR disabled and enabled respectively. The comparison of the two results is shown in Figure 2.4. When FSR is disabled, before the implanting, the scanner found the original page mapping of the *victim process* as shown in the upper box of the left window. Then, right after the implanting, changes made to the page table and suspicious page mappings shown in the lower box of the left window are discovered. On the contrary, when FSR is enabled, the attacker could only find the original page mappings of the *victim process* shown in the right window during the whole experiment. This is because when the scanning process is running, the *implanted process* must have been scheduled out at an earlier time and we have already recovered all the things we modified during implanting at that time using FSR.

2.5.2 Active Introspection Case Studies

Ltrace is the tool to intercept and record the library/system calls and the signals of a specific process. It can attach to a process and monitor its behavior during runtime. We make two experiments to demonstrate using it as the *implanted process*.

Experiment II: Implanting ltrace to trace malware

In this experiment, we implant *ltrace* into the guest VM and leverage it to trace both the library and system calls of a real-world malware. The results are transferred to the hypervisor through hypercalls.

The right window in Figure 2.5 presents the malware whose name is *i-am-sick*. Its main function is to infect the files under the */tmp* directory by copying code into it and execute the infected file afterwards. With the implanted *ltrace*, we can attach to the malware at runtime and monitor the library calls and system calls. The left window in Figure 2.5 is the terminal of the KVM hypervisor. It receives logs directly from the implanted *ltrace*. After inspecting the log in this terminal, the execution path and malicious behavior of this malware can be easily identified.

Experiment III: Implanting ltrace to trace infected application

In this experiment, we implant *ltrace* to trace a *ls* application that is infected by *caline*. *Caline* is an ELF infector using Segment Padding Infection (SPI) technique. It inserts virus code after the code segment of ELF binary to change its behavior. Through tracing the infected *ls*, we can easily identify the deviated execution path by checking the arguments of library calls. The box in Figure 2.6 presents the suspicious execution results.

System events tracing is one of the most effective techniques of computer forensics to collect evidence of malware and is also the weak point of VMI techniques, especially in the era when hardware virtualization technology has been widely deployed. Traditional methods used in QEMU-based [16] system to intercept system call cannot be used any more because the system call instructions are not privileged instructions and would not cause VM exit for hardware virtualization. The common technique now is to set a trap point at the system call table entry address or set a page fault manually to cause the VM exit. These methods introduce great performance degradation because VM entry and exit are heavyweight operations [17]. Plus there is no introspection techniques now can track finer-

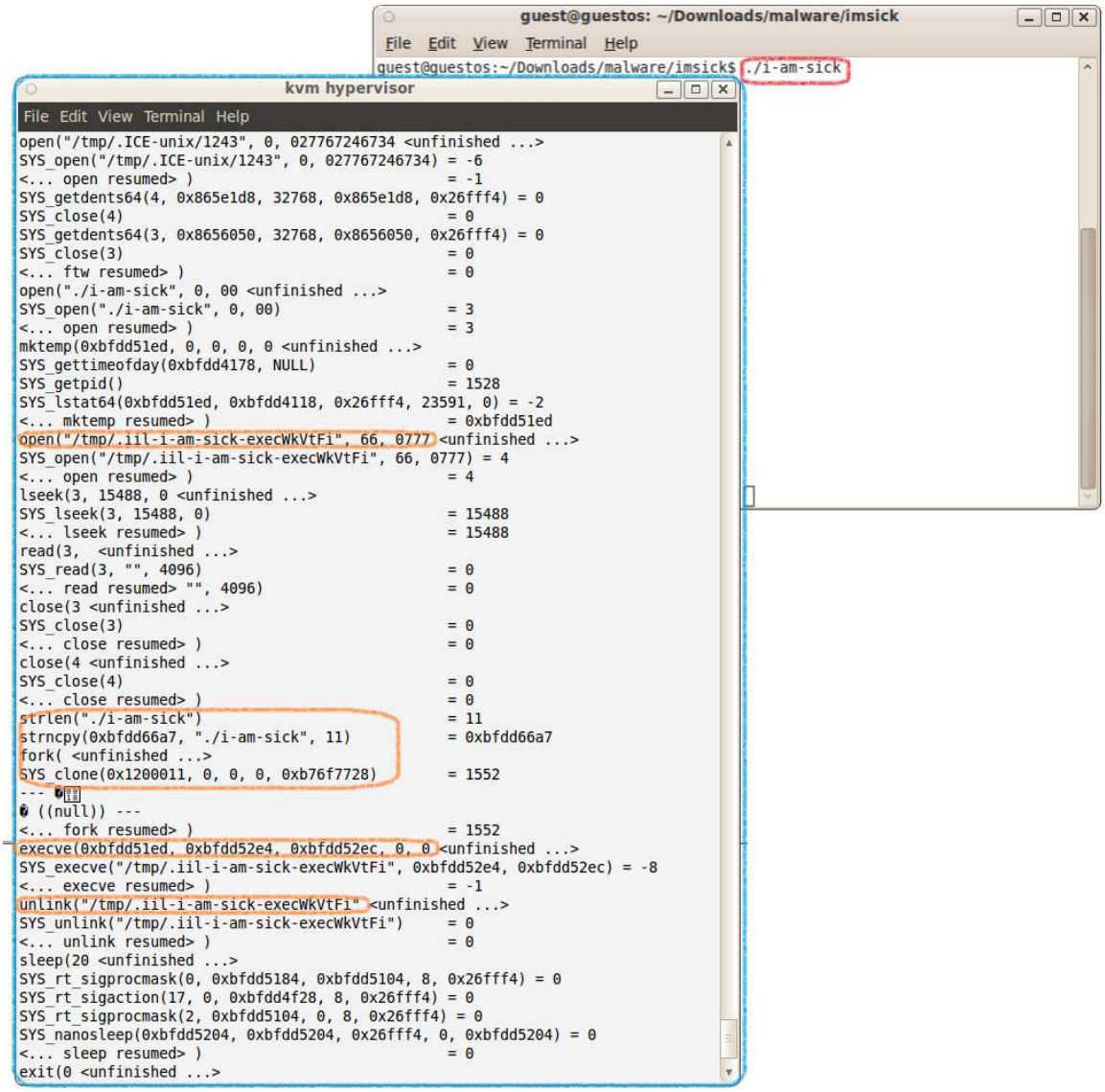

Figure 2.5.: Implanting *ltrace* to trace malware

grained events, such as library function calls. PROCESS-IMPLANTING could efficiently bridge this gap.

Experiment IV: Installing kernel module

A loadable kernel module is an object file to extend the capabilities of a running base kernel. It is a flexible approach to supporting new file system, installing device driver, and

Figure 2.6.: Trace the library call of infected application

adding new system calls. As we mentioned in the previous sections, in the cloud computing environment, PROCESS-IMPLANTING is a feasible approach to installing kernel modules into the guest OS. In this experiment, we compile a kernel module whose name is *PI.ko*. Then we implant an agent which has the function to get the module from the hypervisor through a TCP channel and insert it into the guest OS. Because installing kernel module requires root privilege, we elevate the privilege level of the *implanted process* to complete this task. We have taken a screenshot to demonstrate the result. In Figure 2.7, there are two terminals named “Before implanting” and “After implanting”. Before implanting, we

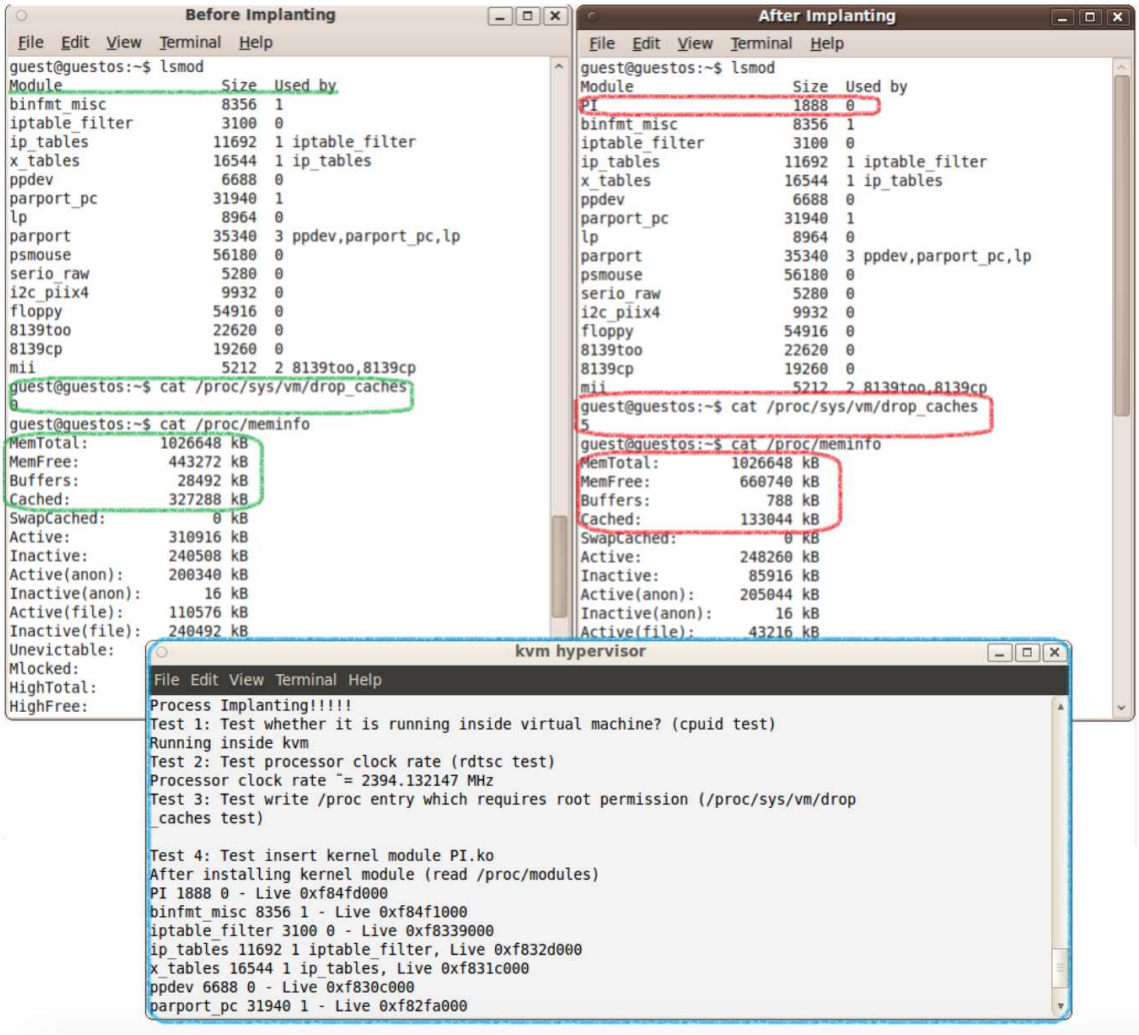

Figure 2.7.: Installing kernel module and adjusting system parameters

use the *lsmod* command to list all the installed kernel modules, there is no module named PI printed out in the terminal. After implanting, we run *lsmod* again and the *PI* kernel module has been installed on it. Please look at the first red box in the window of "After implanting".

Experiment V: Reading system information and adjusting system parameter

In order to demonstrate the application scenario for cloud computing to monitor and adjust the performance of running guest OS, we design this experiment to give some implication. In this experiment, we implant an agent into the guest VM and execute several test cases.

- Test case 1: Issue the instruction of *cpuid* to detect the type of the hypervisor.
- Test case 2: Test the instruction of *rdtsc* to read the CPU clock rate.
- Test case 3: Write value into drop cache entry in the */proc* file system. After writing this value into the entry, the guest OS drops the page, inode, and dentry cache to free the memory.

The effects of *implanted process* can be seen in Figure 2.7. In the blue box, it is the terminal to run the KVM hypervisor on the host. We can see that the information is printed on the screen through the hypercalls of the implanted process. In the green box of Figure 2.7, the value for the entry */proc/sys/vm/drop_caches* is 0. In the red box of the terminal of "After implanting", the value is 5 instead. We can compare the free memory in these terminals. In the "Before implanting" terminal, the free memory is 443272 KB. In the "After implanting" terminal, the free memory is 660740 KB. Buffers and caches are released through this method. Reading and adjusting value from */proc* cannot be achieved by traditional introspection approach because files in */proc* are memory-based and callback function handlers are only triggered in the event of */proc* reading or writing operations.

This capability can greatly simplify the procedures of performance debugging in the large-scale cloud computing environment. *Implanted process* can act as an agent to collect the performance data. Compared with these methods that only rely on statistical inference, the data directly from the guest system is more intuitive for performance diagnosis.

2.5.3 Performance Measurements

Our testing platform is Dell Optiplex 755 with Intel® Core™2 Quad Q6600 2.40GHz CPU and 3GB memory. We allocate 1GB memory to the guest VM. The performance of *implanted process* is measured in three scenarios:

Implanting Disabled

In this scenario, all the capabilities of PROCESS-IMPLANTING are disabled. It is used as the baseline for performance measurement.

Implanting Enabled

The function of PROCESS-IMPLANTING is enabled in this scenario. But we still disable the feature of FSR which is used to enhance the security by restoring the execution scene when the *implanted process* is scheduled out.

Enable Both Implanting and FSR

The FSR is enabled along with implanting in this scenario to measure the performance overhead that is introduced by this feature.

We implement a program as a micro-benchmark to test the performance. Its main function is to read and write the entries in the */proc* file system, allocate/free memory. This program runs for 1000 times to get the average running time. Three different kinds of applications in guest operating system are used as *victim processes*, *gnome-power-manager*, *vmstat*, and *gimp*.

Gnome-power-manager is a session daemon to manage the power for the laptop or desktop. It is a good candidate for the *victim process* because it is scheduled periodically to check the status of the battery and the AC power. *Vmstat* is a command-line tool to report the virtual memory statistics. It has no interaction with the user. *Gimp* is an image manipulation program under Linux. It is an interactive GUI program. Figure 2.8 shows

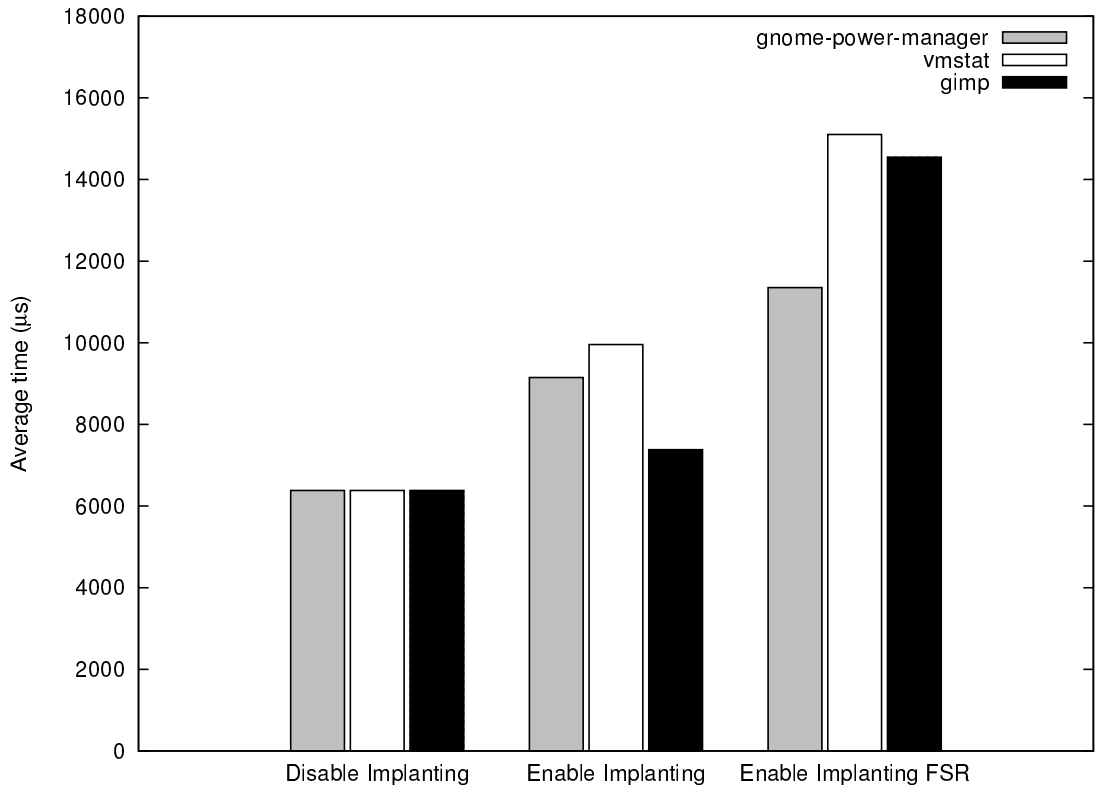

Figure 2.8.: Performance comparison of *implanted process*

the performance results of this micro-benchmark. The y axis is the average time to run micro-benchmark for one time. If we only enable the implanting without FSR, it introduce 43.4% 55.6% and 15.4% performance overhead separately for these three *victim processes* comparing with running the process directly on the guest operating system. With FSR enabled, the performance overhead increases by 24.1%, 51.6% ,and 97% comparing with the system with only implanting enabled. The performance overhead comes from two sources. The first one is introduced by the VMI. Debug register is set at the entry address of context switch function of the guest kernel. Virtual machine exits when there is a process scheduled to run. The other source of performance overhead is from the FSR. FSR will restore the execution scenario of *victim process* to eliminate the possibility for other processes to detect the occurrence of implanting. The effectiveness of FSR has been demonstrated in the experiment I. In FSR, the memory region list is restored every time when there is context

switch. It may introduce more overhead if the *victim process* itself is more complex. When the *victim process* is *gimp*, we can see from Figure 2.8 that the running time jumps by 97% after FSR is enabled. It is reasonable because *gimp* is a more complex software than *vmstat* and *gnome-power-manager*.

We need to point it out that the performance overhead is only at the time of implanting. PROCESS-IMPLANTING is designed to be modular and decoupled with other functionalities of KVM hypervisor. The implanting capability can be turn off easily without impacting other components in the hypervisor. When the *implanted process* exits, the function of implanting can be disabled by removing the breakpoints set by the debug registers and the system can recover to its original performance level.

