ABSTRACT

SRINIVASAN, DEEPA. Elevating Virtual Machine Introspection for Fine-grained Process Monitoring: Techniques and Applications. (Under the direction of Xuxian Jiang.)

Recent rapid malware growth has exposed the limitations of traditional in-host malware-defense systems and motivated the development of secure virtualization-based solutions. By running vulnerable systems as virtual machines (VMs) and moving security software from inside VMs to the outside, the out-of-VM solutions securely isolate the anti-malware software from the vulnerable system. However, the external placement of the anti-malware tool introduces a number of limitations, including the well-known semantic gap problem.

In this dissertation, we study the limitations in prior out-of-VM approaches and develop the *process out-grafting framework* in order to effectively address them. First, we address isolation and compatibility challenges in out-of-VM approaches for fine-grained process execution monitoring by developing two key techniques. The first key technique, *on-demand grafting*, relocates a suspect process from inside a VM to run side-by-side with the out-of-VM security tool. This effectively removes the semantic gap and supports existing user-mode monitoring tools without any modification. The second key technique, *mode-sensitive split execution*, forwards system calls back to the VM and enables continued execution of the out-grafted process without weakening the isolation of the monitoring tool. Our experiments with a prototype show that we can effectively use *process out-grafting* to natively support a number of existing tools without any modification. The evaluation results, including measurement with benchmark programs, show the effectiveness and practicality of our approach.

Next, based on the fine-grained monitoring capability, we apply and extend process out-grafting to enable semantically-rich out-of-VM policy enforcement. Specifically, we demonstrate out-of-VM system call policy enforcement, which effectively restricts the behavior of an out-grafted process. Further, in order to facilitate the secure observation of a process that violates system policy, we develop the VMsnare component of our framework. In VMsnare, we have designed and developed our next two key techniques, *attack preservation* and *live analysis*. With these two techniques, we effectively extract live malware processes from a production environment into a honeypot for flexible and extensible analysis. Our experiments with a prototype implementation demonstrate the effectiveness and practicality of our approach.

Finally, in our framework, we facilitate the time-traveling forensic analysis of intrusions and derive valuable insight into attackers' techniques and motivation. Towards this, we have designed and developed the Timescope component of our framework, which leverages insights from previous VM-level deterministic record and replay systems and enables multi-faceted and extensible forensic analysis. We have further extended Timescope and developed a number

of honeypot-specific forensic analysis modules. By repeatedly traveling back in time, multiple phases of analysis can be performed, either in parallel or sequentially.

Elevating Virtual Machine Introspection for Fine-grained Process Monitoring: Techniques and Applications

by
Deepa Srinivasan

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2013

APPROVED BY:

Gregory Byrd	Vincent Freeh
Steven Hunter	Peng Ning

Xuxian Jiang
Chair of Advisory Committee

DEDICATION

To my son, my source of inspiration.
To my husband, my pillar of strength.
To my parents, who gave me wings to soar.

BIOGRAPHY

Deepa Srinivasan is originally from Madras, India. She received her Bachelor of Science in Mathematics from the University of Madras, India in 1997 and her Master of Science in Computer Science from the Oregon Health and Science University, Oregon in 2003. She has worked for over twelve years in industry, particularly at IBM Corp., in the Systems and Technology group. At IBM, she has worked on several key projects including server consolidation, firmware development, systems management, and system virtualization. In addition, she has collaborated with IBM Research on the Trusted Virtual Datacenter project, which was designed to satisfy enterprise security goals and provide trust guarantees for datacenter environments based on virtualization. While working at IBM, she started pursuing the Doctor of Philosophy degree in Computer Science at North Carolina State University. In 2010, she became a full-time student and worked under the guidance of Prof. Xuxian Jiang. Her primary research interests are in the areas of operating systems, virtualization, and systems security. She will graduate with a Ph.D. degree in Computer Science from the North Carolina State University in May 2013. She recently joined Microsoft Corp. as a Senior Software Development Engineer in the Windows Hyper-V group and will continue in this role after graduation.

ACKNOWLEDGEMENTS

My pursuit and completion of the Ph.D. degree has been made possible only due to the guidance, support, and help that I have received from a number of people. I would like to take this opportunity to thank each one of them. First, I would like to thank my advisor Prof. Xuxian Jiang for his guidance and support throughout my graduate research. Prof. Jiang's standard for excellence, strong work ethic, and drive for results have improved me significantly as a researcher and as a professional. I am grateful to Prof. Jiang for allowing me the freedom to pursue research in an area that I am passionate about and for patiently accommodating my personal constraints.

I would like to thank Prof. Peng Ning, Prof. Vincent Freeh, Prof. Gregory Byrd, and Dr. Steven Hunter for their time and efforts serving on my advisory committee. Particularly, I would like to thank Prof. Peng Ning and Prof. Vincent Freeh for their encouragement and advice towards improving my research skills. I would like to thank Prof. Gregory Byrd whose teaching has shaped my learning in the area of computer architecture. I would like to thank Dr. Steven Hunter, IBM Fellow, who has been an invaluable mentor and role model to me, from the early stages of my degree. I would like to thank Prof. William Enck who graciously served as an examiner during my final defense presentation.

I would like to thank Prof. Dongyan Xu from Purdue University, with whom I was fortunate to collaborate on multiple publications. I would like to thank my research group colleagues, Zhi Wang (now at Florida State University), Michael Grace, Yajin Zhou, Chiachih Wu, Minh Tran, Tyler Bletsch, and Jinku Li, for many insightful discussions. It was an invigorating experience to collaborate with them on several different research projects, during my time at the lab. I would like to thank Dr. Reiner Sailer and his team at IBM T.J. Watson Research Center, collaborating with whom, I developed my early research interest in the areas of virtualization and security. I would like to thank my former managers, Pat Genovese and Elizabeth Ball, for their support towards my pursuit of external education, while working full-time at IBM.

I would like to thank the Director of Graduate Programs Prof. Douglas Reeves for his strong guidance during various stages of my Ph.D. degree. I would like to thank the department administration for their support during my time as a graduate research assistant. Particularly, I would like to thank Mr. Ron Hartis who was invaluable in his help for my international conference travel. I would like to thank Ms. Margery Page, Ms. Carol Allen, and Ms. Susan Peaslee for their always-prompt help with my requests. I would like to thank my friends in Raleigh, Radhika, Mutuk, Vaidehi, Vijay, Sajiree, and Siddarth, for their support especially during the final year of my degree. I would also like to thank my management at Microsoft for their encouragement during my final defense exam.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **vii**

LIST OF FIGURES . . . . . **viii**

Chapter 1 Introduction **1**
1.1 Problem Overview 1
1.2 Dissertation Statement and Contributions 2
1.3 Dissertation Organization 4

Chapter 2 Efficient Fine-Grained Monitoring with Process Out-Grafting . . . **5**
2.1 Background 5
2.2 Process Out-Grafting Design 7
2.2.1 Goals and Assumptions 7
2.2.2 On-demand Grafting 9
2.2.3 Mode-sensitive Split Execution 12
2.3 Implementation 16
2.3.1 On-demand Grafting 16
2.3.2 Mode-sensitive Split Execution 17
2.3.3 Process Restoration 18
2.4 Evaluation 19
2.4.1 Security Analysis 19
2.4.2 Case Studies 20
2.4.3 Performance 23
2.5 Discussion 24
2.6 Related Work 26
2.7 Summary 27

Chapter 3 Semantically-Rich Out-of-VM Policy Enforcement **28**
3.1 Background 28
3.2 Design and Implementation 29
3.3 Evaluation 31
3.3.1 Security Analysis 31
3.3.2 Effectiveness 31
3.4 Discussion 34
3.5 Summary 34

Chapter 4 Trapping Intrusions from Production Environments with VMsnare **35**
4.1 Background 35
4.2 Design 37
4.2.1 Goals and Assumptions 37
4.2.2 Attack Preservation 38
4.2.3 Live Analysis 42
4.3 Implementation 45

4.3.1 Attack Preservation . . . 46
4.3.2 Live Analysis . . . 46
4.4 Evaluation . . . 49
4.4.1 Security Analysis . . . 49
4.4.2 Effectiveness . . . 51
4.4.3 Performance . . . 53
4.5 Discussion . . . 54
4.6 Related Work . . . 55
4.7 Summary . . . 56

Chapter 5 Time-traveling Forensic Analysis with Timescope . . . 58
5.1 Background . . . 58
5.2 Design . . . 60
5.2.1 Analysis Modules . . . 62
5.3 Implementation . . . 63
5.3.1 QEMU Record & Replay . . . 64
5.3.2 Analysis Modules . . . 65
5.4 Evaluation . . . 67
5.4.1 R&R Accuracy . . . 67
5.4.2 Time-traveling Analysis . . . 68
5.4.3 Performance . . . 71
5.5 Discussion . . . 72
5.6 Related Work . . . 73
5.7 Summary . . . 74

Chapter 6 Conclusion and Future Work . . . 75

REFERENCES . . . 77

LIST OF TABLES

Table 2.1 Software packages used in our evaluation of process out-grafting 23

Table 4.1 Software packages used in our evaluation of VMsnare 53

Table 5.1 Performance overhead in a Timescope VM record session 72

LIST OF FIGURES

Figure 1.1 Key contributions of the dissertation 3

Figure 2.1 An overview of process out-grafting 7
Figure 2.2 Out-grafted process memory mapping in production and security VMs . . 11
Figure 2.3 Interplay between the production VM stub and the security VM helper LKM 14
Figure 2.4 Production VM slowdown with a contending process out-grafted 24

Figure 3.1 Out-of-VM system call policy enforcement 30
Figure 3.2 Semantically-rich information available for in-VM system call policy enforcement 32
Figure 3.3 Semantically-rich information available for out-of-VM system call policy enforcement 33

Figure 4.1 An overview of VMsnare 36
Figure 4.2 In VMsnare, hypervisor re-maps host physical page frame of guest physical address in production VM to host virtual address of honeypot emulator 40
Figure 4.3 Incoming attack traffic redirection in VMsnare 42
Figure 4.4 Outgoing attack traffic redirection in VMsnare 43
Figure 4.5 VMsnare transfers an attacking process from the production VM to the honeypot VM 50
Figure 4.6 Production VM slowdown after VMsnare transfers an attack to the honeypot VM 54

Figure 5.1 Timescope enables time-traveling forensic analysis of honeypots 61
Figure 5.2 The contamination graph of Slapper worm reconstructed from a Timescope-based replay session 69
Figure 5.3 Timescope-based multi-phase time-traveling forensic analysis of Slapper infection. The replay sessions are run only for the time window indicated by the solid regions in the execution timeline. Results obtained during a replay session are indicated by asterisks. 70
Figure 5.4 SucKIT rootkit analysis using Timescope 71

Chapter 1

Introduction

1.1 Problem Overview

Computer malware (e.g., viruses and trojans), in its seemingly infinite evolution of functionality and forms, is one of the largest threat that end users and enterprises are combating daily, despite the widespread availability of anti-malware software and tools. A recent report from McAfee [7] shows a rapidly exploding malware growth with new record numbers "accomplished" in the previous year. Specifically, as highlighted in the report, "McAfee Labs identified more than 20 million new pieces of malware in 2010," which translates into nearly 55,000 new malware samples discovered every day! Moreover, "of the almost 55 million pieces of malware McAfee Labs has identified and protected against, 36 percent of it was written in 2010!" This alarming trend reveals the disturbing fact that existing malware defenses fail to effectively contain the threat and keep up with the malware growth.

Specifically, when we examine traditional anti-malware tools, we observe that they are typically deployed within vulnerable systems and could be the first targets once malware infect a computer system. In other words, though these traditional *in-host* tools are valuable in monitoring system behavior or detecting malicious activities (with their native access inside the systems), they are fundamentally limited in their isolation capability to prevent themselves from being infected in the first place. To address that, researchers have proposed leveraging the advances in system virtualization and then moving anti-malware tools from inside the systems to outside.

Surveying research in the area of system virtualization over the last 15 years, we observe that it was developed on commodity systems primarily to address factors such as operating system scalability limitations [27], low server utilization, and workload consolidation [12]. System virtualization, which was introduced in the 1960s on mainframe platforms [45, 72], has been proven effective on commodity hardware and has become mainstream with the availability of

several commercial and open-source hypervisors [5, 8, 18, 23]. Further, researchers saw abundant opportunity to leverage the additional layer of indirection, namely the hypervisor, for a variety of other purposes.

Particularly, virtualization has been extensively leveraged to improve many facets of system security. Virtual machines (VMs), running as self-contained environments, offer an elegant way to provide encapsulation and isolation for various security purposes. For example, by moving workloads into VMs, virtualization has been used to protect the running guest [40, 70, 75]. Further, virtualization has been used to improve effectiveness of honeypots [36, 88, 89] and as ideal environments for malware analysis [24, 34].

To address the isolation limitations of traditional anti-malware tools, *virtual machine (VM) introspection* or *out-of-VM* approaches [21, 42, 50, 51, 67, 68, 85] have been proposed which change the malware defense landscape by leveraging virtualization to run vulnerable systems as VMs. By enlisting help from the underlying hypervisor, out-of-VM approaches can effectively overcome the isolation challenge that encumbers traditional in-host approaches. However, by separating anti-malware software from the untrusted systems, they also encounter several limitations. For example, out-of-VM approaches suffer from the well-known semantic gap and are unable to provide efficient fine-grained monitoring capabilities suitable for VMs running production workloads. We argue that these limitations are partly due to the fundamental reliance on external low-level VM states and events in existing out-of-VM approaches.

1.2 Dissertation Statement and Contributions

Departing from prior approaches, in this dissertation, our central thesis is that the effectiveness of an out-of-VM system will be significantly improved by leveraging existing operating system abstractions. Specifically, by leveraging a core abstraction provided by the operating system, namely the *process*, we address several key limitations in existing out-of-VM approaches. Towards this, we present the design, implementation, and evaluation of the integrated *process out-grafting framework* which elevates virtual machine introspection to enable fine-grained process monitoring. Further, we extend the framework with applications in out-of-VM policy enforcement, live intrusion analysis, and time-traveling forensic analysis. Below, we highlight the main contributions of this dissertation, as illustrated in Figure 1.1.

- **Semantic gap in out-of-VM monitoring (Chapter 2)** We have designed and implemented our first two key techniques to enable process out-grafting: *on-demand grafting* and *mode-sensitive split execution*. Our experiments with a prototype [84] demonstrate efficient and effective out-of-VM monitoring while *re-using* existing process monitoring tools such as *strace* (system call monitoring), *ltrace* (library call monitoring) and even *gdb*

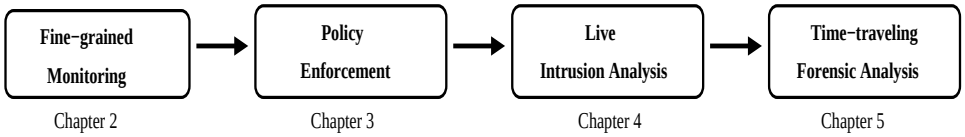

Figure 1.1: Key contributions of the dissertation

(for instruction-level monitoring). Thus, we have demonstrated the practicality and usefulness of process out-grafting in addressing the semantic gap challenge while efficiently monitoring even production VMs.

- **Semantically-rich out-of-VM policy enforcement (Chapter 3)** Next, we have applied and extended process out-grafting to enable semantically-rich out-of-VM policy enforcement, towards intrusion prevention. Specifically, we have demonstrated out-of-VM enforcement of system call policies, which are used effectively to confine untrusted processes or to enforce expected behavior of well-known applications.

- **Trapping intrusions from production environments (Chapter 4)** When a malware infection is detected in a production environment, typically, the response is to terminate the infection or malicious process. However, analysis of the malicious process can provide significant insight into its behavior in a real production environment. In order to facilitate "live analysis" of infections that have targeted production environments, we have developed the VMsnare component of the process out-grafting framework. We have designed and implemented the two key techniques, *attack preservation* and *live analysis*, to transfer live malware infections from production environments into a highly-instrumented honeypot for flexible and extensible analysis. Our experiments with a prototype implementation demonstrate the practicality and effectiveness of VMsnare.

- **Time-traveling forensic analysis (Chapter 5)** After an intrusion is detected and captured, forensic analysis provides valuable insight into attackers' techniques and motivation. We have designed and implemented the Timescope component of our framework, that leverages previous insights from VM-level deterministic record and replay systems to enable multi-faceted and extensible forensic analysis. We have released Timescope [83] as open-source to engage the security community and benefit related research efforts that may require similar features. In addition, we have developed a number of honeypot-specific forensic analysis modules. By repeatedly "traveling back in time", multiple phases of analysis can be performed, either in parallel or sequentially.

1.3 Dissertation Organization

This dissertation is organized into six chapters, including this introductory chapter. In Chapter 2, we present the design, implementation, and evaluation of the first two key techniques in process out-grafting and demonstrate its practicality in address the semantic gap problem. In Chapter 3, we describe applying and extending these techniques to enable out-of-VM policy enforcement. In Chapters 4 and 5, we present the design, implementation, and evaluation of the VMsnare and Timescope components of our framework, respectively. Finally, we conclude this dissertation and describe future directions of research in Chapter 6.

Chapter 2

Efficient Fine-Grained Monitoring with Process Out-Grafting

2.1 Background

Traditional *in-host* tools are valuable in monitoring system behavior or detecting malicious activities. However, they are fundamentally limited in their isolation capability to prevent themselves from being infected in the first place. Prior *out-of-VM* approaches [21, 42, 50, 51, 67, 68, 85], change the malware defense landscape by leveraging recent advances of virtualization to run vulnerable systems as VMs and then moving anti-malware tools from inside the systems to outside. By enlisting help from the underlying virtualization layer (or hypervisor), out-of-VM approaches can effectively overcome the isolation challenge that encumbers traditional in-host approaches. However, by separating anti-malware software from the untrusted systems, they also naturally encounter the well-known semantic gap challenge: these anti-malware software or tools – as out-of-VM entities – need to monitor or semantically infer various in-VM activities.

In the past several years, researchers have actively examined this semantic-gap challenge and implemented a number of introspection-based systems [35, 51] to mitigate it. For example, VMwatcher [51] proposes a guest view casting technique to apply the knowledge of inner guest OS kernel, especially the semantic definition of key kernel data structure and functionality, to bridge the semantic gap. Virtuoso [35] aims to automate the process of extracting introspection-relevant OS kernel information (by monitoring the execution of an in-guest helper program) for the construction of introspection-aware security tools. Though these systems make steady progresses in bridging the semantic gap, the gap still inevitably leads to a compatibility problem. In particular, none of these introspection-based systems is compatible with existing anti-malware software (including various system/process monitoring tools) that were designed to run within a host. Due to the lack of compatibility, significant effort and advanced mechanisms [35, 51]

are still needed to re-engineer these tools and adapt them for different guest OSs. This is especially concerning when fine-grained monitoring of individual processes requires intercepting a wide variety of events (e.g. user-library function calls or system calls) and interpreting them meaningfully (e.g. to determine their arguments). Worse, these introspection-based solutions are sensitive to guest OS versions or variants and to some extent fragile to any change or patch to the guest OS. As a result, this compatibility problem severely limits the effectiveness and the adoption of these out-of-VM approaches.