2.6 Summary

PROCESS-IMPLANTING is a general-purpose active introspection component in our hypervisor-based security framework. It creates a channel to implant a process from hypervisor into a guest VM and run it under the cover of an existing process. Through the coordination and protection from the hypervisor, the *implanted process* can achieve strong tamper-resistance and stealthiness in the guest VM. We also propose a series of application scenarios in the areas of both security and cloud VM management, and demonstrate the feasibility and effectiveness of PROCESS-IMPLANTING in our evaluation.

3 DRIP: DRIVER PURIFICATION COMPONENT

3.1 Problem Statement

In state-of-the-art design of commodity operating systems, drivers usually take the form of loadable kernel extensions. Privileged users could load them dynamically to support new devices or extend functionalities of a base kernel at runtime. They hide the complexity of interacting with hardware devices and present a neat abstract interface for other kernel components. To achieve these properties, drivers execute with the same privilege as the OS kernel, which makes them susceptible targets of malicious attacks. Unlike the kernel, which is either built by trusted companies or with source code opened to the public, kernel drivers could be provided by third-party vendors as a binary blob.

Given a binary driver, it is difficult to tell whether malicious logic is embedded inside it. From customers' perspectives, it may work correctly with no suspicious symptoms, but the embedded malicious code [18, 19] may have already collected confidential information and cloaked its fingerprint under the cover of a legitimate driver. Even if we assume that vendors only perform the functionalities as they claim, there still exist many binary driver infection techniques [20–25] that could implant malicious logic into benign drivers and transform them into trojaned drivers. When the trojaned driver is loaded into an operating system, the hidden malicious code can be loaded simultaneously with the benign code. Hence the challenge is: how can we identify malicious/undesirable logic in the driver and eliminate it at binary level without impairing driver's normal operations?

Existing research efforts to protect device drivers can be divided into two categories, online monitoring and offline profiling. Online approaches [26–29] were proposed to isolate the driver in a protection domain and enforce external runtime checks on its execution. They either cannot target intentionally malicious drivers or require protection from the underlying hypervisor. All of them add non-trivial performance overhead due to the realtime

monitoring. Offline approaches [30, 31] are designed to exercise the driver during testing to find bugs and vulnerabilities, but they are still incapable of distilling benign operations and eliminating malicious behaviors in the driver.

We develop a security component called DRIP in our active protection framework to address this problem from a different angle. Based on our observation, we find that malicious/undesirable logic embedded inside many trojaned kernel drivers is orthogonal to drivers' normal functionalities and most such logic achieves malicious effects through interacting with the base kernel through *kernel API invocations*. Removing these interactions in malicious code will not affect the correct execution of the driver and it can also neutralize the malicious behavior. We leverage test suites for the semantic-level behavior of applications [32–34] in order to ensure that the driver works correctly when used by those applications. By testing the different application level behaviors, we simultaneously test and ensure all of the underlying benign driver functionality that applications use.

We record interactions between a subject driver and the kernel during testing. Then we try to select and remove a subset of driver-kernel interactions to test whether this removal operation will violate the correct execution of the test suite. We iterate this testing process until all unnecessary interactions are removed, and consequently we can generate a purified driver with malicious/undesirable behaviors removed.

DRIP has following contributions:

- A testing approach for differentiating between benign and malicious logic of a trojaned driver. DRIP only requires a high-level test suite to cover and retain core legitimate functionalities of the driver.

- A Test-and-Reduce algorithm to incrementally reduce unnecessary kernel-driver interactions and extract a minimal subset to ensure the correct execution of the driver.

- A clustering mechanism to group kernel-driver interactions according to current execution context. It provides additional semantic information to speed up the removal of kernel API invocations in the Test-and-Reduce algorithm.

The rest of this chapter is organized as follows. Section 3.2 presents the motivation and overview of the DRIP component. Section 3.3 provides the detailed design of DRIP. Section 3.4 gives functional studies of some representative cases and evaluates the performance. We summarize DRIP in Section 3.5.

3.2 System Overview

3.2.1 Goals and Assumptions

The goal of DRIP is to purify a device driver with malicious/undesirable logic embedded that may jeopardize the base kernel. The newly generated driver should have the benign functionalities of a vanilla driver with malicious effects eliminated.

Our approach is based on the assumption that the trojaned driver includes the functionalities of a benign driver. The malicious logic is parasitically attached to the benign logic within the driver's binary and executes persistently when the driver is loaded. We do not target time-bomb malware in which the malicious functions can only be triggered at a specific time because the malicious logic may not be active during our testing. This problem can be addressed by using symbolic execution [35] to cover more execution paths. There are some existing efforts [31, 36, 37] to apply symbolic execution to driver testing and we can leverage them to complement our work. In addition, we do not target the malicious code that interacts with kernel through direct memory manipulation. We could consider kernel memory accesses as part of driver-kernel interactions and plan to include this feature in our future work.

We assume that a test suite is available that covers the high-level behaviors of a specific application. As previously mentioned, testing those behaviors also means that the test suite covers the necessary driver functionality that they depend upon. Because we test the application level behaviors, our technique ensures that the application continues to behave correctly with the purified driver. This assumption is reasonable for current software development processes, in which developers often create test cases from requirements even

before implementation as part of the design phase. We can also leverage existing test generation techniques [35, 37, 38] to automatically synthesize test cases.

3.2.2 Approach Overview

For a particular application and the environment in which it executes, we need to ensure that the application continues to behave correctly. This includes correctly executing any low-level behaviors in the driver that the application relies upon and triggers during its operation. We can do this by treating the driver like a blackbox, without considering the specifics of its implementation. For example, we might examine a network interface controller (NIC) driver. We can cover the functionality of an FTP server through test cases from curl-loader [34]. If we can ensure the correct execution of curl-loader when using a purified NIC driver, then we have empirically preserved the functionalities of the driver needed by curl-loader. In general, covering the tests of an application will also cover and preserve the low level driver functionality necessary for that application.

Based on our experience of analyzing conventional rootkits, we gain the insight that the common goals of malicious code in kernel space are to retrieve information from base kernel and manipulate kernel data to hide footprints of user space malware. It is difficult to generate a completely self-contained malicious module to achieve all these effects without invoking kernel APIs. When we face a trojaned kernel driver, the execution of malicious code is mixed with the execution of benign code at runtime. Benign code of the driver will also invoke kernel APIs to request services from base kernel. So we need to differentiate benign kernel API invocations from malicious ones. With the availability of a test suite covering benign functionalities of the driver, we can iteratively eliminate some of the kernel API invocations at runtime to test whether it will violate the correct execution of the test suite. If the removal will not affect the benign behavior, we consider these invocations unnecessary; therefore, they can be removed from the binary.

Based on this observation, we first take a snapshot of the system and execute the test suite from a deterministic state. We record all kernel API invocations from the driver to the

kernel during testing, which can be captured as control flow transitions across the boundary of driver's loading memory region. Then we try to restore to the snapshot, remove a subset of these invocations in memory, and re-execute the same test suite to test whether the removal will affect its correct execution. We chop the removal set of invocations iteratively until all the invocations left are critical to the correct execution of the driver. Because benign functionalities of the driver are covered by the test suite, the removal of kernel API invocations within benign code will fail the test suite, so we consider them critical and preserve them. On the other hand, because malicious code embedded is either orthogonal or complementary to core functionalities of its "host" driver, removal of invocations within malicious code will not violate the correct execution of the test suite, thus they are considered unnecessary. Finally we can generate a purified driver with all the unnecessary invocations removed; therefore, the malicious effects from driver are eliminated concomitantly.

3.2.3 Procedure Overview

Figure 3.1 depicts the overall workflow of DRIP to demonstrate how to purify a trojaned driver. We divide the whole procedure into three phases, i.e., profiling, testing, and rewriting, as in Figure 3.1(a). These three phases are transparent to each other. We give a brief description of the specific functionality of each phase first and will elaborate upon them in the following section.

Before starting the purifying process, we construct the *Testing Environment* in Figure 3.1(b) and prepare the binary file of the trojaned driver. In the profiling phase, we execute the test suite to trigger the execution of this driver, record kernel API invocations, and cluster them according to their execution context. The output of this phase is the *Profiling Data* and it is organized in the structure presented in Figure 3.1(c). In the testing phase, we select and remove a subset of these kernel API invocations and test their influence on the correct execution of the test suite. The *Testing Data* shares the same structure as the *Profiling Data*. The only difference is that we mark testing status on every entry in the *Test-*

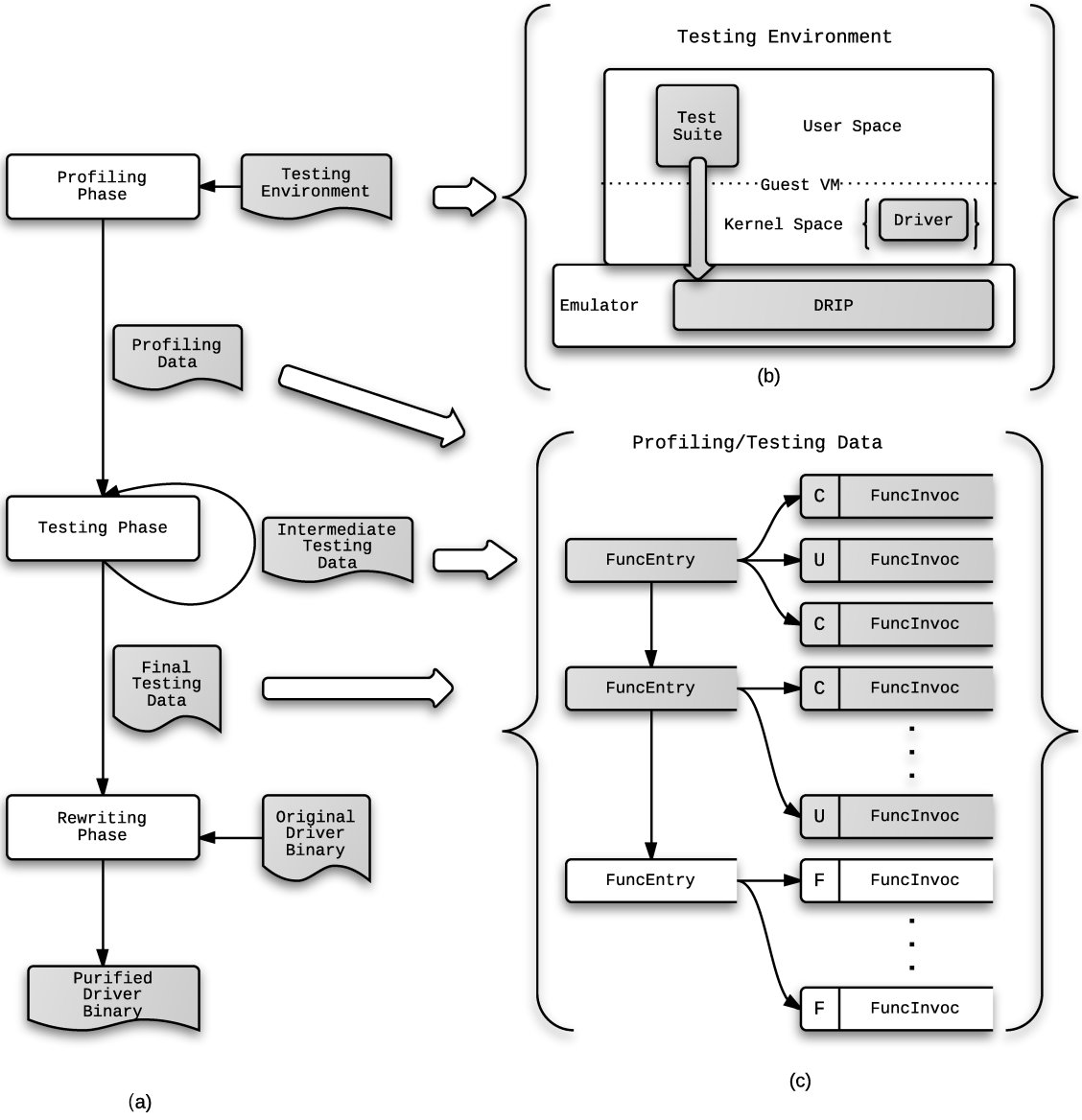

Figure 3.1.: Workflow of DRIP

ing Data. For example, in Figure 3.1(c), shaded entries in *Testing Data* indicate that they have been tested. We feed the *Intermediate Testing Data* back as input to the testing phase. The testing phase terminates when all the entries in the *Testing Data* have been tested. In the last rewriting phase, we summarize the testing result, apply the changes on the trojaned binary, and generate a purified driver.

3.3 Detailed Design

In this section, we describe DRIP following the workflow of the driver purification procedure and discuss the key design DRIP component in detail. First, we describe the setup of the *Testing Environment*. Then we demonstrate the profiling, testing, and rewriting phases respectively to explain the procedure of generating a purified driver. Finally, we present the technical details of the prototype implementation.

3.3.1 Environment Setup

Before the purification procedure, we set up the *Testing Environment*, prepare the trojaned driver, and design a communication channel to send test results from the test suite to DRIP.

Testing Environment

As shown in Figure 3.1(b), the *Testing Environment* consists of a guest VM and its underlying emulator (we use QEMU [16] in our environment) as the analysis platform. We integrate our DRIP as a component into the emulator. In the guest VM, we load the trojaned driver in the kernel space and monitor the code execution within its loading memory region. We select or synthesize an automated test suite for the target application to cover the benign behavior of the subject driver and launch it in the user space. In order to ensure that the test suite executes from a deterministic state, we take a snapshot of the VM at the time right before the test suite is about to run.

Communication Channel

If we pick up an existing test suite, it would have no knowledge about the underlying system including DRIP. However DRIP needs to make decisions based on the current status of the test suite. So we design a communication channel between the test suite and DRIP. We can leverage special instructions like *hypercall* or *cpuid*, to send signals to the

underlying emulator. The emulator can extract signals when translating these instructions. We design three signals, *TESTON*, *TESTSUCC*, and *TESTFAIL*, which respectively stand for the beginning of the test, the end of the test with a successful result, and the end of the test with a failing result. Then we embed the communication channel in the test suite to send these signals at their corresponding events.

3.3.2 Profiling Phase

In the profiling phase, we record all kernel API invocations/returns during the execution of the test suite. Because all recorded invocations in different *process context*s are mixed, we design a technique called *Context-Sensitive Clustering* to de-interleave invocations into clusters and label each cluster with *FuncEntry* tag. After the recording and clustering of invocations, we organize the runtime information captured into the *Profiling Data* and transfer it to the next testing phase.

Tracking of Driver-Kernel Interactions

Because QEMU can translate every instruction in the guest VM, we track the execution of the driver through monitoring its program counter at the granularity of a basic block. If the current basic block is within the driver's memory region and the previous one is located outside, it means that control flow transits from the kernel into the driver. If the previous basic block is within the driver's region and the address of the current one is out of the driver's boundary, it indicates that the control flow transits from the driver into the kernel. Then all control flow transitions passing the driver boundary can be recorded. The transitions between kernel and driver are either in the form of a *call/jump* instruction or a *ret* instruction.

As mentioned earlier, we prepare a test suite for the subject device driver we want to test. When the test suite begins to execute, we issue *TESTON* to notify DRIP of the start of the test. When the test finishes successfully or terminates due to an assertion failure, it also notifies our system with the result through *TESTSUCC/TESTFAIL* respectively. We denote

it as one *Testing Cycle* from the beginning of a test to the end. We record all the transitions that are issued through *call/jump* instructions from the driver to the kernel in one *Testing Cycle* and we treat them as kernel API invocations.

After recording kernel API invocations from the driver to the kernel, we need to capture the return value of each invocation because it may be used by subsequent instructions. We record the transitions that are issued through the *ret* instruction from the kernel to the driver and we treat them as kernel API returns. The return value is stored in a general register, e.g., *EAX* under x86. Some kernel APIs are void functions or the return values are not used any further. We check the def-use of *EAX* in subsequent instructions to determine whether the return value is used or not. If *EAX* is defined first, it indicates that the return value is not used and has no effect on later instructions. If *EAX* is used first, we need to record this value and map it to the function invocations recorded before.

Due to multitasking in the OS, a kernel driver's code can be executed concurrently in different *process context*s. Most OSes also enable the features of kernel reentrancy and kernel preemption, which mean all processes can be interrupted in the kernel mode and resumed from a previous checkpoint when the interrupt is handled. These properties make it complicated to create one-to-one mapping from the kernel API return to its invocation. Fortunately, the starting address of the kernel stack for different processes/threads is different and can be used to uniquely identify the *process context*. We leverage this property to identify the current context of the driver code being executed. Processes may be interrupted to handle hardware interruptions and nested interrupts are possible. It conforms to the Last In First Out (LIFO) order in the same *process context*. We maintain a call stack for every active process to record the last function invocation and its expected return address. When a function returns, we can find the call stack according to the current *process context* and map the return value to the last function invocation stored in this call stack and pop it.