In this chapter, we introduce *process out-grafting*, an architectural approach that addresses both isolation and compatibility challenges for out-of-VM, fine-grained user-mode process execution monitoring. Similar to prior out-of-VM approaches, out-grafting still confines vulnerable systems as VMs and deploys security tools outside the VMs. However, instead of analyzing the entire VM on all running processes, in out-grafting, we leverage a core OS abstraction and focus on each individual process for fine-grained execution monitoring. More importantly, our approach is designed to naturally support existing user-mode process monitoring tools (e.g., *strace*, *ltrace*, and *gdb*) outside of monitored VMs on an internal suspect process, without the need of modifying these tools or making them introspection-aware (as required in prior out-of-VM approaches). For simplicity, we use the terms "production VM" and "security VM" respectively to represent the vulnerable VM that contains a suspect process and the analysis VM that hosts the security tool to monitor the suspect process.

To enable process out-grafting, we have developed two key techniques. Specifically, the first technique, *on-demand grafting*, relocates the suspect process on demand from the production VM to security VM (that contains the process monitoring tool as well as its supporting environment). By doing so, grafting effectively brings the suspect process to the monitor for fine-grained monitoring, which leads to at least two important benefits: (1) By co-locating the suspect process to run side-by-side with our monitor, the semantic gap caused by the VM isolation is effectively removed. In fact, from the monitor's perspective, it runs together with the suspect process inside the same system and based on its design can naturally monitor the suspect process without any modification. (2) In addition, the monitor can directly intercept or analyze the process execution even at the granularity of user-level function calls, without requiring hypervisor intervention, which has significant performance gains from existing introspection-based approaches.

To still effectively confine the (relocated) suspect process, our second technique enforces *mode-sensitive split execution* of the process. Specifically, only the user-mode instructions of the suspect process, which is our main focus for fine-grained monitoring, will be allowed to execute in the security VM; all kernel-mode execution that requires the use of OS kernel system services is forwarded back to the production VM. By doing so, we can not only maintain a smooth continued execution of the suspect process after relocation, but ensure its isolation

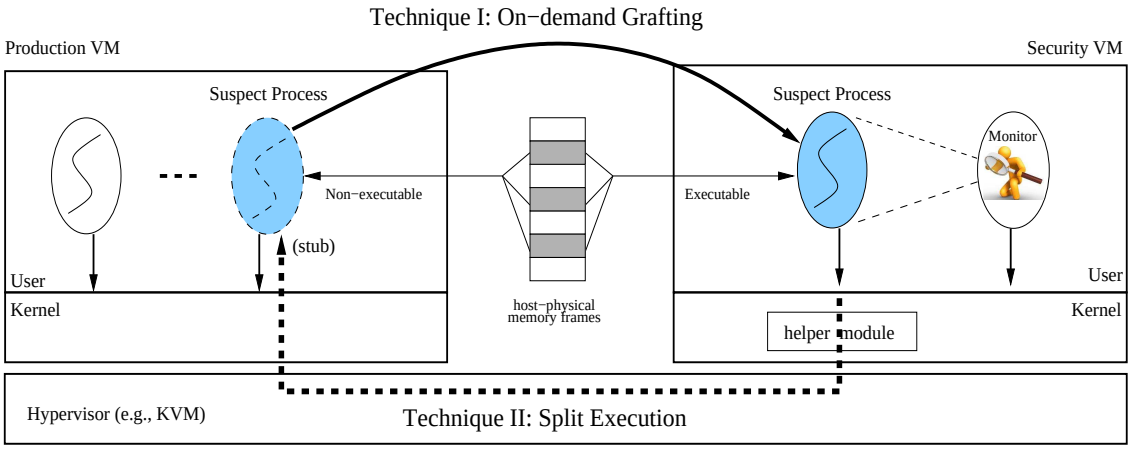

Figure 2.1: An overview of process out-grafting

from our monitoring tools. Particularly, from the suspect process' perspective, it is still logically running inside the production VM. In the meantime, as the suspect process physically runs inside the security VM, the monitoring overhead will not be inflicted to the production VM, thus effectively localizing monitoring impact within the security VM.

We have implemented a proof-of-concept prototype on KVM/ Linux (version kvm-2.6.36.1) and tested it to out-graft various processes from different VMs running either Fedora 10 or Ubuntu 9.04. We have evaluated it with a number of different scenarios, including the use of traditional process monitoring tools, i.e., *strace*/*ltrace*/*gdb*, to monitor an out-grafted process from another VM. Note that these fine-grained process monitoring tools cannot be natively supported if the semantic gap is not effectively removed. Moreover, we also show that advanced (hardware-assisted) monitoring tools [59] can be deployed in the security VM to monitor a process in the production VM, while they may be inconvenient or even impossible to run inside the production VM. The performance evaluation with a number of standard benchmark programs shows that our prototype implementation incurs a small performance overhead and the monitoring overhead is largely confined within the security VM, not the production VM.

2.2 Process Out-Grafting Design

2.2.1 Goals and Assumptions

Process out-grafting is a virtualization-based approach that advances current out-of-VM approaches for fine-grained process-level execution monitoring. It is introduced to effectively support existing user-mode process-level monitoring tools while removing the inherent semantic gap in out-of-VM approaches. To achieve that, we have three main design goals.

- *Isolation* Process out-grafting should strictly isolate process monitoring tools from the untrusted process. In other words, the untrusted process will be architecturally confined without unnecessarily exposing monitoring tools. This essentially achieves the same isolation guarantee as existing out-of-VM approaches.

- *Compatibility* Our solution should naturally support existing fine-grained process monitoring tools (e.g., *strace* /*ltrace*/*gdb*) without modification. Accordingly, all required semantic information by these tools need to be made available in the security VM.

- *Efficiency* Process out-grafting needs to efficiently support existing process monitoring tools without much additional performance overhead caused by isolation. Due to its process-level granularity, we also need to localize the monitoring overhead to the monitored process, without unnecessarily impacting the production VM as a whole.

In this work, we assume no trust from the suspect process being monitored. An attacker may also introduce malicious software (either user-mode or kernel-mode) to compromise the production VM. However, we assume the presence of a trusted hypervisor [56, 90], which properly enforces the isolation between running VMs. By focusing on fine-grained user-mode process-level monitoring, process out-grafting does not address OS kernel monitoring. But a compromised OS kernel should not affect our design goals. Also, we do not attempt to hide the fact that an out-grafted process or malware is being actively monitored. But we do guarantee that the monitoring process itself cannot be disabled by the monitored process.

Figure 2.1 shows the overview of process out-grafting with two key techniques: *on-demand grafting* and *split execution*. Before presenting each technique in detail, we define some terminology used throughout this chapter. Process out-grafting is designed to work with hardware virtualization extensions available in commodity CPUs, including support for efficient memory virtualization. Leveraging the underlying hardware support, a CPU can enter either host or guest mode. The hypervisor code runs directly in host mode while the VM runs in guest mode. In a virtualized system, there are three different kinds of memory addresses: A *guest-virtual* address is the virtual address observed by a running process inside the guest; a *guest-physical* address is the physical addresses seen by the guest when it runs in the guest mode; a *host-physical* address is the actual machine address seen by the CPU hardware. While executing in guest-mode, a VM never sees the host-physical addresses directly.

We point out that when memory virtualization support is enabled, the CPU utilizes an additional level of page tables managed by the hypervisor to translate from guest-physical to host-physical addresses (i.e., the Nested Page Table (NPT) [20] in AMD CPUs or the Extended Page Table (EPT) [48] in Intel CPUs). Since our current prototype uses an Intel processor, we simply use the term EPT to represent both. With that, the guest OS is free to manage its own

guest-physical page frame allocation, with no intervention from the hypervisor. The hypervisor controls only the EPT to allocate host-physical memory, as needed for the guest. While the CPU EPT support significantly improves performance for the running VM [19, 20, 48], such support also poses challenges and some of them will be highlighted below as we describe our design.

2.2.2 On-demand Grafting

Our first key technique is developed to re-locate a suspect process running in a production VM to a security VM for close inspection. Specifically, it enables efficient, native inspection from existing process-level monitoring tools by avoiding unnecessary hypervisor intervention and eliminating the inherent semantic gap from VM isolation. Relocating a suspect process can be initiated as determined by an administration policy, say a process can be brought under scrutiny either periodically, at random time intervals or using certain event triggers. The monitoring duration can be arbitrary, including the entire lifetime of a process. Regardless of the out-grafting policy, in this chapter, we mainly focus on the mechanisms for the out-grafting support.

In order to out-graft a running process, we will need to first accurately locate it (e.g. using the base address of its page table directory). Once it is located, the hypervisor can then redirect or transfer its execution from the production VM into the security VM. In the following, we examine when, what, and how to transfer the suspect process execution across the two VMs.

When to Out-graft

To determine the appropriate moment for process execution transfer, we need to ensure it is safe to do so, i.e., the transfer will not corrupt the execution of the out-grafted process and the OS kernel. Particularly, once a process is selected for out-grafting, the hypervisor first pauses the production VM, which is akin to a *VM Exit* event and causes the VM's virtual CPU (VCPU) state to be stored in hypervisor-accessible memory. At this particular time, the to-be-grafted process may be running in either user- or kernel-mode. (If it is not actively running, it is then waiting in the kernel-mode to be selected or dispatched for execution.) If the VCPU state indicates the VCPU is executing the process in user mode, we can immediately start out-grafting the process.

On the other hand, if the VCPU was running in privileged mode (in the context of either the suspect process or another process), we should not start the out-grafting process to avoid leading to any inconsistency. For example, the suspect process may have made a system call to write a large memory buffer to a disk file. If its execution is transferred at this point to another VM, we may somehow immediately resume execution (in the security VM) at the next user-mode

instruction following the system call. As we are only transferring the user-mode execution, this will implicitly assume the production VM kernel has already completed servicing the system call, which may not be the case. Therefore, we choose to wait till the process is selected to execute and eventually returns to user-mode. One way the hypervisor could detect this is by monitoring context switches that occur inside the VM. However, in systems that support EPT, the hypervisor is no longer notified of in-VM context switches. Instead, based on the process' page tables, we mark the corresponding user-level host-physical pages non-executable (NX) in the EPT. When the kernel returns control of the process back to user-mode, it will immediately cause a trap to the hypervisor and thus kick off our out-grafting process.

What to Out-graft

After determining the right moment, we then identify the relevant state that is needed to continue the process execution in the security VM. As our focus is primarily on its user-mode execution, we need to transfer execution states that the user-mode code can directly access (i.e. its code and data). It turns out that we only need to transfer two sets of states associated with the process: the execution context (e.g., register content) and its memory page frames. The hypervisor already identifies the process' register values from the VCPU state (stored in the hypervisor-accessible memory). To identify its memory page frames, we simply walk through the guest OS-maintained page tables (located from the guest *CR3*) to identify the guest-physical page frames for the user-mode memory of the process. For each such page, we then further identify the corresponding host-physical page frame from the EPT. At the same time, we mark NX bit on each user-mode page frame in EPT that belongs to the process. (Although this seems to duplicate the setting from Section 2.2.2, this is required in case the guest OS may allocate new pages right before we start to out-graft.) After that, if the user-level code is executed, inadvertently or maliciously, when the process has been out-grafted for monitoring, the hypervisor will be notified. Note that we mark the NX bit in the EPT, which is protected from the (untrusted) production VM.

We point out that the transferred resources or state do not include those OS kernel-specific states, which the process may access only via system calls (e.g. open file descriptors and active TCP network connections). This is important from at least three different aspects. First, the OS kernel-specific states are the main root cause behind the semantic gap challenge. Without the need of interpreting them, we can effectively remove the gap. Second, keeping these specific states within the production VM is also necessary to ensure smooth continued execution of the out-grafted process in the security VM – as the system call will be redirected back to the production VM. It also allows for later process restoration. Third, it reduces the state volume that needs to be transferred and thus alleviates the out-grafting overhead.

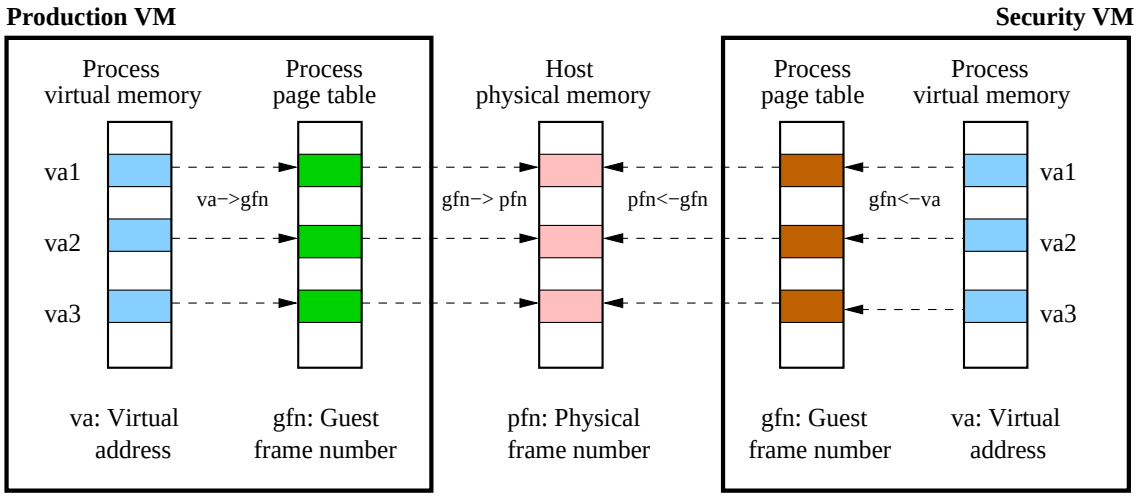

Figure 2.2: Out-grafted process memory mapping in production and security VMs

How to Out-graft

Once we identify those states (e.g., the execution state and related memory pages), we then accordingly instantiate them in the security VM. There are two main steps. First, we lock down the guest page table of the out-grafted process in the production VM. Specifically, we mark the page frames that contain the process' page table entries as read-only in the EPT so that any intended or unintended changes to them (e.g. allocating a new page or swapping out an existing page) will be trapped by the hypervisor. This is needed to keep in-sync with the out-grafted process in the security VM. These hardware-related settings are the only interposition we need from the hypervisor, which are completely transparent to, and independent of the monitoring tools (in the security VM). The lock-down of page tables is due to the previously described lack of hypervisor intervention over in-guest page tables. In our design, these settings are temporary and only last for the duration of our monitoring.

Second, we then populate the transferred states in the security VM. For simplicity, we collectively refer to those states as S_{rd}. For this, we prepare a helper kernel module (LKM) running inside the security VM. The hypervisor issues an upcall to the helper module to instantiate S_{rd}. In that case, the helper module retrieves S_{rd} and creates a process context within the security VM for the out-grafted process to execute. At this point, the memory content of the process needs to be transferred from the production VM to the security VM. In a non-EPT supported system, the hypervisor could simply duplicate the page table between the production and security VMs. In the presence of EPT however, we aim to avoid large memory transfers by enabling the memory transfer as follows: The helper module allocates the guest-physical page frames for those virtual addresses that were present in S_{rd} and sends this information to the

hypervisor; The hypervisor then simply maps each such page to the host-physical page frame for the corresponding virtual address in the production VM. In other words, this mechanism ensures that a user-level virtual address A of the out-grafted process in the security VM and the user-level virtual address A of the process in the production VM are ultimately mapped to the same host-physical page (as illustrated in Figure 2.2). After that, the helper also ensures that any system call from the out-grafted process will not be serviced in the security VM. Instead, they will be forwarded back to the production VM and handled by our second key technique (Section 2.2.3).

When a process is out-grafted for monitoring, its state in the production VM is not destroyed. As mentioned earlier, the production VM still maintains the related kernel state, which not only serves the forwarded system calls but also greatly facilitates the later restoration of the process from the security VM back to the production VM. Meanwhile, because of the separate maintenance of the process page tables inside both VMs, we need to ensure they are kept in-sync. In particular, the production VM may make legitimate changes (e.g. swapping out a page). To reflect these changes back in the security VM, our previous read-only marking on related page tables can timely intercept any changes and then communicate the changes back to the security VM.

With the populated states in a new process inside the security VM, existing process-level monitoring tools such as *strace*, *ltrace*, and *gdb* can naturally access its state or monitor its execution. For example, when the out-grafted process executes system call instructions in the security VM (although they are not actually serviced by the security VM kernel), these can be examined in a semantically-rich manner (i.e., interpreting the arguments) without any modification to existing tools. Specifically, different from prior out-of-VM approaches, the monitor in our case no longer needs to walk through external page tables to identify the physical addresses for examination. In other words, they can be transparently supported! Finally, in order to support tools that may need to access disk files used by the monitored process, we make the file system that is used by the production VM available in the security VM. We mount this file system as read-only and non-executable. Note that the file system is accessed only by the monitor to access any semantic information. The requests by the out-grafted process to access files are not handled in the security VM, but in the production VM through forwarded system calls.

2.2.3 Mode-sensitive Split Execution

After selecting and out-grafting a process to the security VM, our second key technique ensures that it can smoothly continue its execution in the security VM, even though the out-grafted process may consider itself still running inside the same production VM. Also, we ensure that the

untrusted process cannot tamper with the security VM, including the security VM's kernel and the runtime environment (libraries, log files etc.). We achieve this by splitting the monitored process' execution between the two VMs: all user-mode instructions execute in the security VM while the rest execute in the production VM. In the following, we describe related issues in realizing this mechanism and our solutions.

System Call Redirection

To continue the out-grafted process execution and isolate it from the security VM, there is a need for it to access the kernel-specific resources or states maintained in the production VM. For instance, if the process already opened a file for writing data, after the relocation to the security VM, it must be able to continue writing to it. As the process needs to make system calls to access them, we therefore intercept and forward any system call from the out-grafted process back to the production VM.

To achieve that, there exist two different approaches. The first one is to simply ask hypervisor to intervene and forward the system call (by crafting an interrupt and preparing the appropriate execution context). However, it will unfortunately impact the entire production VM execution. The second one is to have a small piece of stub in place of the out-grafted process. The stub is mainly designed to receive forwarded system calls from the security VM, invoke the same in the production VM, and then return the results back to the security VM. We take the second approach in our design as it can effectively localize the effect within the out-grafted process itself and avoid heavy hypervisor intervention for every forwarded system call.