We give a simplified example in Figure 3.2. We assume processes 1 and 2 are running simultaneously in the system and both request the same service of the kernel driver (dotted red paths 1 and 4 for process 1 and dotted blue paths 2 and 3 for process 2). For the execution of driver's code in Process 1's context, it invokes API 1 (solid red path 6) of the

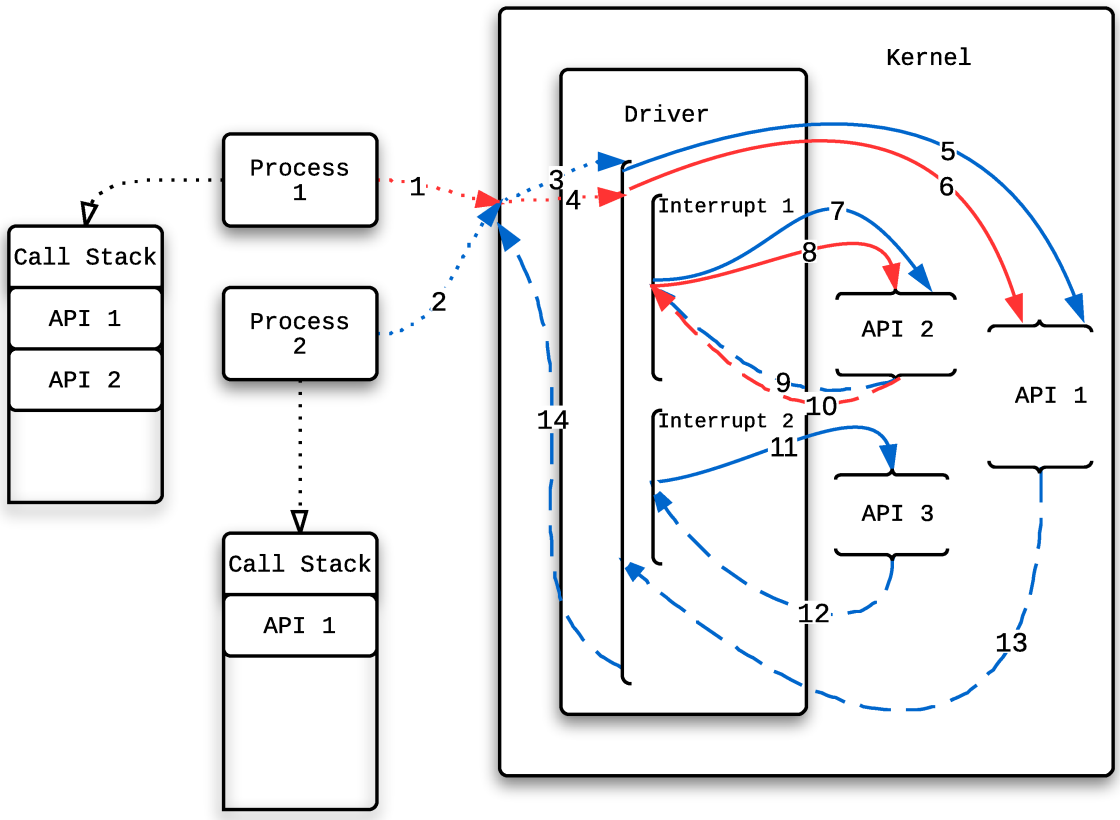

Figure 3.2.: Function return value mapping

kernel and is interrupted before returning from API 1. Then it calls API 2 (solid red path 8) in the handler of interrupt 1. Before returning from API 2, the call stack of Process 1 contains both API 1 and API 2. When returning from API 2 (dashed red path 10), the current *process context* is Process 1 and it can map the return to API 2 in Process 1's call stack and pop API 2 from call stack. For the execution in Process 2's context, both API 2 and API 3 in two interrupt handlers have returned (dashed blue path 9 and 12) and popped from the call stack. There is only API 1 in the call stack. When API 1 returns from the kernel (dashed blue path 13), its current *process context* is Process 2 and then we can map the return value to API 1 and pop it from the call stack.

Figure 3.3.: Context-sensitive clustering

Context-Sensitive Clustering

After recording all kernel API invocations in one *Testing Cycle* linearly, we find that invocations in different *process context*s may interleave with each other due to multitasking and kernel preemption. In Figure 3.3(a), we present recorded invocations of a trojaned E1000 NIC driver. It is compromised by the module injection technique [22] and the payload is a DR rootkit. Each entry contains a symbol name (just used for clear demonstration, symbols of the driver are not needed by DRIP), *funcaddr*, and *apiaddr*. *Funcaddr* is the function invocation's call site address in the driver and *apiaddr* is the API's entry address

in the core kernel. Interleaved invocations make it difficult to design an efficient removal strategy in the following phase because there is no information of connections among invocations.

We design a technique called *Context-Sensitive Clustering* to de-interleave kernel API invocations recorded during the profiling phase. It is based on the observation that, for the trojaned driver, each function in the driver either belongs to the benign logic or to the malicious logic and we can group kernel API invocations issued under the same function together. Thus after clustering according to each function address in the driver, interleaved kernel API invocations belonging to benign/malicious logic are naturally separated and become easier to process in the next phase.

In Figure 3.3(b), we present the result after applying *Context-Sensitive Clustering* and organize the kernel API invocations in reverse chronological order. The entries in red are function invocations from the DR rootkit and those in blue are from the E1000 NIC driver. We denote the group clustered as a *Context Group* and present one specific example in the red rectangle. This *Context Group* is headed with *hook_execve* entry addr:0xf81f08b0 and it contains three function invocations, *ptregs_execve*, *strstr* and *getname*. It means during the execution of function *hook_execve* whose entry address is 0xf81f08b0 in the driver, it invokes these three kernel APIs. We combine the clustered kernel API invocations with the return values to generate the *Profiling Data* and transfer to the testing phase.

3.3.3 Testing Phase

In the testing phase, DRIP eliminates kernel API invocations that do not affect the correct execution of the test suite, which ensures the preservation of the driver's benign functionalities. We obtain the *Profiling Data* that contains clustered kernel API invocations from the preceding profiling phase and rename it as *Testing Data*. Initially, entries in the *Testing Data* are not marked with any status.

Algorithm 2 Test-and-Reduce algorithm in the testing phase

```
Input:  ContextGroupListhead ← FirstContextGroup
        FuncStack ← EmptyStack
        CurrContextGroup ← NULL
        CurrFuncList ← NULL
        ENTRYFUNC ← DISPATCH(signal)
1:  procedure DISPATCH(signal)                                  ▷ Dispatch based on signal
2:      if signal = TESTON then
3:          PATCHTESTEDFUNCS()
4:          PATCHCURRFUNCLIST()
5:      else if signal = TESTSUCC then
6:          MARKCURRFUNCLIST(UNNEC)
7:          LOADSNAPSHOT()                                      ▷ Load Snapshot of VM
8:      else if signal = TESTFAIL then
9:          if SIZE(CurrFuncList) = 1 then
10:             MARKFUNCLIST(CRITICAL)
11:         else
12:             RECOVERCURRFUNCLIST()
13:             {FuncList_1, FuncList_2} ← SPLITLIST()
14:             PUSHSTACK(FuncStack, {FuncList_1, FuncList_2})
15:         LOADSNAPSHOT()
16: procedure PATCHTESTEDFUNCS(void)
17:     ContextGroupIter ← ContextGroupListhead
18:     while ContextGroupIter ≠ NULL do
19:         if ContextGroupIter.status = TESTED or TESTING then
20:             for all Func in ContextGroupIter.funclist do
21:                 if Func.status = UNNEC then
22:                     REMOVEFUNC(Func)                        ▷ Remove the invocation in Memory
23:         if ContextGroupIter.status = TESTING then
24:             ASSERT(CurrFuncList ≠ NULL)
25:             return
26:         if ContextGroupIter.status = UNTESTED then
27:             CurrContextGroup ← ContextGroupIter             ▷ Init CurrContextGroup
28:             CurrFuncList ← ContextGroupIter.funclist        ▷ Init CurrFuncList
29:             ContextGroupIter.status ← TESTING
30:             return
31:         ContextGroupIter ← ContextGroupIter.next
```

Algorithm 2 Test-and-Reduce algorithm in the testing phase (continued)

```
32: procedure PATCHCURRFUNCLIST(void)
33:     for all Func in CurrFuncList do
34:         REMOVEFUNC(Func)
35: procedure MARKCURRFUNCLIST(status)                ▷ Mark statuses in the CurrFuncList
36:     if status = UNNEC then
37:         for all Func in CurrFuncList do
38:             Func.status ← UNNEC
39:     else if status = CRITICAL then
40:         ASSERT(SIZE(CurrFuncList)=1)
41:         CurrFuncList[0].status ← CRITICAL
42:     RECOVERCURRFUNCLIST()
43:     if ISEMPTY(FuncStack) then
44:         CurrContextGroup.status ← TESTED
45:         CurrFuncList ← NULL
46:     else
47:         CurrFuncList ← POPSTACK(FuncStack)
48: procedure RECOVERCURRFUNCLIST(void)               ▷ Recover CurrFuncList
49:     for all Func in CurrFuncList do
50:         RESTOREFUNC(Func)                         ▷ Restore Invocation in Memory
```

When the *Testing Cycle* begins, we load the snapshot to execute the test suite from a deterministic state. Upon receiving *TESTON*, a subset of kernel API invocations that have not been marked will be selected and removed from the memory. As aforementioned, we cluster kernel API invocations in the profiling phase into different *Context Groups*. Selection of candidates for removal is based on the clustering. First we select one *Context Group* that is marked as *UNTESTED* and try to remove all function invocations in it. Then we change its status to *TESTING*. We maintain a *FuncStack* to record the current function invocation list that is being tested. Then we enter the VM to resume executing the test suite. If it runs to completion successfully, we mark the current kernel API invocations as *UNNEC*, which means that they do not violate the correct execution of the test suite and can be removed before the next *Testing Cycle*. If the removal causes failure of the test suite, we utilize a divide and conquer approach to split the *Context Group* into two equal subsets and push them into the *FuncStack*. Then we recover current invocations being tested in memory and re-launch the next *Testing Cycle*. If the current set contains only one

```
asmlinkage int h4x_unlink(const char __user *pathname) {
    . . .
    char *kbuf=(char*)kmalloc(256,GFP_KERNEL);
    copy_from_user(kbuf,pathname,255);
    . . .
    r=(*o_unlink)(pathname);
    kfree(kbuf);
    . . .
}
```

Figure 3.4.: Reverse chronological order

function invocation, which cannot be divided any further, we mark this function invocation as *CRITICAL* if test fails. If all kernel API invocations in the same *Context Group* have been tested, we mark this context group as *TESTED* and continue to process the next one. We iterate this process until all kernel API invocations in the *Testing Data* are marked. The detailed algorithm is presented in the Test-and-Reduce Algorithm 2.

Recall that we list every function invocation in the *Context Group* using reverse chronological order. The reason is that the result of earlier function invocations will probably impact the later function invocations. But removing the later invocation first will not impact the earlier ones. In Figure 3.4, we present the function *h4x_unlink* from KBeast, which is one of the malicious payloads in our evaluation. *h4x_unlink* is used to hijack the *sys_unlink* system call from Linux. It analyzes the pathname argument and protects its own malicious files from being deleted. We highlight 3 function invocations in blue, which are *kmalloc*, *copy_from_user* and *kfree*, in the function body. If we remove these 3 function invocations together in one *Testing Cycle*, it is safe and will not cause problem. But other kernel API invocations located between these 3 invocations may be marked as *CRITICAL*.

In this example, *o_unlink* cannot be removed because it is the function pointer to the original *sys_unlink*. Removing it can make deletion of files ineffective. This critical function invocation splits the current *Context Group* and forces removal of these 3 function invocations to occur in different *Testing Cycles*. If we do not use reverse chronological order, we will try to remove *kmalloc* first and assign *kbuf* with a fake address. The subse-

quent function *copy_from_user* will write to an unsafe address and *kfree* will free a memory block that has never been allocated. This will probably crash the system. Then we will mistakenly mark *kmalloc* as *CRITICAL*, but in fact it is not. If we remove backwards in the following order: *kfree*→ *copy_from_user*→ *kmalloc*, all 3 invocations will be safe to remove. This greatly reduces the risk of mutual influences of function invocations.

In order to accelerate the handling of failing cases, we add two optimization techniques to handle different failing scenarios:

(i) *Test suite halts in the middle:* Removal of some function invocations will cause the test case to freeze without progress. We handle this by setting a timer and the time interval is estimated by profiling the execution time of previous successful cases. If the timer expires, we consider this *Testing Cycle* a failure and proceed to execute the next one.

(ii) *Test suite causes OS crash and rebooting:* Removal of a critical function invocation may cause OS crash and rebooting. In this case we do not have to wait for the timer to expire. Instead, we add rebooting detection logic by checking whether the paging bit is set in the control register. We can determine that it is a failing case if the system is rebooting after we remove certain invocations.

To eliminate kernel API invocations in the driver, we patch them in the driver's memory. The method of patching varies according to platforms and file formats. For ELF under Linux, the destination address of call instructions is unknown before loading. The module loader resolves symbols of the kernel API in the entries of the relocation section and fixes up the destination in the code section with the absolute address when loading the kernel module. For Portable Executable (PE) under Windows, it utilizes the import address table (IAT) to store the absolute virtual addresses of kernel APIs. The contents are populated when that driver is loaded into the system. The kernel API invocations in PE drivers use two calling styles. The first one uses the indirect call generated by the compiler and retrieves its destination address from IAT. The second one makes a direct call to an indirect jump and the jump destination is stored in the IAT. If the return value is not used by the subsequent

instructions or it is a void function, we can simply replace the *call* or *jmp* instruction with a series of nops in memory. If the return value is used later (e.g., as a predicate condition) and can determine the control flow, replacing the instruction with nops will lead to an undefined situation.

As we have already recorded the return value of every kernel API invocation in the profiling phase, we can replace the calling instruction with a *mov* instruction that fills the return value in the *EAX* register. This kind of replacement can be applied to the ELF driver and the first calling style of PE drivers. For the second calling style of PE drivers, we replace the indirect jump with *ret* to return to the original direct call to eliminate this invocation.

The other issue we need to consider during memory patching is the calling convention of the kernel API. If the caller is responsible for cleaning up the stack, no additional effort is needed because *push* and *pop* operations are performed in the same function. If the callee is responsible for cleaning up the stack, the situation becomes more complicated. In this scenario, arguments are pushed into stack by the caller and the callee unwinds the stack before returning. We choose to remove the *push* operations before the function invocation in the caller to solve this problem. We can record the number of stack bytes that need to be unwound. This is determined by the 16-bit parameter of the last *ret* instruction in the kernel function. We then trace back from the kernel API invocation instruction to search for *push* instructions and replace these instructions with nops.

If the patching operation is successful for the current *Testing Cycle*, which means the function invocation is tested to be *UNNEC*, we record all the modified content and the address of this function invocation for the rewriting phase. After writing new content into the memory address of a kernel API invocation, we mark this specific basic block as a candidate for memory invalidation. When the snapshot is reloaded in the next *Testing Cycle* and the emulator tries to execute this basic block, we invalidate the cache of this basic block and force the emulator to perform binary translation on it because the instructions inside it have been modified and it should execute the newly translated code.

3.3.4 Rewriting Phase

The last phase of DRIP's driver purification procedure is to remove kernel API invocations marked as *UNNEC* in the binary file. We have already tested and retrieved the list of unnecessary API invocations and their addresses in the memory from previous phases. The procedure of patching the binary is similar to patching memory in the testing phase. We need to map the loading addresses of API invocations to their relative addresses inside the binary and apply the changes recorded in the testing phase to code sections.

Finishing these steps is not enough for the purified driver to work correctly. Every relocatable driver has its own relocation table consisting of a list of pointers. These pointers point to addresses in the binary that need to be fixed up after the driver is loaded into the system. If we remove the function invocations whose addresses are included in the relocation table, we also need to remove these relocation entries in the relocation table. Otherwise the loader of the OS will still fix up the function address and cause memory corruption. Because holes are not permitted in the relocation table for both ELF and PE, we swap the value of each removed entry with the value of last entry in the relocation table to fill the hole and adjust the table size in the header accordingly. For PE files, we also need to calculate the new checksum value and write it into its PE header, otherwise Windows will refuse to load the driver with the wrong checksum.

After finishing all these steps, we generate a new relocatable binary as a purified driver and it can be loaded into the system for execution.

3.3.5 DRIP Prototype

We have implemented a proof-of-concept prototype of DRIP. The prototype is built as a component of QEMU. As a full system emulator, QEMU dynamically translates the guest VM's code at the granularity of basic blocks and executes them on the emulated CPU. Such a platform enables us to perform binary analysis on the code region of drivers, intercept dynamic control flow, and patch the memory at runtime to test effects of our kernel API invocation removal operations. In addition to processor emulation, QEMU also provides a

set of emulated devices, which provides an alternative to verify the correctness of test cases through mapping the high-level program to low-level hardware events. For example, we can simulate keystrokes in emulated hardware and capture the keys in the test suite to test the keyboard driver.

To prove the generality of DRIP, We have tested the prototype on two guest operating systems, Ubuntu 10.04 and Windows XP SP2[1]. We believe that it is easy to extend our current system to support more operating systems of different versions because DRIP does not rely on the semantics of a guest VM. We support relocatable file formats for both PE and ELF, which are standard formats for Windows and Linux drivers.

3.4 Evaluation

In this section, we present the evaluation results for the DRIP prototype in two aspects, effectiveness and performance. The hardware configuration of our testing platform is a Dell OptiPlex 780 with Intel® Core™ 2 Duo CPU E8400 3.00GHz CPU and 4GB memory. We develop and run the DRIP component on Ubuntu 11.10 (Linux kernel version 3.0.0) to generate the purified driver. To prove that changes in the underlying infrastructure do not affect the functionality of purified drivers, we use VMware Workstation 8.0 as the hypervisor and Windows 7 as the host operating system to perform evaluation on purified drivers. We allocate 1GB memory for each guest VM. The guest OSes are Ubuntu 10.04 (Linux kernel version 2.6.32) and Windows XP SP2.

3.4.1 Evaluation of Effectiveness

In the effectiveness evaluation, we use trojaned drivers infected by binary driver rewriting tools as input to DRIP and generate the corresponding purified drivers. Then we scrutinize the behavior of the generated driver manually to validate that the malicious behavior has been eliminated and the functionality of the benign parts of the driver and the kernel are

[1]We use Ubuntu 10.04 and Windows XP SP2 because the trojaned driver samples we perform evaluation on do not support newer versions yet.

not impaired. We present five representative case studies on drivers in different categories in detail and present other results briefly in Table 3.1.

Case Study I: E1000 NIC Driver with DR Rootkit Implanted under Linux

In phrack issue 61 [20], truff described a driver infection technique to hide the rootkit and ensure that it will be reloaded after rebooting. The basic idea is to rename the malicious function *evil* with *init* in the section *.strtab* to trick the system to load it. It only applies to Linux kernel 2.4.x, so it is no longer valid for the latest Linux kernel because the module loading procedure has been changed in new kernel version. From Linux Forum [22], coolq extended this approach to Linux Kernel 2.6.x by modifying the module *init* function entry in the relocation section *.rel.gnu.linkonce.this_module* to guide the system to load the initialization function in the malicious module. In the latest issue 68 of phrack [21], styxˆ proposes a similar approach to infecting modules in kernel versions 2.6.x and 3.0.x. It redirects *init_module* to load function *evil* instead of original *init* function. In order to enable malicious modules to invoke the original init function, it also updates the symbol binding of *init* from local to global. The effects of these two approaches are equivalent and we choose to use the former method to inject DR rootkit into an E1000 NIC driver as our target.