The placement of the stub code deserves additional consideration. Since the guest page tables are not managed by the hypervisor, it cannot simply allocate a separate guest-physical page for the stub code. As our solution, we choose to temporarily "steal" an existing code page in the process, by saving the original content aside and overlaying it with the stub's code. Recall (from Section 2.2.2) that the host-physical memory frames corresponding to the process address space are mapped in both VMs. To steal a code page, the corresponding host-physical page frame is replaced with another one that contains the stub code. To protect it from being tampered by the production VM, we mark it non-writable in the production VM's EPT for the duration of out-grafting.

The stub's main function is to proxy the forwarded system call from the security VM and replay it in the production VM. In order to facilitate direct communication without requiring hypervisor intervention, during the out-grafting phase, we set up a small shared communication buffer accessible to the stub and the helper module in the security VM. Also, note that if a system call argument is a pointer to some content in the process address space, our design

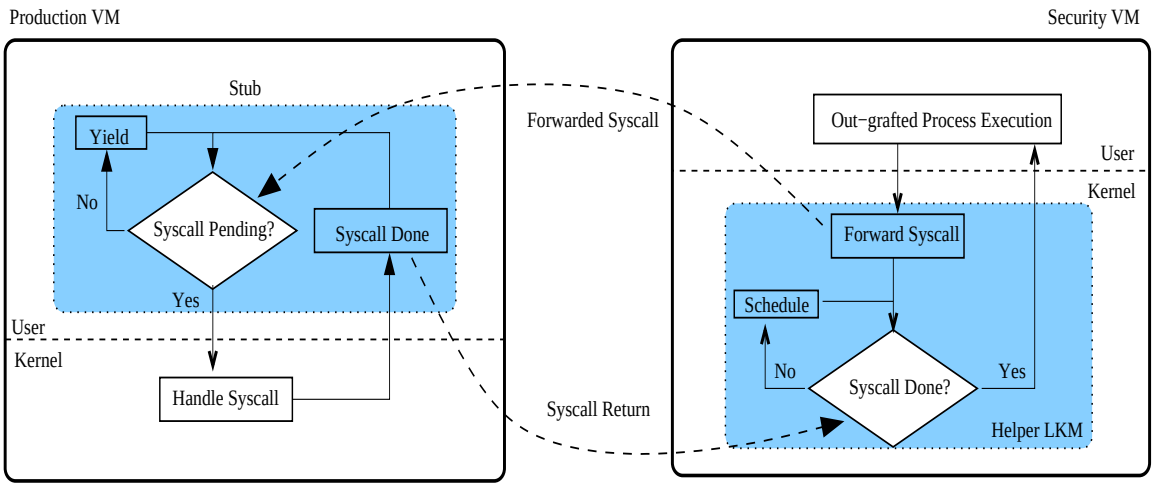

Figure 2.3: Interplay between the production VM stub and the security VM helper LKM

ensures that there is no need to copy this data between the production and security VM. Since the process' memory is mapped in both VMs, the production VM kernel can simply access this as it would for any regular process. This eliminates unnecessary memory copy overhead.

An interesting dilemma arises if the stub code needs to use the stack. Specifically, the process' stack is part of its data pages and these pages are mapped faithfully in both VMs. Any user-space instructions in the out-grafted process will use this memory region as the stack. Further, these memory regions may also contain the arguments to be passed down to the production VM kernel for system calls. Hence, the stub code cannot use the original stack pointer as its stack frame, or it will collide with the user-mode execution in the security VM. In our design, we simply avoid using the stack in the stub code execution.

The interplay between the stub code and our helper module is shown in Figure 2.3. The stub code is self-contained and will directly invoke system calls without relying on any additional library calls. Its size has been kept to the minimum since we are overlaying over existing code memory for this. Also, the stub can handle signals from the production VM kernel and communicate them to the security VM helper module. Specifically, when the production VM invokes a previously registered signal handler, it will cause an exit to the hypervisor, which will be eventually relayed to the helper module to deliver the same signal to the out-grafted process for handling.

Page Fault Forwarding

In addition to forwarding system calls, we also need to forward the related page faults for the out-grafted process. More specifically, at the time when a process' execution is being redirected

to the security VM, some of its pages may not be present (e.g., due to demand paging). Hence, when the process is executing in the security VM, if it accesses a non-present page, it will be trapped. Since the security VM has no knowledge of how to correctly allocate and populate a new page (e.g., in case of a file-mapped page), we forward page faults to the production VM. This also ensures that when the production VM kernel attempts to access a new memory page during process execution, it will be immediately available.

To forward related page faults, out helper module in the security VM registers itself to receive notification from the security VM kernel for any page fault (or protection fault) related to the out-grafted process. When it receives such a notification, it places the virtual address that causes the page fault and a flag to indicate a read-fault (including instruction fetches) or a write-fault, in the communication buffer and notifies the stub in the production VM. To service a page fault, the stub will attempt to either read in a byte from the address or write a byte to the address. This will cause the same page fault in the production VM. After completing this read or write, the stub sends a notification back to our helper module. The stub's write to the memory will be immediately overwritten with the correct value by the out-grafted processes when it re-executes a faulting instruction, thus ensuring correct process execution. Unlike a system call return value, the stub cannot return the guest-physical page allocated for the new address in the production VM. Instead, since we have write-protected the process page tables, when the production VM makes a change to it (i.e. to set the page table entries for the new page), this is intercepted by the hypervisor and our helper module will be notified. After that, it then allows the page fault handling routines in the security VM to continue processing. As previously described, instead of duplicating a memory page, we simply tell the hypervisor to map the same host-physical memory page corresponding to new page in the production VM.

Later on, if we decide not to continue monitoring the out-grafted process, we can place it back in the production VM. In this case, since we have already maintained synchronized page tables between the two VMs, we only need to restore the execution context states back in the production VM. For those memory pages overlaid for the stub use, they need to be properly restored as well. Specifically, if we find the VCPU was executing stub code, which is located in the user mode, we can simply change the VCPU state contents to restore the execution context. Otherwise, we need to detect when the user-mode execution resumes by marking the stub's code page as NX and then restore the VCPU values. In the security VM, any state for the out-grafted process is then simply destroyed (i.e. guest-physical page frames allocated to it are freed and helper module requests the hypervisor to release the mappings in the security VM EPT).

2.3 Implementation

We have implemented a process out-grafting prototype by extending the open-source KVM [5] hypervisor (version 2.6.36.1). All modifications required to support out-grafting are contained within the KVM module and no changes are required to the host OS. Our implementation increases KVM's 36.5K SLOC (x86 support only) by only 1309 SLOC, since most functionality required for out-grafting (e.g. manipulating a VM's architectural state) is already present in the stock KVM. Our prototyping machine runs Ubuntu 10.04 (Linux kernel 2.6.28) with an Intel Core i7 CPU and hardware virtualization support (including EPT). Though our prototype is developed on KVM, we believe it can be similarly implemented on other hypervisors such as Xen and VMware ESX. Our current prototype supports both 32-bit Fedora 10 and Ubuntu 9.04 as guests (either as the production VM or the security VM). In the rest of this section, we present details about our KVM-based prototype with Intel VT support.

2.3.1 On-demand Grafting

KVM is implemented as a loadable kernel module (LKM), which once loaded extends the host into a hypervisor. Each KVM-powered guest VM runs as a process on the host OS, which can execute privileged instructions directly on the CPU (in the so-called guest mode based on hardware virtualization support). The VM's virtual devices are emulated by a user-level program called QEMU [10] that has a well-defined *ioctl*-based interface to interact with the KVM module. In our prototype, we extend the interface to define the out-grafting command *graft_process*. This command accepts a valid page table base address (or guest *cr3*) to directly locate the process that needs to be out-grafted. The guest *cr3* could be retrieved either by converting from the process ID or process name via VM introspection, or directly reading from the guest kernel with a loadable kernel module.

Once the *graft_process* command is issued, the KVM module pauses the VM execution, which automatically saves the VM execution context information in the VM control structure (VMCS [49]). KVM provides a number of functions that wrap CPU-specific instructions to read and write to different fields in this VMCS structure. From this structure, we retrieve the current value of the VM's *cr3* register and compare with the argument to the *graft_process*. If they match and the current VCPU is running in user-mode, we then read the VCPU's register values (eax, ebx, esp etc.) and store them in memory. Otherwise, we take a walk through the process page table from the given base address to locate all the user-mode guest-physical page frames (*gfn*) used by the to-be-grafted process. Using the *gfn_to_pfn()* function in KVM, we obtain the host-physical page frames (*pfn*) and set the NX bit in the VM's EPT. As a result, when the VCPU returns back to execute any user-mode instruction in this process, an *EPT violation* occurs and the control is transferred to KVM.

At this point, KVM can read the register values from VMCS and save a copy. After retrieving its execution context (i.e., those register values), we then examine the page table (from the given base address) to determine which virtual addresses in the suspect process have pages allocated or present. That is, it determines the *gfn* for each virtual address and then invokes *gfn_to_pfn()* to determine the corresponding *pfn* in the production VM's EPT. KVM stores this information in a local buffer. It then makes an upcall in the form of a virtual IRQ to the security VM (using the *inject_irq()* function).

In the security VM, we have implemented a helper module which is registered to handle this (virtual) IRQ. The helper module then instantiates a process context for the out-grafted process so that it can continue the execution. Specifically, it first allocates a memory buffer and makes a hypercall to KVM with the address of this memory buffer, so that KVM can copy the state S_{rd} to it. Since spawning a new process is a complex task, our helper module creates a simple "dummy" process in the security VM, which executes in an infinite sleep loop. Upon the IRQ request from KVM, it proceeds to replace the dummy process with the out-grafted process state. Specifically, our helper module retrieves register values from S_{rd} and instantiates them in the dummy process (using the *pt_regs* structure) After that, we simply destroy any pages (i.e., via *do_unmap()*) allocated to the dummy process and then allocate "new" pages for virtual addresses, as indicated in S_{rd}. Note that we do not actually allocate new host-physical memory pages to accommodate these transferred memory pages. Instead, KVM simply maps (*__direct_map()*) those host-physical memory frames that are used by the out-grafted process to the dummy process in the security VM. After the mapping, the out-grafted process is ready to execute its user-mode instructions in the security VM. An execution monitor (such as *strace*) in this security VM can now intercept the process-related events it is interested in.