The DR rootkit leverages a debug register-based hooking engine, which does not require modification to the system call table, to perform traditional rootkit behavior, like hiding processes, sockets, and files. To be more specific, it determines the name (in the version we obtain the name is *AAA*) of a file it wants to hide. Then it hides the presence of this file in the file system by modifying the file listing result in the directory. When executing this file, the rootkit escalates *AAA*'s privilege to root, hides all the sockets created, hides all the child processes forked, and prevents other processes from opening files owned by *AAA*.

The trojaned driver contains both the functionality of a benign E1000 NIC driver and a malicious kernel rootkit. We pass it to DRIP to deactivate its malicious behavior and retain the benign NIC driver behavior. We select and synthesize test cases from LTP (Linux Test

Table 3.1.: Results of effectiveness evaluation against a spectrum of trojaned drivers

Name	Infection Type	Platform	Purified	Note
E1000+KBeast	Module injection	Linux	✓	E1000 driver infected with KBeast as payload
E1000+DR	Module injection	Linux	✓	Case Study I
E1000+Adore-ng	Module injection	Linux	✓	E1000 driver infected with Adore-ng 0.56 as payload
E1000+Sebek	Module injection	Linux	✓	E1000 driver infected with Sebek-3.2.0b as payload
E1000+Redir	ERESI	Linux	✓	Cast Study II
Kbdevents	Embedded	Linux	✓	Case Study III
Null+SSDT	DaMouse	Windows	✓	Null.sys infected by DaMouse
Kbdclass+SSDT	DaMouse	Windows	✓	Case Study IV
E1000325+SSDT	DaMouse	Windows	✓	E1000325.sys infected by DaMouse
Beep+Klog	Binary Transformaion	Windows	✓	Case Study V
E1000325+Klog	Binary Transformation	Windows	✓	E1000325.sys infected with Klog as payload

Project) [32], Linux utility programs, and Iperf to cover the benign functionalities of E1000 NIC driver and the reliability of the overall system. We have validated that the purified driver behaves the same as a benign E1000 NIC driver with the malicious operations from the DR rootkit eliminated.

Case Study II: E1000 NIC Driver with Kernel Function Redirection under Linux

From case study I, we learn that we can implant malicious code inside the initialization function to install system call hooks. In fact, when the driver code invokes the kernel function, we can intercept and redirect any function invocation to a malicious function first. The malicious function can act as a proxy to invoke the original function and return the result to the original invocation. This kernel function redirection technique is proposed in the libkernsh of ERESI [25].

We prepare an interposition kernel module, which contains malicious functions from the KBeast rootkit and link it with the E1000 NIC driver to generate a trojaned driver. The relocation table of this new driver contains all the addresses of code/data that need to be fixed up during loading. We scan this table to find the function invocation we want to hijack and modify it to detour to the malicious function in the interposition module. The payload, KBeast, is a new kernel rootkit based on other well-known rootkits and supports the latest Linux kernel versions. It contains traditional rootkit functionalities, e.g., process hiding, files hiding, keystroke logging, and local root escalation. Its basic idea is to patch the system call table of Linux and detour system calls to its fake functions that are crafted by the attacker. Because system calls are hijacked, KBeast can easily manipulate the intermediate results and return fake results to the user. We select similar test cases as in Case Study I to build our test suite to ensure the reliability of the system and core benign functionalities of the E1000 NIC driver. After purification, we validate that KBeast's cloaking effects on the system have been eliminated and we still preserve the E1000 driver's original functionalities.

Case Study III: Kbdevents under Linux

Kprobes [39] is a lightweight debugging mechanism in the Linux Kernel that allows developers to intercept kernel routines at runtime to collect debugging information. Kbdevents [40] is a Linux Kernel module based on Kprobes to intercept keyboard events. It can be used as a debugging tool to verify the correctness of the keyboard driver. On every key pressed, Kbdevents has additional functionality to launch user scripts from kernel space, e.g., *keylogger* to dump keystrokes into a file, *printscr* to take screenshots and *typewriter* to imitate typewriter sounds. These supplementary capabilities are not necessary for debugging purposes. So we can perform purification on Kbdevents to minimize it to contain only the debugging functionality. We build a special test suite to simulate keystrokes out of VM, i.e., generate keyboard interrupt from QEMU, and capture them in the guest VM to verify the correctness of Kbdevents' debugging functionality. After purification, we find all kernel API invocations related to launching user scripts from the kernel (e.g., $call\_usermodehelper\_\{setup,exec\}$) have been removed from the driver. The purified driver can still intercept keystrokes to debug the Linux keyboard driver.

Case Study IV: Infected Kbdclass Driver by DaMouse under Windows

DaMouse [23] is a PE driver infection technique under Windows. It implants existing malicious code into a windows device driver in the system. It utilizes a virus coding technique called Entry-Point Obscuring (EPO) to patch API invocation inside the device driver. When this patched API is invoked, it installs a permanent System Service Dispatch Table (SSDT) hook to redirect the system call to the hook function inside the driver. The hook function contains malicious code to filter the results and can eventually complete the procedure by invoking the original system call.

In this case study, we use DaMouse to infect *kbdclass.sys*, the keyboard class driver in Windows. DaMouse patches the Kbdclass driver and install the SSDT hook at *NtOpenProcess*. Then system calls to *NtOpenProcess* are redirected to the hook function called *NewNtOpenProcess*. The filter code in the hook function determines whether the target

process is *iexplorer.exe*, which belongs to the Internet Explorer. If so, the *NtOpenProcess* request will be denied. The symptom noticeable to the user is that he/she cannot open a new web page in the Internet Explorer. For other processes, the malicious code extracts the *NtOpenProcess*' arguments, e.g., pid and name, of calling process and dumps the result through *DbgPrint*. We build the test suite for Kbdclass through sending keystrokes from QEMU into VM, which is similar to Case Study III, and verify them in the test program within the VM. After purification, we can keep the keyboard driver's functionality, Internet Explorer can open new tabs successfully and there is no process information leakage any more.

Case Study V: Beep Driver Infected with Klog as Payload under Windows

In previous case studies, we have applied DRIP to purify drivers infected by existing binary infection tools. In this case study, we try to prove the generality of DRIP by purifying trojaned drivers generated by a binary transformation tool called BISTRO [41]. BISTRO enables transplanting binary functional module extracted from one binary into another binary. We extract the malicious functions, i.e., keyboard attaching and keystrokes dumping, from klog, which is a well-known Windows keyboard sniffer. Then we utilize BISTRO to implant the extracted functions into the beep driver of Windows. In order to check if the beep driver works properly, we add some functionality-checking logic in the emulated pc speaker in QEMU to verify the beep events. After purification, we load the purified beep driver into the production environment and it works as expected and keyboard can no longer dump sniffed keystrokes to a file any more.

3.4.2 Performance Evaluation

The time it takes for DRIP to purify a specific driver is highly dependent on the driver's code complexity, coverage of test suite, and hardware configuration. We present the complete performance statistics of purification process for each trojaned driver in Table 3.2. It shows the ratio of "Removed Function Invocations" to "Recorded Function Invocations",

Table 3.2.: Performance evaluation results with a spectrum of trojaned drivers

Name	Ratio[1]	Time	NTC[2]
E1000+KBeast	57/69	42min 13s	37
E1000+DR	13/25	21min 23s	40
E1000+Adore-ng	7/23	20min 46s	39
E1000+Sebek	13/34	19min 19s	35
E1000+Redir	37/53	35min 38s	34
Kbdevents	8/12	8min 25s	13
Null+SSDT	5/7	4min 4s	12
Kbdclass+SSDT	13/21	15min 31s	32
E1000325+SSDT	20/24	22min 15s	19
E1000325+Klog	22/28	24min 35s	19
Beep+Klog	24/35	31min 1s	44

[1] Ratio here represents the ratio of "Removed Function Invocations" to "Recorded Function Invocations".

[2] NTC stands for "Number of Testing Cycles"

the purification time, and the number of testing cycles. Our results indicate that DRIP is suitable for offline driver purification.

We next measure the system performance overhead with the purified driver and compare it with system performance with the trojaned driver. We use SPECINT 2000 under Windows and UnixBench under Linux to measure the CPU performance. We normalize the performance results and present them in Figure 3.5. The left bars indicate the normalized performance scores (the higher the better) after loading the original trojaned driver. The right bars are normalized performance scores after loading the purified driver. In the experiments with trojaned E1000+KBeast/E1000+Redir, the system crashed when executing the test case *file copy* in UnixBench. The reason is that KBeast rootkit cannot survive

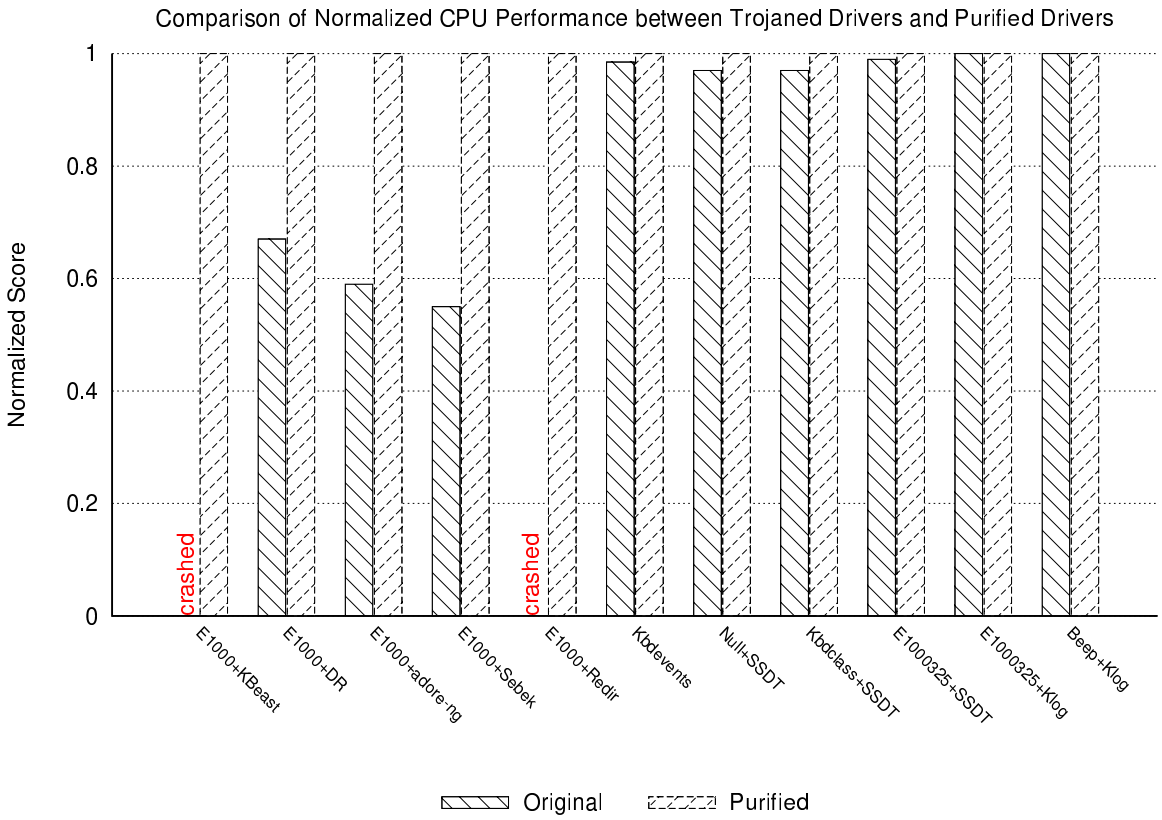

Figure 3.5.: Comparison of CPU performance

the workload of test case *file copy* in the UnixBench and both trojaned drivers contain the KBeast's code. After purification, both drivers support the benchmark successfully because the KBeast functionality has been eliminated. For the other experiments, the purified drivers improve benchmark performance by 1% to 45% compared with that under trojaned drivers. This is intuitive to understand because the purified drivers are without unnecessary kernel API invocations and thus execute less code than the trojaned drivers.

Besides testing CPU performance, we also utilize Iperf to measure the network throughput for all cases involving the NIC driver. We compare the TCP throughput of the trojaned driver with the purified driver and present the result in Figure 3.6. The left bars are bandwidths for trojaned drivers and the right bars are for purified drivers. From the results, we observe that 4 out of 7 purified drivers maintain the same or slightly better throughput

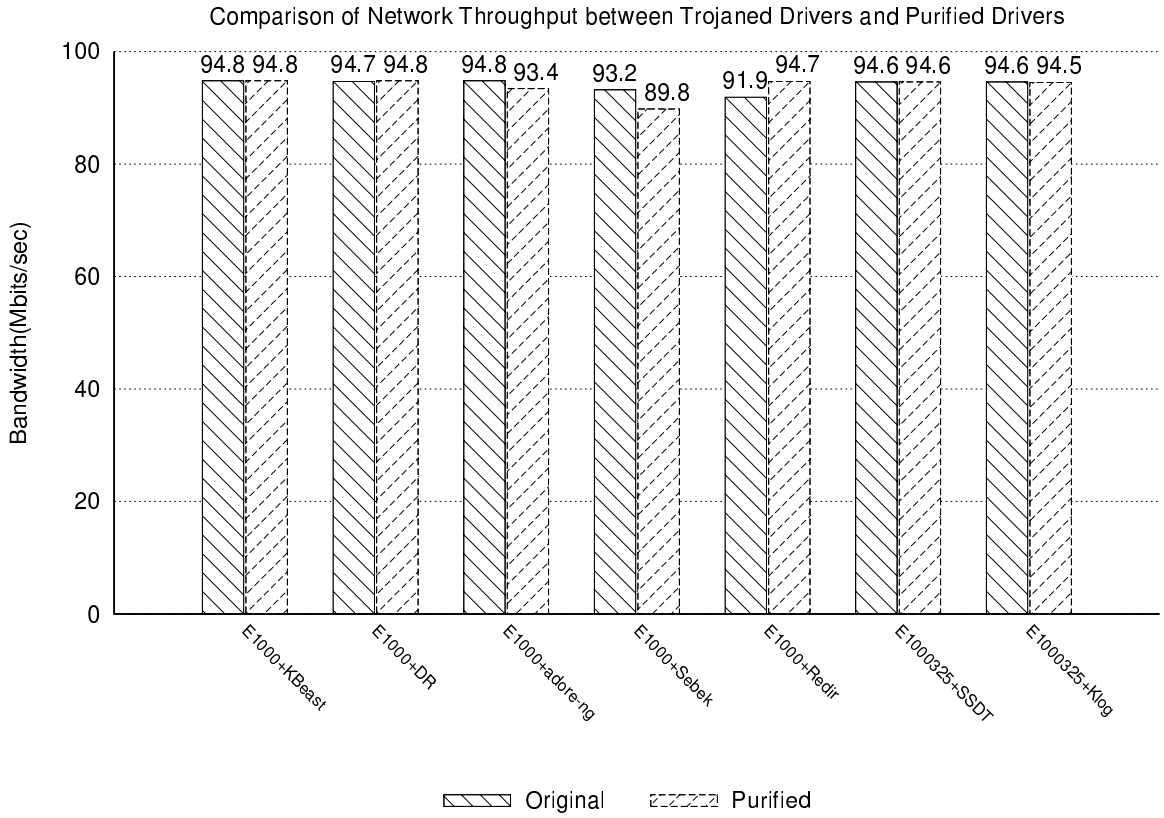

Figure 3.6.: Comparison of network throughput

compared with the trojaned drivers. The worst-case overhead observed is only 4% for the purified E1000+Sebek driver.

Our performance evaluation results demonstrate that purified drivers generated by DRIP can maintain (almost) the same network performance as under their trojaned versions. Moreover, the purified drivers lead to better CPU performance with the removal of embedded malicious operations.

3.5 Summary

We develop and evaluate DRIP to eliminate malicious/unnecessary behaviors of a trojaned kernel driver and preserve its benign functionalities for a target application. Through our evaluation, we demonstrate the effectiveness of DRIP to achieve this goal. After load-

ing a purified driver, we can maintain or even improve the system's performance compared with running the same workload under the trojaned driver.

4 FACE-CHANGE: KERNEL MINIMIZATION COMPONENT

4.1 Problem Statement

Modern operating systems strive to shrink the size of the trusted computing base (TCB) to ease code verification and minimize trust assumptions. For a general-purpose operating system like Linux, kernel minimization has already been established as a practical approach to reducing attack surface. But existing approaches [42–45] have a number of problems:

Coarse-Grained Profiling

In order to eliminate unnecessary code from the kernel, one must identify the kernel code that is required to support the multiple applications within a system. The conventional approach is to generate typical workloads and measure all active kernel code in a training session. Profiling is performed on the whole system and does not distinguish among the requirements of different applications [42]. This approach is well suited for generating a customized kernel for a static, special-purpose system (e.g., an appliance or embedded system). But for a general-purpose operating system supporting a variety of applications, whole-system profiling unnecessarily enlarges the kernel attack surface of the system.

In practice, we observe that kernel code executed under different application contexts varies drastically. Our experiments show that two distinct applications may share as little as 33.6% of their executed kernel code — thus system-wide kernel minimization would over-approximate both applications' kernel requirements. For example, the kernel functionality needed by task manager *top* is to read statistics data from the memory-based *proc* file system and write to the tty device. In sharp contrast, the *Apache* web server primarily requires network I/O services from the kernel. If we profile a system running *top* and *Apache* simultaneously, we will expose the kernel's networking code to *top* simply because *Apache* is in the same environment. Further, assume *top* is the target of a malicious attack, the

compromised *top* may be implanted with a parasite network server as a backdoor without violating the minimized kernel's constraint.

Flexibility to Adapt to Runtime Changes

The output of traditional kernel minimization approaches is a static kernel image customized for a specific workload. However, it is nearly impossible to cover all execution paths within an application's code to trigger every possible kernel request. Even when leveraging automatic test case generation techniques [35, 38, 46], profiling may still suffer from the path coverage problem for large programs. Insufficient profiling may lead to an underestimation of the kernel code required to support some application(s) at runtime. Further, the required kernel code may change when running a new application that was not profiled before or when the workload of an existing application suddenly changes. If this newly requested kernel code is not included in the customized image, the violation may crash the application or even panic the kernel.

To address these problems of whole-system-based kernel minimization, we have developed FACE-CHANGE, a component in our hypervisor-based active protection framework to support dynamic switching among multiple minimized kernels, each for an individual application. Throughout this chapter, we use the term *kernel view* to refer to the in-memory kernel code presented to an individual application. In conventional kernels, all concurrently running user-level processes share the same kernel view containing the entire kernel code section, which we refer to as a *full kernel view*. FACE-CHANGE aims to present each process with a different, customized kernel view, which is prepared individually in advance by profiling the application's needs. Any unnecessary kernel code is eliminated to minimize the attack surface accessible to this specific application. At runtime, FACE-CHANGE identifies the current process context and dynamically switches to its customized kernel view.

To support applications that were not previously profiled, we are able to profile them in independent (offline) sessions to generate their kernel views. We then load the kernel view for a new application dynamically without interrupting the system's execution. This

removes the burden of re-compiling and/or installing a new customized kernel upon the addition of a new application.

Furthermore, we include a kernel code recovery mechanism for the event that an application tries to reach code outside of the boundary of its kernel view. This may be due to incomplete profiling (e.g., interrupt handler's code with no attachment to any process or some workload not completely exercised) or malicious tampering (e.g., some injected logic requests new/different kernel features). We are able to recover the missing code and backtrack its provenance to identify the anomalous execution paths. Such capability can be leveraged by administrators to analyze the attack patterns of both user-level and kernel-level malware.

FACE-CHANGE makes the following contributions:

- A quantitative study of per-application kernel requirements in a multi-programming system.
- A hypervisor-based dynamic kernel view switching technique. FACE-CHANGE is transparent to the guest VM and requires no patching or recompilation of the guest OS kernel.
- A kernel code recovery mechanism to recover requested but missing code and backtrack the provenance of such an anomaly/exception.

The rest of this chapter is organized as follows. Section 4.2 presents the motivation, goals and assumptions of FACE-CHANGE. Section 4.3 provides the detailed design of FACE-CHANGE. Section 4.4 gives case studies on the effectiveness of FACE-CHANGE on user/kernel malware attacks and evaluates its performance. We summarize FACE-CHANGE in Section 4.5.

4.2 System Overview

In this section, we introduce a quantitative method to measure the kernel code requirements of a specific application. We then use these measurements to evaluate the similarity

of kernel code requirements between applications. The result of this quantitative study motivates the development of FACE-CHANGE. Finally, we present the goals and assumptions of our design.

4.2.1 Motivation

Each application, including both the base program and any libraries loaded into the user address space, interacts with the OS through system calls to request services (e.g., manipulating files, spawning threads, IPC, etc.). The set of system calls utilized by an application varies substantially across different application types and workloads, and intuitively, different system calls will reach different parts of the kernel's code. Further, different values passed as parameters to the same system calls may lead to totally different execution paths within the kernel. For example, because of Linux's *virtual file system* (VFS) interface, a *read* system call for disk-based files in *ext4-fs* and memory-based files in *procfs* will be dispatched to entirely different portions of the kernel's code.

To accurately measure a target application's kernel code requirements, we monitor the system execution at the basic block level. We briefly describe the profiling tool here and will present the detailed design in Section 4.3.1. We record any executed basic blocks which satisfy the following two criteria:

(i) The basic block belongs to the kernel, i.e., its memory address is in kernel space.

(ii) The basic block is executed in the target application's context.

After merging any adjacent blocks, we get a range list $\mathrm{K}_{[\mathrm{app}]}$ for a target application (denoted by subscript $[\mathrm{app}]$) in the form:

$$\mathrm{K}_{[\mathrm{app}]} = \{([\mathrm{B}_1, \mathrm{E}_1], \mathrm{T}_1), \cdots, ([\mathrm{B}_\mathrm{i}, \mathrm{E}_\mathrm{i}], \mathrm{T}_\mathrm{i})\}$$

B_i and E_i denote the beginning and end addresses for the i-th in-memory code segment. T_i indicates the type for this memory segment, where T_i can be either "base kernel" or the

name of a kernel module. For kernel modules, we record addresses relative to the module's base address because a module's loading addresses may change at runtime.

We introduce three definitions for comparing two distinct application's kernel code requirements:

1. $\mathrm{K}_{[\mathrm{app1}]} \cap \mathrm{K}_{[\mathrm{app2}]}$

   The intersection of two range lists outputs the overlapping address ranges between them. The result is still a range list.

2. $\textsc{len}(\mathrm{K}_{[\mathrm{app}]})$

   The LEN of a range list outputs the number of elements in this list.

3. $\textsc{size}(\mathrm{K}_{[\mathrm{app}]}) = \sum_{\mathrm{i} \in [1, \textsc{len}(\mathrm{K}_{[\mathrm{app}]})]} (\mathrm{E_i} - \mathrm{B_i})$

   The SIZE of a range list outputs the size of kernel code in this range list.

We use Equation (4.1) below to define the similarity index *S* between $\mathrm{K}_{[\mathrm{app1}]}$ and $\mathrm{K}_{[\mathrm{app2}]}$:

$$\mathrm{S} = \frac{\textsc{size}(\mathrm{K}_{[\mathrm{app1}]} \cap \mathrm{K}_{[\mathrm{app2}]})}{\textsc{max}(\textsc{size}(\mathrm{K}_{[\mathrm{app1}]}), \textsc{size}(\mathrm{K}_{[\mathrm{app2}]}))} \tag{4.1}$$

A similarity index *S* indicates the proportion of the overlapping of kernel code required between two applications. Besides common system call execution paths, the overlapping kernel code also consists of functionality needed by every application, e.g., process scheduler and interrupt handling code. Through the profiling of well-known Linux applications, we find that similarity indices range from 33.6% for applications that are orthogonal in type (such as *top* vs. *Firefox*) to 86.5% for similar applications (such as *Apache* vs. *vsftpd*). Table 4.1 (Section 4.4) shows the similarity indices for all profiled applications. These measurements support our earlier hypothesis that kernel code execution paths vary substantially across different application types. This also indicates that application-specific kernel views can minimize the kernel attack surface far beyond that of system-wide kernel minimization.

4.2.2 Goals and Assumptions

We state the goals for our system in four aspects: strictness, robustness, transparency and flexibility.

Strictness: The kernel view generated for a specific application should only contain the kernel code that is necessary for the correct execution of this application under a normal usage scenario. We should eliminate all other excessive code from the kernel view to avoid enlarging the kernel's attack surface. If an application reaches kernel code that does not belong to its kernel view, we should record the access in detail for later analysis.

Robustness: If an application is running under the same workload and same usage scenario as during profiling, the behavior of this application running with a customized kernel view should be no different than with a full kernel view. If the application accesses any kernel code that is not included in the customized kernel view, we should recover the missing code and record this violation silently without being detected by the application.

Transparency: There is no need to change any code in the applications or operating system. The hypervisor controls all FACE-CHANGE operations, which remain transparent to the guest VM.

Flexibility: Administrators can dynamically load, unload, and switch the kernel view for a specific application at any time. This should neither jeopardize the functionality of the currently running application nor the system as a whole.

We assume that, when we generate customized kernel views in the profiling phase, the environment, including both the applications and the kernel, should not be tampered with by malware.

4.3 Design and Implementation

In this section, we give a detailed description of the overall design of FACE-CHANGE, highlight the challenges we face and the solutions we propose. Then we discuss the detailed implementation of our prototype system.

We divide the whole system into two phases in chronological order: the profiling phase and the runtime phase. The profiling phase monitors a target program's execution and, based on the active kernel code in this process' context, generates a configuration file describing the application's customized kernel view. In the runtime phase, FACE-CHANGE builds a new customized kernel view based on each application's configuration file and forces the process to use this customized kernel view whenever the guest OS schedules it.

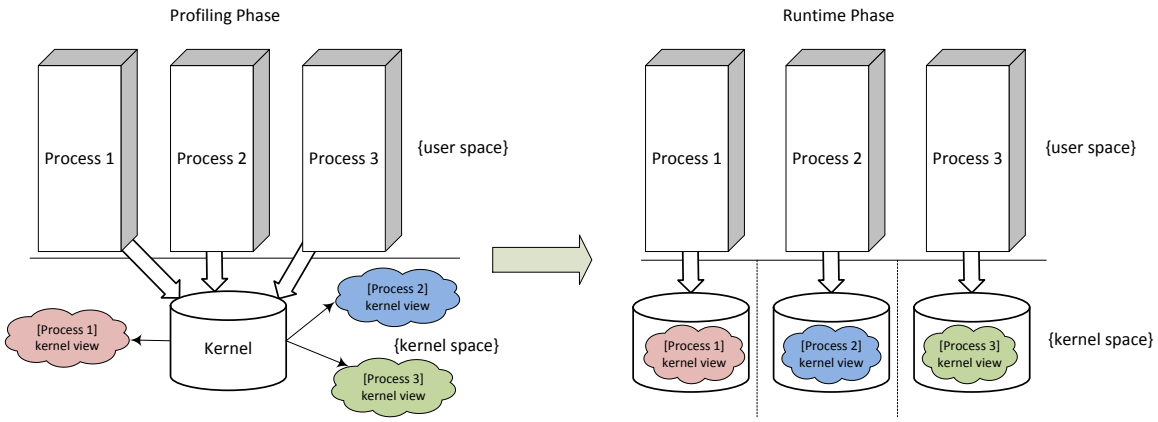

Figure 4.1.: Overview of FACE-CHANGE

Figure 4.1 shows a high-level example of these two phases. Assume we want to profile *Process 1* in the profiling phase. When the kernel schedules *Process 1* to run, we start to record all the kernel code executed in its context. When *Process 1* is scheduled out, we pause the recording until the process is re-scheduled. This procedure also applies to *Processes 2* and *3*. At last we generate three configuration files for the kernel views of these three processes respectively. In the runtime phase, we load each customized kernel view for the corresponding process. For example, *Process 1* can only access *[Process 1] kernel view* when it is running.

4.3.1 Profiling Phase

Design of the Profiler

We implemented our profiler as a component of the *QEMU* [16] 1.6.0 full system emulator. This enables the profiler to track an application's execution at the granularity of a basic block. We use VMI techniques to track context switches within the guest OS. When the guest OS schedules the target application, the profiler records any address ranges of kernel code executed in this process' context. For code within a kernel module, we record addresses relative to the module's base address. Once the application has been sufficiently profiled, the profiler exports all recorded kernel code segments to a kernel view configuration file.

Test Suite Selection

For each application to be profiled, the user should choose a test suite to simulate the expected real-world workload for this application. For instance, when profiling a server application, the user may deploy it in the real environment to handle requests, or for an interactive application, one may simulate the I/O operations of a typical user. To give a specific example, when profiling a *mysql* server, we set up a RUBiS[1] [47] server and used its own simulated client to generate workloads for the *mysql* database.

It is difficult to ensure that all code paths through an application are executed during profiling, thus it is possible that at runtime the application may access some kernel code missed by the profiling phase. One alternative to a test suite driven profiler is to use symbolic execution to generate high-coverage test cases, but this approach may not scale to large applications. To address this problem, we employ a kernel code recovery mechanism in the runtime phase to recover any missing kernel code. We explain this mechanism in detail in Section 4.3.2.

[1]RUBiS is an ebay-like auction service that heavily uses *mysql*.

Interrupt Context

In modern OS kernels, hardware triggered asynchronous interrupts can happen at any time, and thus interrupt handler code is not attached to any single process' context. We choose to include the interrupt handler's code in every application's kernel view to avoid having to repeatedly recover this code at runtime. Our profiler leverages *QEMU* to identify the occurrence of an interrupt. If this interrupt is not a software interrupt (such as a system call), we can infer that the system has entered interrupt context. At this point, we record all kernel code addresses accessed in the interrupt's context for use in all applications' customized kernel view.

4.3.2 Runtime Phase

We describe the general design of the runtime phase in Algorithm 3 and discuss some interesting features below in detail.

Kernel View Initialization

When loading a new kernel view configuration, FACE-CHANGE allocates memory pages for both the base kernel code and any kernel modules' code and fills them with undefined instruction (UD2) "*0xf 0xb*" (UD2 will raise an invalid opcode exception when executed). FACE-CHANGE then loads the kernel code specified in the kernel view configuration into it's appropriate locations in the new pages. Recall that during profiling, we track the kernel control flow at the basic block level. However, rather than loading individual basic blocks, we slightly relax the condition to load the entire kernel function which contains the valid basic blocks. The rationales for this relaxation are: (1) The adjacent code within the same kernel function is more likely to be accessed at runtime. Thus, we can reduce the frequency of kernel code recovery by loading the whole kernel function. (2) UD2 is a 2-byte instruction. If an address range in the kernel view configuration starts from an odd-numbered address, only the first byte of UD2 will be in the kernel view; therefore, the

Algorithm 3 Kernel View Switching/Kernel Code Recovery

Input: modulelist ← kernel module list
context_switch_addr ← Address of context_switch function
resume_userspace_addr ← Address of resume_userspace function
full_kernel_view_index ← Index of full kernel view

```
- - - - - - - - Kernel View Switching - - - - - - - - - -
procedure SWITCH_BASE_KERNEL(index)
    kernel_range ← GET_KERNEL_RANGE()
    LOAD_KERNEL_VIEW_EPT(kernel_range, index)
procedure SWITCH_KERNEL_MODULES(index)
    for all mod in modulelist do
        module_range ← GET_MODULE_RANGE(mod)
        LOAD_MODULE_VIEW_EPT(module_range, index)
procedure SWITCH_KERNEL_VIEW(index)
    SWITCH_BASE_KERNEL(index)
    SWITCH_KERNEL_MODULES(index)
procedure HANDLE_KERNEL_VIEW_TRAP(vcpu)
    if vcpu.rip = context_switch_addr then
        procinfo ← READ_PROC_INFO(vcpu)
        index ← KERNEL_VIEW_SELECTOR(procinfo)
        if index = full_kernel_view_index then
            CLEAR_RESUME_USERSPACE_TRAP()
            SWITCH_KERNEL_VIEW(index)
        else
            ENABLE_RESUME_SPACE_TRAP()
            lastindex ← index
    else if vcpu.rip = resume_userspace_addr then
        CLEAR_RESUME_USERSPACE_TRAP()
        SWITCH_KERNEL_VIEW(lastindex)
- - - - - - - - Kernel Code Recovery - - - - - - - - - -
procedure BACK_TRACE(rip, rbp)
    iter_rbp ← rbp
    prev_rip ← rip
    while IS_VALID(prev_rip) do
        DUMP_BACKTRACE(prev_rip)
        prev_rip ← READ_PREV_RIP(iter_rbp)
        prev_rbp ← READ_PREV_RBP(iter_rbp)
        if PREV_RIP = "0B 0F" then
            RECOVER_BACKTRACE(prev_rip)
        iter_rbp ← prev_rbp
procedure HANDLE_INVALID_OPCODE(vcpu)
    BACK_TRACE(vcpu.rip, vcpu.rbp)
    mem_page ← GET_MEMORY_PAGE(vcpu.rip)
    start_addr ← SEARCH_BACKWARDS(vcpu.rip)
    end_addr ← SEARCH_FORWARDS(vcpu.rip)
    FETCH_FILL_CODE(page, start_addr, end_addr)
```

processor may misinterpret the fragmented UD2 as a different instruction. Loading entire kernel functions avoids this problem because the boundaries of kernel functions are aligned on powers-of-two[2].

To identify function boundaries, we search for a function header signature backwards and forwards from the basic blocks marked in the kernel view configuration. For example, a common function header signature in the x86 Linux kernel is "*push ebp; mov ebp, esp*"(binary opcodes "*0x55 0x89 0xe5*"). There is a possibility that one kernel function may cross two memory pages and further, one single instruction may split across pages. In this case, we continue searching from the head of the next page or the tail of the previous page to locate the complete kernel function.

After all of the kernel view's code is identified and loaded into the new pages, FACE-CHANGE redirects any kernel code access made by this application to the customized kernel view. We implement our FACE-CHANGE runtime component within a *KVM* hypervisor (i.e., kvm-kmod-3.6 and qemu-kvm-1.2.0) and leverage Extended Page Tables (EPT) to manipulate kernel code mappings. When using EPT, the guest VM maintains its own page table to translate guest virtual addresses to guest physical addresses. The hypervisor then uses EPT to transparently map the guest physical addresses to host physical addresses. During guest OS context switches, FACE-CHANGE changes the page table entries in the EPT to direct any kernel code accesses to the customized kernel view for the application (instead of the original kernel's code). This procedure is explained in the Section 4.3.2.

Again, FACE-CHANGE must take care when handling kernel modules' code in a customized kernel view. Recall that kernel modules are dynamically loaded at runtime in the kernel's heap, and thus, during the profiling phase, we record these addresses relative to the module's base address. Before we load modules' code into a kernel view, we traverse the kernel's module list to identify the loading addresses for any modules marked in the kernel view configuration. Then we load the valid kernel code in the code pages for the kernel modules.