2.3.2 Mode-sensitive Split Execution

After relocating the suspect process to the security VM, any system call made from it will be intercepted by our helper module and forwarded back to the production VM. Specifically, our helper module wraps the exposed system call interface in the security VM to the out-grafted process. For each intercepted system call, we collect the corresponding system call number and its argument values in a data structure (*sc_info*) and save it in the shared communication buffer so the stub code in production VM will pick it up to invoke the actual system call. (As noted in Section 2.2.3, memory contents for pointer arguments are not copied since the user-level memory is present in both VMs).

More specifically, the stub code is created when the process is being out-grafted from the production VM to the security VM. Its main purpose is to proxy the forwarded system calls from the security VM to the production VM. As mentioned in Section 2.2.3, we need to "steal"

an existing code page to host the stub code. We have written the stub code in a few lines of assembly with an overall size of 167 bytes. The stub code itself does not make any use of a stack while executing (Section 2.2.3). Similarly, with the help of KVM, we set up a shared communication buffer between the stub code and our helper module. When a system call is to be forwarded to the production VM, our helper module copies the *sc_info* data structure described above to this buffer. It then sets a flag (in the same buffer) to indicate to the stub that a new system call is to be serviced and waits in a loop for this flag to be cleared by the stub. To avoid blocking the entire security VM during this time, it yields from inside the loop.

The stub code checks the flag and then retrieves the *sc_info* values and copies them to the registers in the production VM. It then invokes the requested system call so that the production kernel can service the request. Once the request is complete, the stub places the return value in the buffer and modifies the flag indicating service completion. After that, our helper module in the security VM can now return the same value to the out-grafted process. In addition to forwarding system calls from the out-grafted process, we also need to forward related page faults to the production VM. Naturally, we leverage the above communication channel between the stub code and our helper module. Specifically, when a page fault occurs in the out-grafted process (while running in the security VM), the security VM's page fault handler invokes a callback defined in our helper module, which then forwards the page fault information to the stub. Based on the fault information, the stub either reads or writes a dummy value in the faulting address in the production VM to trigger a page fault of the same nature in the production VM. When the page fault handler in the production VM attempts to update the page tables with a new page table entry (pointing to a new page frame we denote by *gfn_p*), this causes an "EPT violation" and control is transferred to KVM. KVM examines the root cause and saves a copy of the *gfn_p-¿pfn* mapping while fixing the violation and resuming the production VM. The helper module notifies KVM with the new guest-physical frame it allocated in the security VM (*gfn_s*). KVM then maps *gfn_s* to *pfn* and ensures the same memory content is available in both VMs. After that, the out-grafted process can continue its execution with the new memory page.

2.3.3 Process Restoration

In our prototype, process out-grafting is initiated through a QEMU command *graft_process*. As mentioned earlier, other mechanisms can also be added to trigger the out-grafting process. An example is an event-based trigger that runs inside the production VM (Section 5.4).

In our current prototype, we also implemented another QEMU command *restore_process*, which can be invoked to notify KVM (via an *ioctl* interface) to restore the out-grafted process back to the production VM. Similar to the out-grafting procedure, when KVM receives the

restore_process command, it injects an IRQ to the security VM, which will be received by the helper module. If the module is currently waiting on a forwarded system call's completion, the restoration operation cannot be immediately carried out. Instead, it will wait for the completion of the system call. After that, it fetches register values stored in the process' *pt_regs* data structure and sends this down to KVM with a hypercall. KVM then restores this register state back in the production VM. Due to the similarity with the earlier out-grafting steps, we omit the details here. The key difference however is that, instead of copying values *from* the VCPU fields, we copy values *to* it. For the page tables, as they are kept in-sync between the two VMs, no further actions will be needed. For those process contexts and guest memory pages allocated to the out-grafted process in the security VM, we simply discard them. At this point, the process can seamlessly resume its execution in the production VM.

2.4 Evaluation

In this section, we first perform a security analysis on the isolation property from our approach. Then, we present case studies with a number of execution monitoring tools. Finally, we report the performance overhead with several standard benchmarks.

2.4.1 Security Analysis

Monitor isolation and effectiveness To allow for fine-grained out-of-VM process monitoring, one key goal is to ensure that the process monitoring tool and its supporting environment cannot be tampered with or disabled by the out-grafted process. In the following, we examine possible attacks and show how our design can effectively defend against them. Specifically, one main way that a suspect process can tamper with another process (or the monitor in our case) is through system calls. However, such attack will not work since in our design, we strictly forward all system calls from the suspect process back to the production VM. Moreover, the controlled interaction is only allowed from the monitoring process to the suspect process, *not* the other way around. From another perspective, the suspect process may choose to attack the (production VM) OS kernel when it services system calls (for e.g. exploiting a buffer overflow by sending in an invalid argument). Such attack will only impact the production VM and its own execution.

According to our threat model (Section 5.2), we stress that our design does not attempt to guarantee stealthy monitoring as out-grafted monitoring could be detected by sophisticated malware. But we do enable reliable monitoring in protecting our system from being tampered with by the suspect process. In addition, a strong administrative policy might reduce the time window for such out-grafting detection. For example, out-grafting can be initiated randomly (at

any instance in a process' lifetime) and can span arbitrary durations. In such cases, malware would be forced to continuously check for out-grafting which can be costly and expose its presence. Note that the out-grafted process is ultimately serviced by the untrusted production VM, and as such, we cannot guarantee that it accurately services the out-grafted process' system calls, but any inappropriate handling of system calls will not violate the isolation provided by our approach.

Protection of helper components The out-grafting operation itself is initiated and controlled by the hypervisor and cannot be disabled by a malicious production VM kernel. However, there are two helper components in the production VM to support out-grafting: the stub code and shared communication buffer, which may be open to attack. We point out that since the stub code's host-physical page is marked as read-only in the EPT, any malicious attempts to write to it will be trapped by the hypervisor. The stub code's guest virtual-to-physical mapping cannot be altered by the production VM since the page tables of the process are write-protected by the hypervisor. The untrusted kernel in the production VM is responsible for scheduling the stub process as well as saving and restoring its execution context according to the scheduling policy. If it tampers with the execution context states (such as the instruction pointer), then the stub code itself will not execute correctly, which cascadingly affects the execution of the suspect process itself. As mentioned earlier, if the production VM does not properly serve the forwarded system calls or attempts to alter the system call arguments or return incorrect results, such behavior may result in incorrect execution for the out-grafted process, but will not affect the isolation or the integrity of our monitoring process.

2.4.2 Case Studies

Next, we describe experiments with a number of execution monitoring tools, including the most common ones: *strace*, *ltrace*, *gdb*, as well as an OmniUnpack[59]-based tool (to detect malware unpacking behavior). The common tools are used to demonstrate the effectiveness of our approach in removing the semantic-gap, while the OmniUnpack tool requires special hardware support for the monitoring in the security VM and such support may not be enabled or provided in production VM. As a test process for out-grafting, we chose the *thttpd* web server that uses both disk and network resources and has a performance benchmark tool readily available to automatically exercise it.

Tracing System Calls

In our first experiment, we demonstrate semantically-rich system call tracing. This type of monitoring has been widely applied to detect malicious behavior [38] such as accesses to sensitive resources or dangerous system calls. For this, we install the standard Linux *strace* tool in the

security VM. *strace* makes use of the underlying OS facilities to monitor system calls invoked by another running process, which in our case logically runs in another VM. For each intercepted system call, it retrieves and parses the arguments. The results will allow us to know what file was opened by a process, what data is read from it etc.

In prior "out-of-VM" systems, the code to determine system call number and interpret each of its arguments has to be completely re-written. In fact, one of our earlier systems, i.e., VMscope [50], took one of the co-authors more than one month to correctly intercept and parse the arguments of around 300 system calls supported in recent Linux kernels. This task is expected to become even more complicated especially for closed-source OSs.

To better understand the effectiveness, we perform a comparative study. Specifically, we first run *strace* inside the production VM to monitor invoked system calls from *thttp* when it handles an incoming HTTP request. After that, we out-graft *thttp* and run *strace* in the security VM for out-of-VM monitoring when it handles another incoming HTTP request. We verify that both *strace* runs lead to the same system call patterns in the handling of incoming HTTP requests by accurately capturing system calls invoked by the same *thttpd* process and interpreting each related argument.

Tracing User-level Library Calls

In our second experiment, we show the capability of reusing existing tools for user-level library call tracing. User-level library call tracing is a fine-grained monitoring technique that allows for understanding which library functions are being used by a running process and what are their arguments. It has advantages over system call tracing in collecting semantically-rich information at a higher abstraction level.

As one can envision, the number of library functions available to a program and the type and definition of each argument for such functions can be very large. This can make it complex, expensive, or even impossible to examine such events in a semantically-rich manner using prior "out-of-VM" approaches. Fortunately, in our approach, we can simply re-use an existing tool *ltrace* to intercept and interpret user-level library calls of a running process in one VM from another different VM. In our experiment, we found that *ltrace* extracts process symbol information from the process' binary image on disk. As we mounted the production VM's filesystem read-only in the security VM, *ltrace* works naturally with no changes needed. Our results show that *ltrace* indeed accurately captures and interprets the user-level library calls invoked by the out-grafted *thttpd* process. In a similar setting, we replace *ltrace* with *gdb*, which essentially allows for debugging an in-VM process from outside the VM!