[2]Linux kernel is by default compiled with -O2 that contains optimization flag -falign-functions

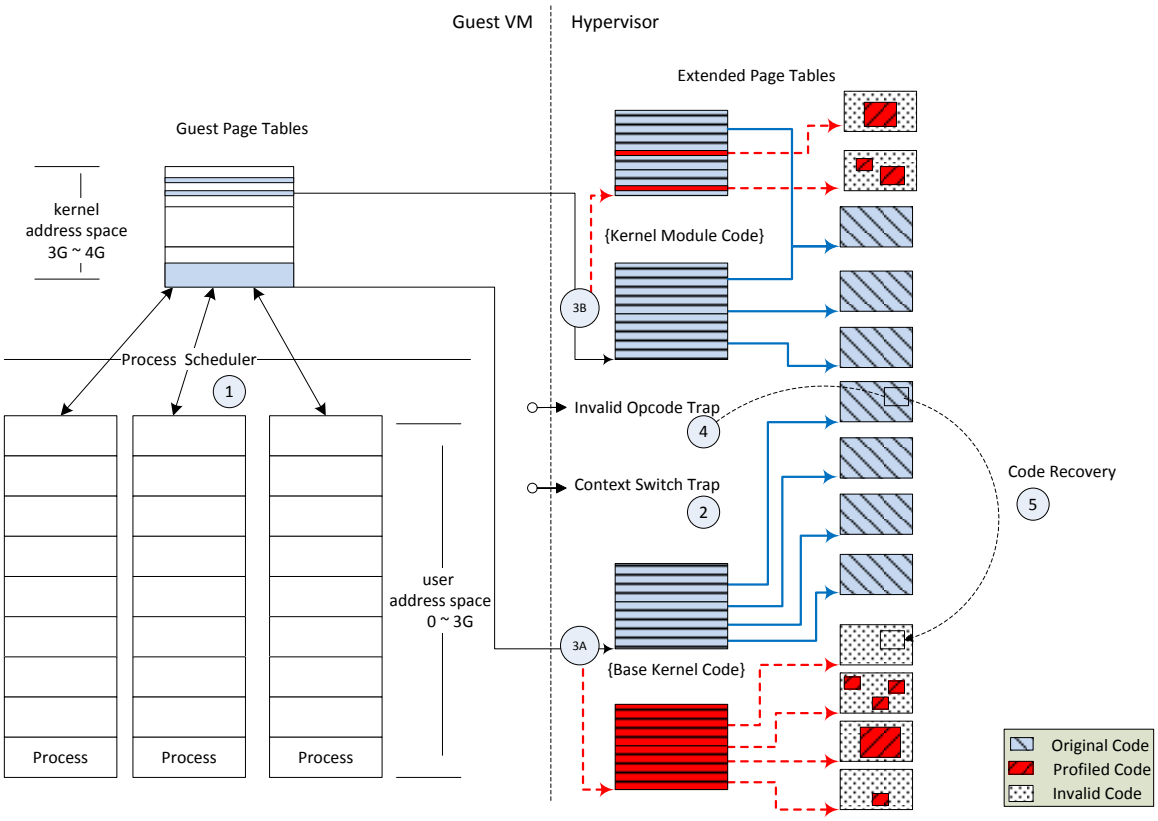

Figure 4.2.: The procedure of dynamic kernel view switching and kernel code recovery

Kernel View Switching

Figure 4.2 illustrates each step of the kernel view switching procedure. In step 1, the guest OS chooses a process to run and prepares to context switch to the new process. In step 2, using VMI, we intercept this context switch and determine which customized kernel view is needed for the new application. In step 3A and 3B, we modify the pointers to the page directory (level 2 in the EPT) corresponding to the base kernel code and all kernel modules' code respectively. Because kernel modules' code pages are scattered in the kernel heap, we reuse any entries in the page directory that point to kernel data and only modify the entries pointing to the modules' code.

We also develop a set of optimizations to improve performance. Through our experimentation, we find that switching kernel views immediately at context switches may cause

the application to miss interrupts, and thus jeopardize I/O performance. We choose instead to switch kernel views when the code resumes user space execution after the context switch. This will still satisfy the strictness goal (minimize the attack surface) and mitigate the performance degradation caused by missed interrupts. We also check whether the previous process and the next process use the same kernel view, and if so, we can avoid one additional kernel view switch.

Kernel Code Recovery

There are two situations where FACE-CHANGE may need to recover missing kernel code:

(i) An incomplete kernel view generated during profiling: Testing in a controlled runtime environment without introducing any attacks, we find that the majority of the benign kernel recoveries are triggered due to missing code for handling interrupts. For example, *KVM* provides a para-virtualized clock device to the guest VM. This *KVM* specific code cannot be included in the kernel view during the profiling in *QEMU*. Thus, at runtime, FACE-CHANGE needs to recover the missing kernel functions shown below in chronological order:

$$kvm\_clock\_get\_cycles \rightarrow kvm\_clock\_read$$
$$\rightarrow pvclock\_clocksource\_read \rightarrow native\_read\_tsc$$

In addition, interrupt handling code is not bound to any process and can be triggered by hardware interrupts at any time. In the profiling phase, we may not observe all possible interrupts for this application. Before missing code recovery, we inspect the current call stack to determine whether the current execution is in interrupt context (through backtracking the current function traces). Thereafter we recover the missing kernel code to correctly handle those interrupts.

All other benign kernel code recoveries due to incomplete profiling of the application's execution paths are recorded as a reference for the administrator to ameliorate the profiling test suite.

(ii) Anomalous execution caused by malicious attacks: User level malware may hijack a normal process to execute shellcode which requests kernel services that are not in the customized kernel view. Additionally, kernel level rootkits can detour the kernel's execution path to their payload's malicious code, and obviously, this malicious payload will not be in any application's kernel view. FACE-CHANGE is designed to report the suspicious execution traces, but still recover the kernel code in this case. In order to track the provenance of the attack, we not only record any recovered functions, but also backtrack the anomalous execution's call stack to find the origin of the invocation chain for later analysis.

As we mentioned in Section 4.3.2, we fill any kernel code space that is not in the kernel view with UD2 "*0xf 0xb*". When executed, UD2 raises an invalid opcode exception which causes a trap to the hypervisor. We illustrate this as step 4 *invalid opcode trap* in Figure 4.2. After intercepting the trap, we check the faulting address and try to fetch the missing kernel function from the original kernel code pages (step 5 in Figure 4.2).

During our implementation of the kernel code recovery mechanism, we fixed an interesting cross-view bug in FACE-CHANGE that is worth mentioning here. If no customized kernel view is enabled for a specific process, it will have a full kernel view. When executing this process, its kernel execution may be interrupted or the process may voluntarily give up the CPU. If we enable a customized kernel view for that process at this time and the process is re-scheduled by the kernel, some functions in the process' execution stack may not be in the new kernel view. We give an example of this situation in Figure 4.3. In this case, the process is re-scheduled while executing *pipe_poll* at address *0xc0211370*. The invocation chain in the stack is as follows:

$$syscall\_call \rightarrow sys\_poll \rightarrow do\_sys\_poll$$

We find that because *sys_poll* and *do_sys_poll* are not in the new customized kernel view, their code regions are filled with UD2 (shown in red). If we recover *pipe_poll* and return to its caller (*do_sys_poll*), the process will execute undefined instructions. For *do_sys_poll* this is not a problem because the return address (*0xc021a526*) is an even number. Execution will return to the "*0xf 0xb*" opcode (UD2), causing an invalid opcode trap, and we can recover *do_sys_poll* as normal. We call this as a *lazy recovery*. But for *sys_poll*, the return address (*0xc021a759*) is an odd number; therefore, the opcode starting at address *0xc021a759* is "*0xb 0xf*." This opcode will be misinterpreted by the processor and not cause a trap. Our solution is, during code recovery, to backtrack the stack and recover any caller whose return-target opcode starts with "*0xb 0xf*" in the new kernel view. We call this *instant recovery*. In this example, when we recover the code for *pipe_poll*, we recover *sys_poll* instantly.

Disable Customized Kernel View

We can load/unload customized kernel views dynamically at runtime to satisfy our flexibility goal. When we disable a kernel view, FACE-CHANGE de-allocates all memory pages for that kernel view and switches the EPT back to a full kernel view without interrupting the running application. This enables us to adapt to an altered environment smoothly by "hot-plugging" kernel views.

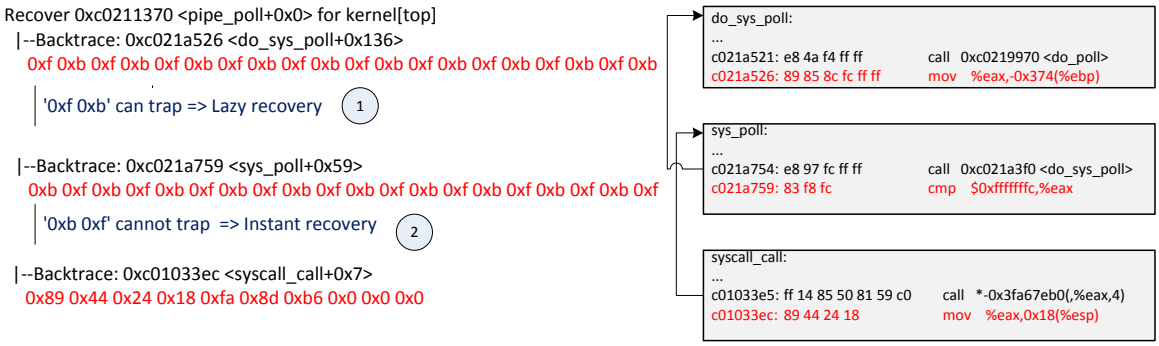

Figure 4.3.: Cross-View kernel code recovery

4.4 Evaluation

In this section, we present the evaluation of FACE-CHANGE in two aspects: security and performance. For the security evaluation, we first use the similarity index to measure the similarities of kernel views among applications. Then we demonstrate the effectiveness of our system to track attack provenance of both user-level malware and kernel-level rootkits. For the performance evaluation, we measure the overall system performance with FACE-CHANGE enabled and the I/O performance for an *Apache* web server with a minimized kernel view. The hardware configuration of our testing platform is a Lenovo Ideapad U410 with Intel® Core™ i7 3.10GHz and 8GB memory. We run FACE-CHANGE on Linux Mint 13 x86_64 (Linux kernel version 3.5.0). We test our prototype with a guest VM using Ubuntu 10.04 (Linux kernel version 2.6.32) i386 LTS release[3], further since FACE-CHANGE requires minimal domain knowledge, it will be convenient to extend our current system to support more Linux kernel versions with only minor changes to the implementation. The guest VM's memory is 2GB and it uses bridged networking.

4.4.1 Security Evaluation

Kernel View Variation among Applications

We use the similarity index defined in Section 4.2 to measure the difference of kernel views among 12 well-known Linux applications from different categories. For example, *Apache* and *vsftpd* are server applications that handle network requests. *Firefox* and *gvim* are interactive applications that respond to user input. We present the profiling results as a square matrix in Table 4.1. The main diagonal($\searrow$) of the matrix is marked with gray cells. Each cell on the main diagonal presents the size of the kernel view for this specific application (e.g., *Vsftpd* executes 341KB kernel code in the profiling phase). We compare the kernel code address ranges between every two applications to get the overlapping size. All entries above the main diagonal represent the overlapping size between two ap-

[3]We use Ubuntu 10.04 because the kernel rootkit samples we use in the evaluation do not support newer Linux kernel yet.

Table 4.1.: Similarity matrix for applications' kernel views

	firefox	totem	gvim	apache	vsftpd	top	tcpdump	mysqld	bash	sshd	gzip	eog
firefox	443KB	275KB	251KB	302KB	284KB	149KB	218KB	305KB	221KB	316KB	213KB	286KB
totem	62.1%	286KB	239KB	210KB	217KB	140KB	166KB	228KB	196KB	220KB	174KB	257KB
gvim	56.7%	83.6%	262KB	206KB	206KB	142KB	160KB	220KB	190KB	211KB	166KB	247KB
apache	68.2%	62.7%	61.5%	335KB	284KB	141KB	210KB	265KB	203KB	292KB	200KB	215KB
vsftpd	67.9%	63.6%	60.5%	83.5%	341KB	145KB	208KB	272KB	205KB	293KB	206KB	222KB
top	33.6%	49.2%	54.2%	42.2%	42.7%	167KB	135KB	138KB	147KB	153KB	121KB	143KB
tcpdump	49.2%	58.0%	61.1%	62.6%	61.0%	57.6%	234KB	203KB	165KB	216KB	169KB	168KB
mysqld	68.7%	68.1%	65.4%	78.9%	79.8%	41.1%	60.5%	336KB	186KB	260KB	212KB	230KB
bash	50.0%	68.7%	72.6%	60.6%	60.1%	60.8%	68.3%	55.5%	242KB	223KB	158KB	215KB
sshd	71.3%	58.4%	55.9%	77.5%	77.7%	40.5%	57.3%	68.9%	59.0%	378KB	216KB	233KB
gzip	48.1%	60.9%	63.4%	59.6%	60.4%	49.5%	69.0%	63.2%	64.6%	57.1%	245KB	177KB
eog	64.6%	86.5%	83.2%	64.2%	65.2%	48.1%	56.5%	68.7%	72.4%	61.7%	59.7%	297KB

plications' kernel views (e.g., *tcpdump* and *Firefox* have 218KB overlapping kernel code). Entries below the main diagonal represent the similarity index calculated using Equation (4.1) in Section 4.2. The similarity index demonstrates the similarity of kernel attack surface between different applications. For applications of different types, lower percentages are better as this ensures a distinct minimized kernel in both cases, and for similar applications high percentages are expected since both require similar kernel services. As Table 4.1 shows, the similarity indices range from 33.6% for dissimilar applications to 86.5% for applications with common kernel requirements. This proves our intuition that if two applications are from different categories they have relatively low similarity index and leverage different parts of the kernel.

Attack Detection and Provenance

Because the kernel attack surface for each individual application is reduced according to the profiling results, we can reveal malicious attack patterns whenever a process goes beyond the boundary of its kernel view. Further, we backtrack the requested kernel code to identify the exact attack provenance.

This result is a step further than traditional system-wide kernel minimization techniques [42–45] because FACE-CHANGE is able to detect anomalous execution based on an individual application's kernel view. To demonstrate that FACE-CHANGE can reveal attack evidences that may go unnoticed under traditional system-wide minimization techniques, we also create a "union" kernel view (the union of all kernel views from the applications we have profiled) as the system-wide minimized kernel. System-wide minimization may fail to detect an attack if the attack utilizes kernel code required by any application in the system. FACE-CHANGE greatly reduces this "blind spot" because it is able to detect kernel execution anomalies specific to a single application.

In this chapter, we evaluate the effectiveness of attack detection with 13 user-level malware (8 of them use online runtime infection and 5 use offline binary infection) and 3

Table 4.2.: Results of security evaluation against a spectrum of user/kernel malware

Name	Infection Method	Payload	Note
Injectso	Online infection	UDP server	Case study I
Cymothoa v1	Online infection	Bind /bin/sh to TCP port and fork shell	Recover *sys_fork* and TCP server
Cymothoa v2	Online infection	Bind /bin/sh to TCP port and fork shell	Recover *sys_clone* and TCP server
Cymothoa v3	Online infection	Remote file sniffer	Recover *sys_settimer* and signal handler
Cymothoa v4	Online infection	Single process backdoor	Case study II
Hotpatch	Online infection	File writing of injecting timestamp	Recover injection and file writing procedure
Xlibtrace	Online infection	Tracking function invocation	Recover tty procedures on terminal
Hijacker	Online infection	Redirection of library function	Recover the procedure of hijacking
Infelf v1	Offline infection	Remote shell server	Recover remote shell socket operations
Infelf v2	Offline infection	Register dumping	Case study III
Arches	Offline infection	Register dumping	Recover register dumping operations on terminal
Elf-infector	Offline infection	Register dumping	Same as above
ERESI	Offline infection	UDP server	Recover creation of udp server
KBeast	Kernel rootkit	File/Process hiding, keystroke sniffer	Case study IV
Sebek	Kernel rootkit	Confidential data collection	Recover kernel code in sebek module
Adore-ng	Kernel rootkit	File/Process hiding	Recover kernel code in adore-ng module

kernel-level rootkits. This data is presented in Table 4.2. We highlight four of these attack case studies in detail.

Case Study I — Injectso

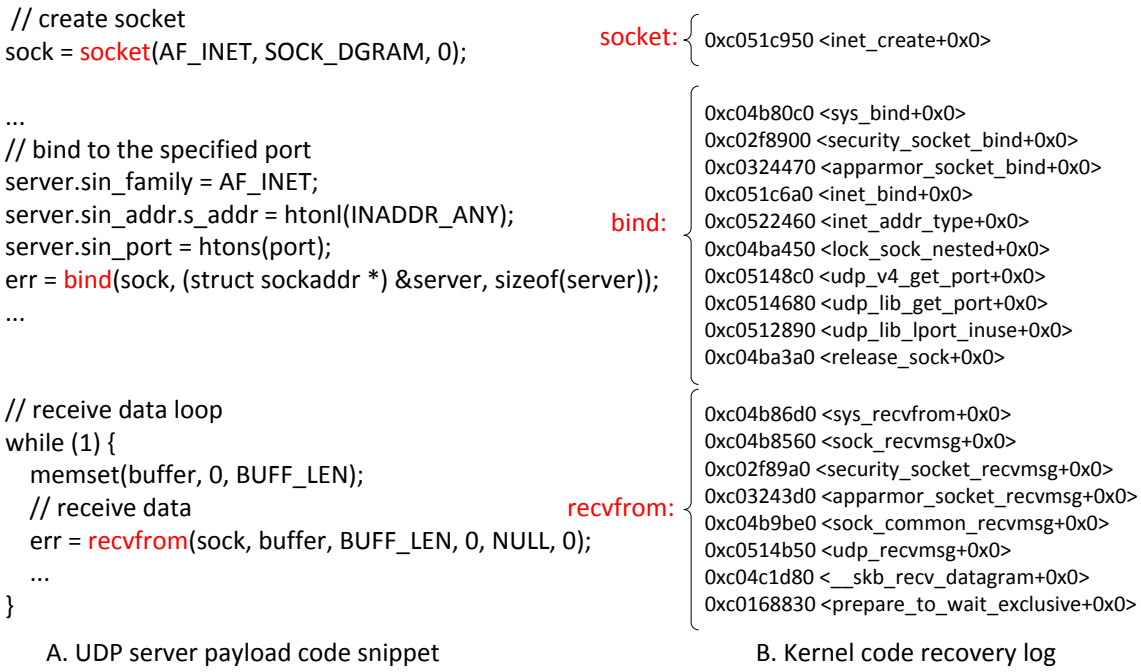

Figure 4.4.: The attack pattern of an *injectso*'s payload

Injectso [48] is a well-known hot-patching tool used to modify the behavior of a running process by injecting a dynamic shared object into its address space. It detours the current instruction pointer to *__libc_dlopen_mode* and builds a fake stack to invoke the shared object's code. The shellcode's payload is a UDP server, and the target program is *top*. Obviously, the kernel view for *top* does not contain any kernel code needed to run a UDP server (even if the kernel views of other co-existing applications do), and thus *Injectso*'s payload triggered the kernel code recovery mechanism.