Detecting Malware Unpacking Behavior

Most recent malware apply obfuscation techniques to evade existing malware detection tools. Code packing is one of the popular obfuscation techniques [47]. To detect packed code, efficient behavioral monitoring techniques such as OmniUnpack [59] have been developed to perform real-time monitoring of a process' behavior by tracking the pages it writes to and then executes from. When the process invokes a "dangerous" system call, OmniUnpack looks up its page list to determine whether any previously written page has been executed from. If so, this indicates packing behavior, at which point a signature-based anti-virus tool can be invoked to check the process' memory for known malware. Since OmniUnpack was developed only for Windows and also is not open source, we wrote a Linux tool that faithfully implements OmniUnpack's algorithm. We stress that if OmniUnpack was previously available for Linux, this porting step would not be necessary. In our Linux porting, we do not need to bridge the semantic gap or be aware of any prior introspection techniques. Instead, we just envision a Linux tool that will be used *in-host*. This experience also demonstrates the benefits from our approach.

In our test, we use the freely available UPX packer [16] to pack the Kaiten bot binary [4]. In this experiment, we also utilize a security-sensitive event trigger that initiates out-grafting when a suspect process invokes the *sys_execve* system call. The trigger is placed such that just before the system call returns to user-mode (to execute the first instruction of the new code), KVM is invoked to out-graft the process' execution to the security VM. Inside the security VM, we run the OmniUnpack tool to keep track of page accesses by the process. Since the system calls invoked by the process are also available for monitoring, OmniUnpack successfully detects the packing behavior.

We highlight several interesting aspects this experiment demonstrates. In the past, packer detection has required a trade-off between tool isolation and performance overhead. Specifically, in-host tools [59, 77], including OmniUnpack can efficiently detect packing behavior, but they are vulnerable to attack. "Out-of-VM" techniques [34] ensure packer detection is isolated, but introduce very high overhead, largely limiting its usability for offline malware analysis, not online monitoring. Using process out-grafting, we are able to effectively move an in-host tool "out-of-VM" without introducing significant overhead while still providing the needed isolation. Another interesting aspect is due to the fact that OmniUnpack requires the NX bit in the guest-page tables. For 32-bit Linux, this bit is available only if PAE support is enabled. In our experiments, we enabled PAE only in the security VM, whereas in the production VM page tables, the NX bit is still not available. Thus, if the monitoring tool requires additional features or support from the underlying OS, even if this support is not present in the production VM, we can take advantage of it in the security VM. Also, we point out that the process out-grafting happens at the very beginning of its execution. When a process begins execution, most of its

Table 2.1: Software packages used in our evaluation of process out-grafting

Name	Version	Configuration
Host OS	Ubuntu 10.04	Linux-2.6.32
Guest OS	Fedora 10	Linux-2.6.27
SPEC CPU 2006	1.0.1	integer suite
Apache	2.2.10	ab -c 3 -t 60
thttpd	2.25b	ab -c 1 -t 60

code pages are not yet mapped in by the OS. As such, this experiment also thoroughly tests the page fault forwarding mechanisms in our design (Section 2.2.3).

2.4.3 Performance

To evaluate the performance overhead of process out-grafting, we measure two different aspects: the slowdown experienced by the production VM when a process is out-grafted for monitoring as well as the slowdown to the out-grafted process itself. The platform we use is a Dell T1500 system containing an Intel Core i7 processor with 4 cores, running at 2.6 GHz, and 4 GB RAM. The host OS is 32-bit Ubuntu 10.04 (Linux kernel 2.6.32) and the guests run 32-bit Fedora 10 (Linux kernel 2.6.27). Both VMs are configured with 1 virtual CPU each. The production VM is configured with 2047 MB memory and the security VM is configured with 1 GB memory. Table 2.1 lists the detailed configuration. In all experiments, the two VMs are pinned to run on separate CPU cores in the host (using the Linux *taskset* command).

Production VM overhead First, we measure the slowdown experienced by the VM (i.e. other normal processes running in it) when we out-graft an unrelated process for monitoring. Specifically, we choose a standard CPU benchmark program, i.e., SPEC CPU 2006, and run it twice (1) either with another CPU-intensive process that spins in an infinite loop inside the same VM (2) or with the CPU-intensive process out-grafted to another VM. Our results show benchmarks experience speedups after out-grafting the CPU-intensive process. This is expected as a contending process has been moved to execute in a different VM for monitoring, which is running on a different core. This also confirms the monitoring overhead has been localized inside the security VM, not the production VM. Next, we measure the impact to *Apache* and *pigz* (or parallel *gzip*) when they run together either with *thttp* out-grafted or not. In our experiments, *Apache* and *thttpd* listen on different TCP ports, but their network traffic is handled by the same production VM kernel. Our results are shown in Figure 4.6. It is interesting to note that when *thttpd* is out-grafted, it is scheduled more often (since both *Apache* and *pigz* dominate it when they run in the same VM). However, the redirected system calls from *thttpd* will not get serviced until the stub is scheduled for execution in the production VM, thus its impact

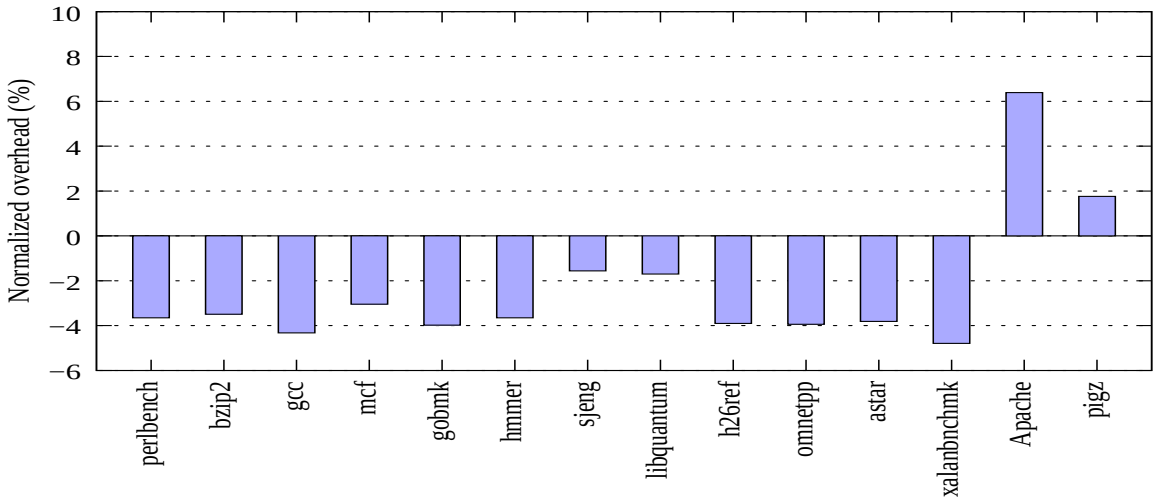

Figure 2.4: Production VM slowdown with a contending process out-grafted

to the dominant processes is still low. Finally, we also measured the time it takes to identify the state for out-grafting during which time the production VM is paused. While this would vary depending on the memory size of a process, we observed an average time of $\sim 250\mu$s in our current experiments. Opportunities still exist to further reduce it (e.g., with lazy updates – Section 2.5).

Out-grafted process slowdown Second, we measure the slowdown an out-grafted process may experience due to the fact that it is running in a different VM. For this, we first measure slowdown in two out-grafted processes: (1) The first one is a simple file copy command that transfers a *tar* file (421MB) from one directory to another in the production VM, which will result in lots of file-accessing system calls being forwarded. (2) The second is *thttpd*, for which we generate traffic using the *ab* benchmark program. In each set of experiments above, we have rebooted both VMs and physical machine after each run to avoid any caching interference. The average slowdown experienced by them is 35.42% and 7.38%, respectively. Moreover, we run a micro-benchmark program to measure the system call delay experienced by *sys_getpid()*. Our result shows that the average time of invoking *sys_getpid()* is $\sim 11\mu$s. As *sys_getpid()* simply obtains the process ID (after it has been forwarded), its slowdown is approximately equivalent to the system call forwarding latency.

2.5 Discussion

While our prototype demonstrates promising effectiveness and practicality, it still has several limitations that can be improved. For example, our current prototype only supports out-

grafting of a single process. A natural extension will be the support of multiple processes for simultaneous out-grafting. Note it is cumbersome and inefficient to iterate the out-grafting operation for each individual process. From another perspective, simultaneous out-grafting of multiple processes can also lead to the unique scenario where multiple security VMs can be engaged in monitoring different groups of out-grafted processes or different aspects of behavior.

Also, our current way of handling *sys_execve()* can be improved. Specifically, as this system call will completely change the memory layout of the out-grafted process, our current prototype simply chooses to first restore the process back to the production VM and then out-graft again the process immediately after this system call is completed by the production VM. Though this approach can leverage the functionalities we already implemented for *graft_process* and *restore_process*, an integrated solution is still desired. Moreover, our current prototype proactively maps all the (user-mode) memory pages at the very beginning when a process is out-grafted. A better solution will be to only map the currently executing code page to the security VM. For the rest of the pages, they can be lazily mapped when they are being actually accessed. This could further improve our system performance.