From the kernel code recovery log, we can precisely identify the anomalous execution caused by *Injectso* in the *top* process. In Figure 4.4, we present the UDP server payload's code and the corresponding kernel code recovery log. The UDP server will create a socket,

bind to an address/port, and receive data using the C library calls *socket*, *bind* and *recvfrom* respectively. It is straightforward to identify which library functions correspond to the recovered kernel code sections (e.g., *bind* executes a kernel code path from *sys_bind* to *release_sock* [4] in chronological order).

We test the system again and apply the "union" kernel view, which includes both *top* and some network applications (such as *Firefox* and *Apache*) to represent a system-wide minimization technique. These network applications require the same kernel networking code as the UDP server payload, and thus this case results in no UDP related kernel functions being recovered. Due to the enlarged attack surface of the system-wide minimized kernel, this attack would achieve its goal with the available kernel code and thus go undetected.

Case Study II — Cymothoa

Cymothoa [49] is a shellcode injection framework that uses different infection methods and payload types. The parasite executable coexists with the host process stealthily while the host process continues to work properly. We test all four working parasites introduced in the article "Single Process Parasite" [50] in Phrack issue 68 and successfully reveal all four attack behaviors. The parasite uses the *sys_fork* and *sys_clone* system calls to create a child process/thread to execute its payload. Later variants are more stealthy, utilizing *settimer* and *signal* to schedule the shellcode inside the host process. Here, we give a detailed description of the most stealthy (variant 4) parasite's control flow. This variant creates a backdoor parasite living within another process (*bash* is the target program in this case). First the shellcode registers a signal handler for the *SIGALRM* signal. Then it opens a nonblocking I/O socket, binds it to a specific port, and sets the *SIGALRM* timer. When the *SIGALRM* signal is handled, the parasite accepts any connection on the socket and launches a remote shell. The parent then sets the timer again and resumes execution of the host process.

[4] Symbols of kernel functions are not necessary for backtracking. We use them here for clear demonstration.

Again, the kernel code executed by the shellcode's actions, e.g., setting the signal handler, creating the TCP server, and setting the alarm clock are recorded in the kernel recovery log. This reveals both the infection method and payload behaviors of the stealthy parasite. Also, like before, existing kernel minimization techniques may fail to detect this attack entirely because other applications will likely add these kernel regions into the union-based minimized kernel.

Case Study III — Infelf

In addition to runtime infection malware, we also apply our techniques to detect compromised applications. *Infelf* [51] is an offline binary infection tool that is able to implant trojan code into an existing binary program. It splits trojan code into multiple instruction blocks, inserts them into free alignment areas between functions, and concatenates their execution path with jump instructions. We use this tool to implant a hardware register printing function into the *gvim* binary and redirect *gvim*'s entry function to this shellcode. During *gvim*'s startup, FACE-CHANGE recovers numerous TTY kernel functions which are not included in *gvim*'s kernel view. Again, in this case, a whole-system kernel minimization technique would be unable to detect this attack on a system containing both *gvim* and terminal applications that require the kernel's TTY functions (such as *tcpdump* or *bash*).

Case Study IV — KBeast Rootkit

In addition to user-level attacks, our system is also able to detect rootkit attacks at the kernel level. Because rootkit attacks originate from shellcode in kernel space, the interpretation of kernel recovery logs is different from user-level attacks. Kernel-level attacks aim to hide their malicious behavior by detouring the kernel's control flow during execution of certain kernel routines (e.g., listing kernel modules, network connections, etc.). Again, we assume that no rootkit is present during the initial profiling phase, and so no rootkit code can be included in the kernel view configuration files. When FACE-CHANGE allocates a new kernel view, if a rootkit has already been installed in the runtime system's kernel, the

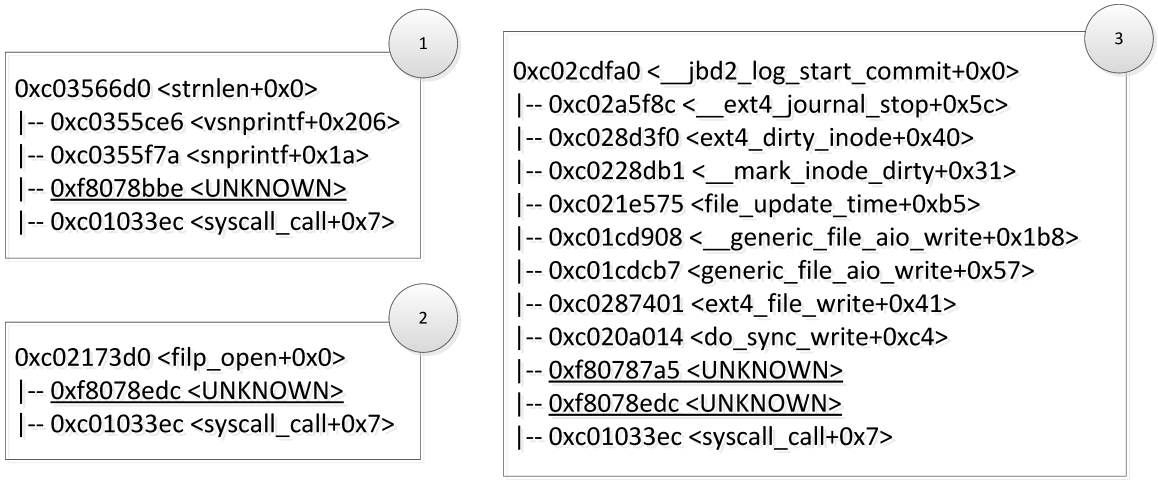

Figure 4.5.: The attack pattern of a *KBeast* rootkit

rootkit's code will not be loaded into the new view and will be filled with UD2 by default. If the application later triggers FACE-CHANGE'S code recovery, the log will allow us to clearly see where the hijack took place. A more complicated scenario that FACE-CHANGE can detect is a rootkit which is installed while FACE-CHANGE is enforcing an application's kernel view. In this scenario, the rootkit will be detected in the same way as user-level malware: by the kernel functionality that it requests to perform its malicious functionalities. Again, this code will be recovered and we can backtrace recovered kernel code to reveal the anomalous execution.

We use the KBeast [52] rootkit as an example to show this process in detail. KBeast is a new rootkit that inherits many features from traditional Linux kernel rootkits (e.g., file/process/socket/module hiding, keystroke sniffer) and it supports recent kernel versions. We use the kernel view for the *bash* program to detect the existence of KBeast. All the keystrokes typed in bash are processed by the keyboard event handler. KBeast is able to intercept and read the keystrokes and store this data into a hidden file, and it will hide its existence by removing itself from the kernel module list. In Figure 4.5, by backtracking the recovered kernel functions, we find code addresses with an *UNKNOWN* tag. This indicates that these memory addresses are not in any identified memory regions. We also find that KBeast's code hijacks the entries of some system calls and invokes *strnlen* to check the

length of the keystroke buffer, *filp_open* to open the hidden file, and *do_sync_write* to write the keystroke data into this file.

4.4.2 Performance Evaluation

System Performance

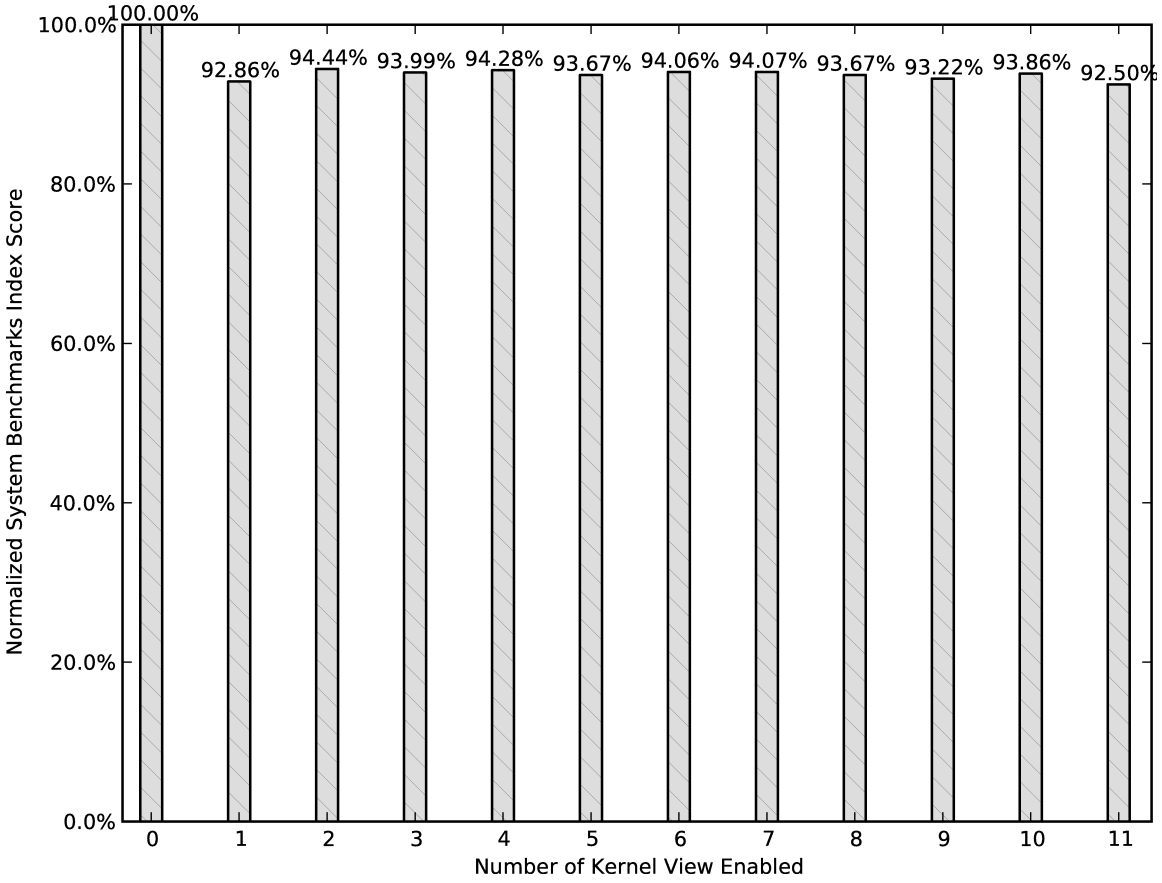

Figure 4.6.: Normalized system performance results from UnixBench

We use the UnixBench benchmark suite to measure and evaluate system performance. Specifically, we take three different measurements:

(i) We run UnixBench without enabling FACE-CHANGE to get a baseline result.

(ii) We enable FACE-CHANGE, load one kernel view (*Apache*), and run the benchmark. This tests whole-system performance overhead after enabling our system.

(iii) Next, we launch the applications[5] from Table 4.1 and load their kernel views one at a time. After each kernel view is loaded, we rerun the benchmark. This measures any performance influence on the whole system after loading multiple kernel views.

In Figure 4.6, we normalize the performance scores of the UnixBench (higher performance score indicates better performance) based on the baseline score from step 1. The x axis represents the number of kernel views we enabled simultaneously. We find that, compared to the baseline result, enabling our system incurs 5%∼7% performance overhead on the whole system. Adding multiple kernel views incurs trivial impact on the system performance. We find that the only performance degradation occurs during the subtest *Pipe-based Context Switching* of UnixBench. This is not surprising because FACE-CHANGE triggers additional traps for each context switch. We could largely minimize the performance overhead with optimization of the context switch handler's code.

I/O Performance for *Apache*

We also evaluate FACE-CHANGE's influence on application's I/O performance. Specifically, we use *httperf* to compare *Apache*'s performance before and after enabling FACE-CHANGE. In this test, we increase the request rate from 5 to 60 requests per second (100 connections in total) to test the I/O performance. We present the ratio of the I/O throughput after enabling FACE-CHANGE to before in Figure 4.7. From Figure 4.7, we find that I/O throughput will not be affected below the threshold rate of 55 reqs/second but may begin to degrade afterwards. This indicates that our system has no influence on the network throughput before the CPU becomes a bottleneck. The reason is that the bursts of network traffic cause frequent kernel view switching in a short period of time. One solution is to measure the rate of requests for an expected workload of the server before enabling FACE-CHANGE.

[5]We exclude *gzip* here because it is not a long running application (i.e. it is difficult to ensure it executes during the entire benchmarking measurement).

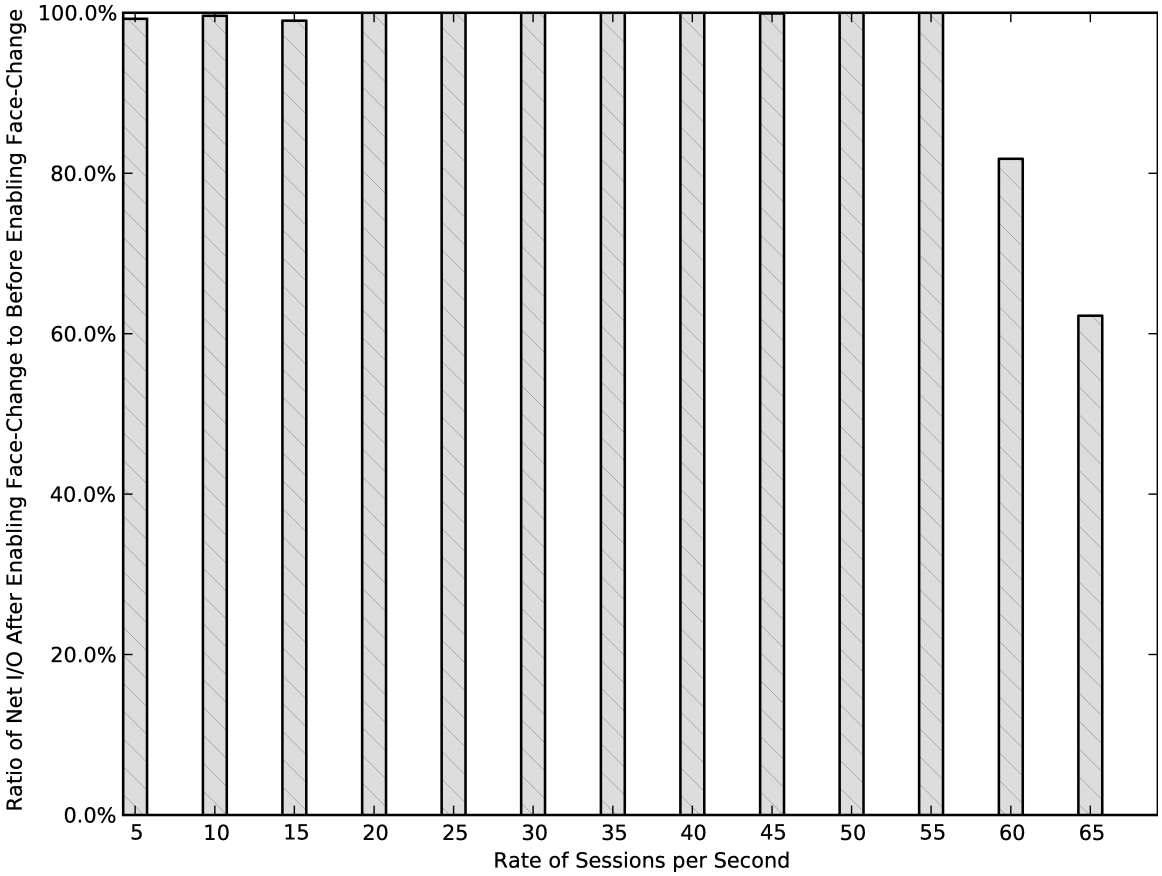

Figure 4.7.: I/O performance results for *Apache* web server

If the rate is below the threshold rate, the application's I/O throughput should be unaffected by FACE-CHANGE. If the rate is far over the threshold rate, FACE-CHANGE may require a more powerful CPU to handle any traffic peaks in the network without slow-down.

4.5 Summary

We make a key observation that the kernel code required by applications of different types varies tremendously. Thus, generating a single system-wide minimized kernel will enlarge the attack surface for all applications involved. We develop FACE-CHANGE to facilitate dynamic kernel view switching among individual applications executed in a VM. FACE-CHANGE transparently presents a customized kernel view to each application to

confine its reachability of kernel code and switch this view upon context switches. In the event that a process breaks its kernel view boundary, FACE-CHANGE is able to recover the missing kernel code and backtrack this anomaly via analysis of the execution history. Our evaluation demonstrates the drastic difference in the size of kernel views of multiple applications, the effectiveness of FACE-CHANGE in revealing the attack patterns of both user and kernel attacks, and the potential of enabling FACE-CHANGE for production VMs.

5 FUTURE WORK

In this chapter we discuss the current limitations of our hypervisor-based active protection security framework and propose the future work we want to explore respectively for each component.

5.1 PROCESS-IMPLANTING

5.1.1 Stealthiness of Implanted Process

Although PROCESS-IMPLANTING tries to maximize the stealthiness of the implanted process, some behaviors (especially for these behaviors that do not exist in the *victim process*) of the *implanted process* may still leave some footprints, e.g., interruption of the *victim process*, anomalous system call invoked from the *implanted process*, inter-process communication, etc. Malicious adversary may leverage such evidences to infer the existence of the *implanted process*. The potential enhancement of stealthiness is to implant a process without relying on any *victim process*. We will explore in this direction further in our future work.

5.2 DRIP

5.2.1 Coverage of Test Suites

A test suite can only ensure the correctness of the tested behaviors within a specific application and the environment in which it executes. Correspondingly, by using a test suite, we only guarantee to preserve the driver functionalities that are covered by this test suite. This may not cover all benign functionality within a driver, or it may require adding new test cases in order to preserve behaviors not originally covered by a test suite. For practical

deployment, we recommend adjusting test suites and generating new purified drivers based on different deployments of application set. This can also be treated as the specialization of a driver. If we do not need some of the redundant features, we can use DRIP to minimize the functionalities of the driver.

5.2.2 False Positives of Removed Kernel API Invocations

We remove function invocations that are not necessary to our test suite, but it does not mean all these removed invocations are useless. For example, we have found some memory deallocation invocations have been removed. This may not impact the execution of the test suite over a short period, but it may cause memory leaks and affect the performance of the system in the long term. We can add some test cases to prevent these invocations from being removed. Another simple solution is to add these well-known functions with specific functionalities into a white list. We can simply skip them when profiling the driver.

5.2.3 Self-contained Malicious Code

Some malicious code can jeopardize the kernel without invoking any kernel APIs. For example, some kernel rootkits can directly modify the kernel memory to achieve their malicious effects. They can evade DRIP's purification as we monitor at the granularity of API invocations. But the functionalities of such self-contained malicious code are limited and it is hard for them to adapt to new kernel versions. In our future work, we will enhance DRIP to monitor at finer-grained memory operations in the profiling phase to address this problem.

5.3 FACE-CHANGE

5.3.1 Crafted Attacks within Minimized Kernel Views

FACE-CHANGE aims to minimize the kernel attack surface for each specific application. If a malicious attack breaks the boundary of the kernel view generated in the profiling

phase, we can detect and report the violations. Compared to system-wide minimization techniques, FACE-CHANGE enforces stricter constraints on kernel code visibility. It is still possible, however, that the kernel code used by the malicious attack is within the subset of the application's kernel view. For example, suppose a web server is compromised and a parasite command-and-control (C&C) server is installed. If this C&C server only uses kernel functionalities that are within the kernel view of the host web server, FACE-CHANGE does not need to recover any missing kernel code and it would be impossible for us to detect its existence in this case. This problem may require a deeper understanding and finer-grained profiling of the semantic behaviors of each application. In addition to recording an application's kernel usage in the profiling phase, we also need to profile the application's behavior, especially its interactions with the kernel. Thereby we can classify the malicious behavior in the runtime phase if it violates the application's known behaviors.

5.3.2 DKOM Kernel Rootkit Detection

For DKOM rootkits [53], which only manipulate kernel data, FACE-CHANGE is unable to identify such attacks because we only monitor anomalies in kernel code execution. In order to detect this kind of attack, we could integrate some existing works [54, 55] into FACE-CHANGE to check the kernel's data integrity. We leave this effort as our future work.

6 RELATED WORK

My work are closely related to several research areas listed below. In this chapter, I describe some representative works from each area and compare our techniques with them.

6.1 Virtual Machine Introspection

Virtual machine introspection has been researched extensively to enhance system security. Security tools are moved from a guest VM to a hypervisor or another trusted VM. The semantic gap [4] problem makes such out-of-VM security tools difficult to leverage the services offered by the guest system. Semantic information needs to be re-created at the hypervisor level. Virtual machine introspection approaches typically can be classified into two categories: passive introspection and active introspection.

In the former category, out-of-VM security tools passively monitor the execution of guest VM to detect anomalous execution. Livewire [5] pioneered the virtual machine introspection methodology to detect malware infections by inspecting the internal states of guest VMs. XenAccess [7], VMwatcher [6], VMscope [56], Antfarm [8] Ether [57] are some representative out-of-box efforts to monitor the guest at the hypervisor level.

However, passive introspection approaches only report, but cannot prevent the malicious behavior from happening. In contrast, active introspection approaches intervene malicious attacks when they are detected. IntroVirt [58] leverages VM introspection to execute vulnerability-specific predicates in a VM for intrusion reproduction. Lycosid [9] detects hidden OS processes through cross-view validation. Then it patches the executable code to influence the runtime of specific processes. Manitou [59] compares instruction-page hashes with memory-page hashes at runtime. If there is no matching, it considers that the instruction page has been corrupted and marked it as non-executable. Out-of-VM active monitoring was proposed in Lares [10] and SIM [11]. Hooks are placed inside the

guest operating system to intercept the events invoking the security tool. Lares can place the security tool at another trusted virtual machine and the hooked system events trigger the virtual machine switch. SIM gains the in-context view by creating a separate guest address space that is protected by the hypervisor to gain the native speed.

PROCESS-IMPLANTING pushes the boundary of active introspection one step further. We dynamically implant a general-purpose security process from the hypervisor into the guest VM. The stealthy nature of the implanted process makes it harder to be identified by malicious adversaries. Furthermore, we provide additional protection and coordination from the hypervisor directly to the implanted process to make it exempt from being tampered and detected.

6.2 Kernel Driver Protection

DRIP is closely related to two categories of kernel driver protection techniques: online driver isolation and offline driver testing.

Online Driver Isolation

Nooks [26] involves a shadow driver mechanism to conceal driver failures from applications by monitoring the state of real drivers during normal operation. It inserts itself when failure occurs, thus improving the reliability of the overall system. SafeDrive [60] improves kernel extension reliability by adding type-based checking to driver code and enforcing runtime memory safety. In order to leverage user level programming tools and reduce kernel level faults introduced by drivers, Microdriver [61] partitions an existing driver into a kernel level driver handling performance critical tasks and a user level driver processing low-performance issues. The Nexus [62] operating system moves the device driver to user space and it leverages device-specific reference monitors to validate that all the interactions between drivers and devices conform to safety specifications. To protect untrusted device drivers from compromising a system, SUD [27] leverages recent hardware support to confine operations of devices and allows unmodified Linux device drivers to run

in user processes by emulating a Linux kernel environment in user space. These research efforts are designed to isolate buggy drivers at runtime. Compared with DRIP, they incur additional performance overhead and cannot target intentionally malicious drivers.

Offline Driver Testing

SDV [30] statically checks source code paths of device drivers to make sure they use the Windows API correctly. DDT [31] exercises the closed source device drivers to find bugs by using symbolic execution. These two offline approaches are designed to test buggy drivers thoroughly but not for removing malicious behaviors from the driver. On the other hand, both of them can complement DRIP for improving the coverage of test suites.

6.3 Emulation-Based Analysis

DRIP leverage system emulation technique to perform driver purification. Emulation-based techniques have been widely used in malware profiling and analysis.

K-Tracer [63] dynamically analyzes a rootkit's malicious behavior by using backward slicing and chopping techniques. Panorama [64] leverages the system-wide taint tracking technique to capture the privacy-breaching behavior of malware. HookFinder [65] and HookMap [66] perform dynamic analysis to identify kernel hooks implanted by rootkits. PoKeR [67] profiles a kernel rootkit's behavior by traversing from static objects to locate dynamic objects and performing address-object mapping. Instead of detecting malware, DRIP extends the emulation platform to eliminate malicious behaviors from trojaned kernel drivers.

Virtuoso [68] involves a technique to create introspection-based security tools automatically out of a VM by tracing and combining the execution traces of In-VM programs. RevNIC [69] is a technique that helps automatically reverse engineer the logic of a network device driver and synthesize a new driver with the same functionality for a different platform. Rather than combining traces to re-create a new binary, the goal of DRIP is to

identify malicious logic in existing drivers and perform binary rewriting to eliminate their malicious effects.

6.4 Kernel Minimization

Earlier research on kernel minimization was not specifically security oriented. The primary goal of these works was to shrink the kernel's in-memory size to adapt to the limited hardware resources of embedded systems. Lee et al. [43] used a call graph approach to eliminate redundant code from the Linux kernel. Chanet et al. [45] applied link-time compaction and specialization techniques to reduce the kernel memory. He et al. [44] reduced the memory footprint by keeping infrequently executed code on disk and loading it on demand. Recent research has focused on minimizing the OS kernel to reduce the attack surface exposed to applications. Kurmus et al. [42] proposed a kernel reduction approach that automatically generates kernel build configurations based on profiling results of expected workloads.

Compared to existing kernel minimization works, FACE-CHANGE customizes the kernel code visibility for each individual application, thus minimizing the kernel attack surface at finer time-granularity. In addition, our system is more flexible and can adapt to changes in the execution environment and support new applications without rebooting the system.

6.5 Sandboxing

Sandboxing is a general security mechanism that provides a secure execution environment for running untrusted code.

One category of sandboxing works is to constrain the untrusted code's capabilities via predefined security policies. Janus [70] is a filtering approach to perform system call interposition based on the predefined policy. Ostia [71] proposed a delegating architecture to virtualize the system call interface and provides a user level sandbox to control the access of resources. Capsicum [72] extends the Unix API to allow an application to perform self-compartmentalization, i.e., confining itself in a sandbox that only allows essential ca-

pabilities. Seccomp [73] is a sandboxing mechanism implemented in the Linux kernel to constrain the system call interface of process. If the process attempts to issue the system call that is not allowed, it will be terminated by the kernel. SELinux [74] is a security module in the Linux kernel that enforces mandatory access-control policies on applications. Similar to SELinux, AppArmor [75] restricts the capabilities of a program through binding a security profile. TxBox [76] is based on system transactions to speculatively execute an untrusted application and recover from harmful effects. Process Firewalls [77] is a kernel-base protection mechanism to avoid resource access attacks through examining the internal state of a process and enforcing invariants on each system call.

Another category of sandboxing approaches is to enforce access control through re-compilation, binary rewriting, and instrumentation: PittSFIeld [78] extends software fault isolation [79] (SFI) to x86. It checks unsafe memory writes and constrains jump targets to aligned addresses. XFI [80] leverages control flow integrity toprevent circumvention and support fine-grained memory access control. Vx32 [81] is a sandbox that confines the system calls and data accesses of guest plugins without kernel modification. NaCl [82] leverages SFI to provide a constrained execution environment for the native binary code of browser-based application. TRuE [83] replaces the standard loader with a security-hardened loader and leverages SFI to run untrusted code. Program shepherding [84] enforces security policies by monitoring control flow transfers during the execution of a program.

In the virtualization/emulation environment, a full system is considered to be confined in a sandbox and the protection is provided at hypervisor level: Secvisor [85] ensures that only approved code can be executed in kernel mode to protect the kernel against code injection attacks. NICKLE [12] enforces that only authorized kernel code can be fetched for execution in kernel space. To guarantee the integrity of kernel hooks, HookSafe [13] relocates hooks to a page-aligned memory space and regulates accesses to them via page-level protection. HUKO [29] is a hypervisor-based approach to enforce mandatory access control policies on untrusted kernel extensions. Gateway [28] isolates kernel drivers in a different address space from the base kernel and monitors their kernel API invocations.

FACE-CHANGE can also be considered as a sandboxing approach. The difference from these previous works is that we sandbox each individual application by constraining its visibility of kernel code. We also enforce our approach at the hypervisor level to be transparent to the guest system.

7 CONCLUSION

Virtualization technology has been widely adopted in the security area to protect guest VMs from being compromised by cyber-attacks. Existing hypervisor-based security approaches, which are traditionally used for intrusion detection, malware analysis, and integrity check, conduct only passive monitoring on the guest systems, but missing the capabilities of performing active protection on the guest VM, such as patching the vulnerabilities, eliminating the malicious logic, and shrinking the kernel attack surface, etc.

My research aims to expand the reach of the hypervisor to actively protect the guest VM. In this dissertation, we present a hypervisor-based security framework that consists of three key components, PROCESS-IMPLANTING, DRIP, and FACE-CHANGE, to provide active protection at the level of user processes, kernel drivers, and OS kernel respectively, within the guest VM.

PROCESS-IMPLANTING is an active virtual machine introspection technique, which enables implantation of a security process directly from the hypervisor into the guest VM. The dynamic nature of such *implanted process* make it harder to be pinpointed by malicious adversaries.

DRIP targets trojaned kernel drivers, i.e., malicious logic is embedded alongside the benign code. We purify the trojaned drivers by systematically eliminating the malicious functionality and preserving the benign logic at binary level.

FACE-CHANGE shrinks the kernel attack surface for applications running within the guest VM. Compared to existing system-wide kernel minimization techniques, we dynamically switch multiple kernel views, each customized for a different application, at runtime to achieve kernel minimalism at finer time-granularity.

From our evaluation results on both security and performance, we demonstrate that PROCESS-IMPLANTING, DRIP, and FACE-CHANGE can effectively provide active protection for guest VMs with minimum negative impact on the guest system execution. Fur-

thermore, the reasonable performance overhead makes it realistic to be deployed in the real-world cloud infrastructures.

REFERENCES

REFERENCES

[1] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process Implanting: A New Active Introspection Framework for Virtualization. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, 2011.

[2] Zhongshu Gu, William N. Sumner, Zhui Deng, Xiangyu Zhang, and Dongyan Xu. DRIP: A Framework for Purifying Trojaned Kernel Drivers. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2013.

[3] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[4] Peter M. Chen and Brian D. Noble. When Virtual Is Better Than Real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001.

[5] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[6] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[7] Bryan D. Payne, Martim D. P. de A. Carbone, and Wenke Lee. Secure and Flexible Monitoring of Virtual Machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference*, 2007.

[8] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[9] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification using Lycosid. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008.

[10] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[11] Monirul I Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[12] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[13] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[14] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, 2007.

[15] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel® Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 2006.

[16] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[17] Leendert van Doorn. Hardware Virtualization Trends. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, 2006.

[18] Sony, Rootkits and Digital Rights Management Gone Too Far. http://blogs.technet.com/b/markrussinovich/archive/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far.aspx.

[19] Legit bootkits. http://www.securelist.com/en/analysis/204792203/Legit_bootkits.

[20] Infecting loadable kernel modules. http://www.phrack.org/issues.html?issue=61&id=10#article.

[21] Infecting loadable kernel modules kernel versions 2.6.x/3.0.x. http://www.phrack.org/issues.html?issue=68&id=11#article.

[22] Module injection in 2.6 kernel. http://www.linuxforum.net/forum/showflat.php?Cat=&Board=security&Number=536404&page=0&view=collapsed&sb=5&o=31&fpart.

[23] Driverless Kernel Mode Rootkit. http://www.rohitab.com/discuss/topic/28440-driverless-kernel-mode-rootkit.

[24] Kernel Driver Backdooring. http://securitylabs.websense.com/content/Blogs/2730.aspx.

[25] The ERESI Reverse Engineering Software Interface. http://www.eresi-project.org.

[26] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[27] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, 2010.

[28] Abhinav Srivastava and Jonathon Giffin. Efficient Monitoring of Untrusted Kernel-Mode Execution. In *Proceedings of the 18th Annual Network and Distributed Systems Security Symposium*, 2011.

[29] Xi Xiong, Donghai Tian, and Peng Liu. Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[30] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough Static Analysis of Device Drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006.

[31] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing Closed-Source Binary Device Drivers with DDT. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, 2010.

[32] Linux Test Project. http://ltp.sourceforge.net.

[33] skipfish web application security scanner. https://code.google.com/p/skipfish.

[34] curl-loader. http://curl-loader.sourceforge.net.

[35] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[36] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. SymDrive: Testing Drivers without Devices. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[37] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[38] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[39] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu. Probing the Guts of Kprobes. In *Proceedings of the 2006 Linux Symposium*, 2006.

[40] Linux kernel module that allows you to set events on pressed keys. http://en.chuso.net/linux-kernel-module-that-allows-you-to-set-events-on-pressed-keys.html.

[41] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. BISTRO: Binary Component Extraction and Embedding for Software Security Applications. In *Proceedings of the 2013 European Symposium on Research in Computer Security*. 2013.

[42] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.

[43] Chi-tai Lee, Jim-min Lin, Zeng-wei Hong, and Wei-tsong Lee. An Application-Oriented Linux Kernel Customization for Embedded Systems. *Journal of Information Science and Engineering*, 1995.

[44] Haifeng He, Saumya Debray, and Gregory Andrews. The Revenge of the Overlay: Automatic Compaction of OS Kernel Code via On-Demand Code Loading. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, 2007.

[45] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. System-wide Compaction and Specialization of the Linux Kernel. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[46] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[47] RUBiS. http://rubis.ow2.org/.

[48] injectso: Modifying and spying on running processes under Linux. http://www.blackhat.com/presentations/bh-europe-01/shaun-clowes/bh-europe-01-clowes.ppt.

[49] Cymothoa — Stealth backdooring tool. http://cymothoa.sourceforge.net/.

[50] Single Process Parasite. http://www.phrack.org/issues.html?issue=68&id=9#article.

[51] Injected Evil(executable files infection). http://z0mbie.host.sk/infelf.html.

[52] kbeast-v1. http://core.ipsecs.com/rootkit/kernel-rootkit/kbeast-v1/.

[53] Jamie Butler. DKOM (Direct Kernel Object Manipulation). http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf.

[54] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*, 2010.

[55] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring. In *Proceedings of the International Conference on Availability, Reliability and Security*, 2009.

[56] Xuxian Jiang and Xinyuan Wang. "Out-of-the-box" Monitoring of VM-based High-Interaction Honeypots. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, 2007.

[57] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[58] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[59] Lionel Litty and David Lie. Manitou: A Layer-Below Approach to Fighting Malware. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[60] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[61] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The Design and Implementation of Microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[62] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device Driver Safety Through a Reference Validation Mechanism. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[63] Andrea Lanzi, Monirul I. Sharif, and Wenke Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, 2009.

[64] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[65] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, 2008.

[66] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering Persistent Kernel Rootkits Through Systematic Hook Discovery. In *Proceedings of the Recent Advances in Intrusion Detection*, 2008.

[67] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Multi-Aspect Profiling of Kernel Rootkit Behavior. In *Proceedings of the 4th ACM European Conference on Computer Systems*, 2009.

[68] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[69] Vitaly Chipounov and George Candea. Reverse Engineering of Binary Device Drivers with RevNIC. In *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[70] Ian Goldberg, David Wagner, Randi Thomas, and Eric A Brewer. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In *Proceedings of the 6th USENIX Security Symposium*, 1996.

[71] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[72] Robert N.M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical Capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[73] Seccomp and sandboxing. http://lwn.net/Articles/332974/.

[74] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[75] Mick Bauer. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux Journal*, 2006.

[76] Suman Jana, Donald E. Porter, and Vitaly Shmatikov. TxBox: Building Secure, Efficient Sandboxes with System Transactions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.

[77] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. Process Firewalls: Protecting Processes During Resource Access. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[78] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[79] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993.

[80] Úlfar. Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation*, 2006.

[81] Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of the 2008 USENIX Conference on USENIX Annual Technical Conference*, 2008.

[82] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009.

[83] Mathias Payer, Tobias Hartmann, and Thomas R. Gross. Safe Loading — A Foundation for Secure Execution of Untrusted Programs. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[84] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[85] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

VITA

VITA

Zhongshu Gu earned his BS degree in Computer Science from Fudan University in 2007, and PhD degree in Computer Science from Purdue University in 2015. His research interests focus on the enhancement of system security and dependable computing for heterogeneous computing environments. His research background spans the areas of system security, operating systems, virtualization technology, program analysis, memory forensics, and malware analysis. In the fall of 2015, he joined IBM Thomas J. Watson Research Center as a Research Staff Member.