One caveat we experienced in our prototype development is related to shared pages. Specifically, most commodity OSs map the same physical pages for common shared library code among different processes. In our design, this means that a single host-physical page can contain code that is used across multiple processes in the production VM. Recall that our design directly maps this host-physical page to the out-grafted process in the security VM. If the monitoring tool modifies such code page (say to install certain code hooks), this could alter other process' behavior in the production VM. Fortunately, this can be resolved in a straightforward manner by co-operating the helper module and KVM to mark all executable code pages for the out-grafted process as read-only in the EPT. By applying the classic copy-on-write technique, if the monitor process (*not* the out-grafted process) attempts to write to this page, a separate copy of the page can be created.

Finally, with the wide adoption of virtualization in data centers, we also envision that different security VMs can be dispatched to each physical machine to inspect running guest VMs and their internal processes (for fine-grained execution monitoring). This is largely feasible as the semantic gap has been effectively bridged to support existing monitoring tools. On the other hand, with our current focus on examining individual suspect process for malicious behavior, we believe other interesting applications and opportunities (e.g., performance monitoring and intelligent parallel job scheduling) remain, which we plan to explore in the future.

2.6 Related Work

Virtualization has been widely proposed to address various computer system problems, including enhancing the effectiveness and robustness of host-based monitoring tools. Specifically, it has been applied in offline malware analysis [24, 34], honeypot-based malware capture [50], intrusion analysis [53, 55] and malware detection [42, 51]. Among the most notable, Livewire [42] pioneered the concept of placing a monitor "out-of-VM" and applying VM introspection techniques to understand in-VM activities. A number of recent systems address the inherent semantic gap challenge to improve VM introspection for various purposes [21, 32, 51, 67, 68, 85]. For instance, one recent work Virtuoso [35] aims to effectively automate the process of building introspection-based security tools. Another system [32] proposes injecting stealthy agents into a monitored VM to solve the semantic gap problem and enhance out-of-VM tools. Similar to most of these efforts, our approach places security tools out-of-VM. However, our approach mainly differs from them in the way to address the semantic gap challenge. In particular, while prior approaches are sensitive to particular guest kernel versions or patches, our approach *brings* the suspect process to the security tool and allows for native support of existing tools. In other words, by effectively removing the semantic gap, our approach enables *re-use* of existing user-mode process monitoring tools. Further, our approach localizes the impact on the out-grafted process and avoids perturbing the monitored VM as a whole.

From another perspective, one recent system SIM [79] utilizes hardware features to place an "in-VM" monitor in a hypervisor-protected address space. While it is not physically running out-of-VM, SIM still suffers from the semantic-gap and cannot natively support existing monitoring tools. In other words, though the in-VM presence leads to unique performance benefits, there is a need to adapt existing tools to take advantage of the SIM support. Also, the main goal of SIM is to protect "kernel hook"-based monitors. Another recent system Gateway [86] leverages virtualization to detect kernel malware by monitoring kernel APIs invoked by device drivers. In contrast, our focus is for fine-grained process-level execution monitoring (e.g. *ltrace*) that typically requires user-mode interception. Process implanting [46] is another "in-VM" approach, where an "undercover agent" process is dynamically implanted in a target VM for surveillance and repair operations. Contrary to process out-grafting, process implanting relies on the integrity of the target VM's kernel and requires special modification to the program executed by the implanted process.

Process out-grafting requires redirecting the process execution across two different VMs, which bears certain similarities to well-known process migration mechanisms [65, 66, 80, 81]. However, one key difference is that process migration techniques typically move the entire execution states, including kernel-maintained resources while our approach only (temporarily) redirects the user-level execution of a process for secure monitoring. Moreover, most process

migration techniques are typically applied for generic purposes such as fault-tolerance and load-balancing [65, 80] and do not consider the isolation challenge for secure monitoring.

We also notice that the idea of system call forwarding has been previously applied in systems to protect a critical application from an untrusted kernel [87]. In this case, a programmer can divide system calls into two sets so that each set will be serviced by either a trusted or untrusted kernel, respectively. In contrast, throughout the out-grafting duration, our system has one single kernel to serve all system calls for the out-grafted process. Moreover, due to the relocated user-level execution, we need to perform mode-sensitive split execution, which leads to an extra but unique need of forwarding page faults. System call forwarding between VMs has also been used [60] to improve the fidelity of runtime environment for better malware behavior monitoring, albeit without isolation guarantees. In contrast, we aim to address compatibility of existing process monitoring tools while ensuring their strong isolation in the context of VM introspection. Further, we dynamically create the memory mapping for the out-grafted process between the production VM and security VM. Also, our on-demand grafting allows for dynamically grafting the execution first and then *restoring* the execution back.

More generally, sandboxing and isolation techniques [41, 44, 74] have been widely researched and applied as effective mechanisms to confine an untrusted process' access to sensitive resources in the host system. Our work is related to them by essentially leveraging the VM isolation provided by the underlying virtualization layer. However, with an out-of-VM approach, process out-grafting can be applied on-demand, which provides certain flexibility in monitoring runtime behavior of suspect processes. Also, our approach is unorthodox when compared with traditional sandboxing and isolation techniques due to its split execution, i.e., the user-mode and kernel-mode execution of an out-grafted process run in two different VMs.

2.7 Summary

In this chapter, we have presented the design, implementation and evaluation of process out-grafting. Specifically, we have developed an architectural approach to address isolation and compatibility challenges in out-of-VM approaches for fine-grained process-level execution monitoring. In particular, by effectively relocating a suspect process from a production VM to the security VM for close inspection, process out-grafting effectively removes the semantic gap for native support of existing process monitoring tools. Moreover, by forwarding the system calls from the out-grafted process back to the production VM, it can smoothly continue its execution while still being strictly isolated from the monitoring tool. The evaluation results with a number of performance benchmarks show its effectiveness and practicality. The two key techniques presented in this chapter lay the foundation for our process out-grafting framework. Next, we apply process out-grafting to enable semantically-rich out-of-VM policy enforcement.

Chapter 3

Semantically-Rich Out-of-VM Policy Enforcement

In the previous chapter, we presented process out-grafting to address isolation and compatibility challenges in out-of-VM monitoring. Next, we apply and extend process out-grafting to enable out-of-VM enforcement of semantically-rich policies.

3.1 Background

Today's computers are under constant threat of being targeted by increasingly complex and sophisticated malware. In order to defend against such threats, effective security policy enforcement is critical. Particularly, when we examine behavioral techniques, researchers have proposed defining and enforcing system call policies [41, 44, 74] for a variety of purposes. For example, system call policies have been leveraged for confining untrusted processes and enforcing the expected behavior of well-known applications. By enforcing policies at the system call layer, we can effectively limit the impact of an ongoing malware attack on a vulnerable system.

In this chapter, we extend and apply process out-grafting to enable out-of-VM enforcement of system call policies, which are by definition semantically-rich. While prior research has leveraged system virtualization to provide out-of-VM policy enforcement, they are limited by their fundamental reliance on VM introspection techniques. For example, the VMwall system [85] provides out-of-VM policy enforcement only at the network layer, by filtering and restricting network traffic targeted at a running VM. The Lares [68] system provides an architecture for installing external kernel-mode hooks which are invoked when a specific event occurs. However, Lares relies on VM introspection techniques to provide meaningful policy enforcement which leads to limitations such as the semantic gap. Here, we leverage process out-grafting to address such limitations in prior approaches.

In our framework, similar to the isolation goals presented in Chapter 2, we still protect the out-of-VM policy enforcement tool from tamper by malware targeting the production VM. Further, by extending the process out-grafting techniques to enforce out-of-VM policy, we effectively restrict the behavior of an out-grafted process. Below, we present the design and implementation of our extension to process out-grafting and demonstrate semantically-rich out-of-VM policy enforcement.

3.2 Design and Implementation

Recall that in process out-grafting, we temporarily transfer the user-mode execution of a process from its production VM to a security VM. Further, to enable continued seamless execution of the process, we forward its system calls and page faults back to the production VM. In applying and extending the out-grafting design for out-of-VM policy enforcement, our main goal is to demonstrate that we can effectively limit the impact of a process that violates system policy. Towards this goal, we configure the production VM with a list of executable names and modify the OS kernel such that whenever a process is created, the kernel checks if the corresponding binary file is present in the list. If this is the case, the created process is immediately out-grafted to the security VM. Since the process out-grafting architecture ensures that a process can be out-grafted at the very beginning of its execution, at this point, no instructions (which implies no system calls) have been executed by the process.

Next, we extend the security VM to support out-of-VM system call policy enforcement, which can be implemented by a number of different mechanisms. For example, one option is to extend an existing monitoring tool (such as a system call tracer) to also enforce policy. In our design, we provide flexible policy enforcement of out-grafted processes by modifying the security VM kernel. Specifically, prior to forwarding a system call from the out-grafted process to the production VM, we check the system call and its parameter values against the system call policy. If the specified call is permitted according to the policy, the security VM places the system call information in the shared communication buffer (Section 2.3.2) for servicing by the stub in the production VM. If the specified call is denied according to the policy, the security VM reads the error code from the policy and returns this value to the untrusted process. Thus, the security VM kernel effectively filters system calls executed by the out-grafted process and limits its impact on the production VM.

We leverage the process out-grafting prototype environment (Section 2.3) and use the open-source KVM [5] hypervisor (version 2.6.36.1) for our implementation. In our environment, the host kernel is Ubuntu 10.04 (Linux kernel 2.6.28) and we have experimented with 32-bit Fedora 10 and Ubuntu 9.04 as guests (either as the production VM or the security VM). In our prototype, we have extended the security VM kernel to enforce policies such as those

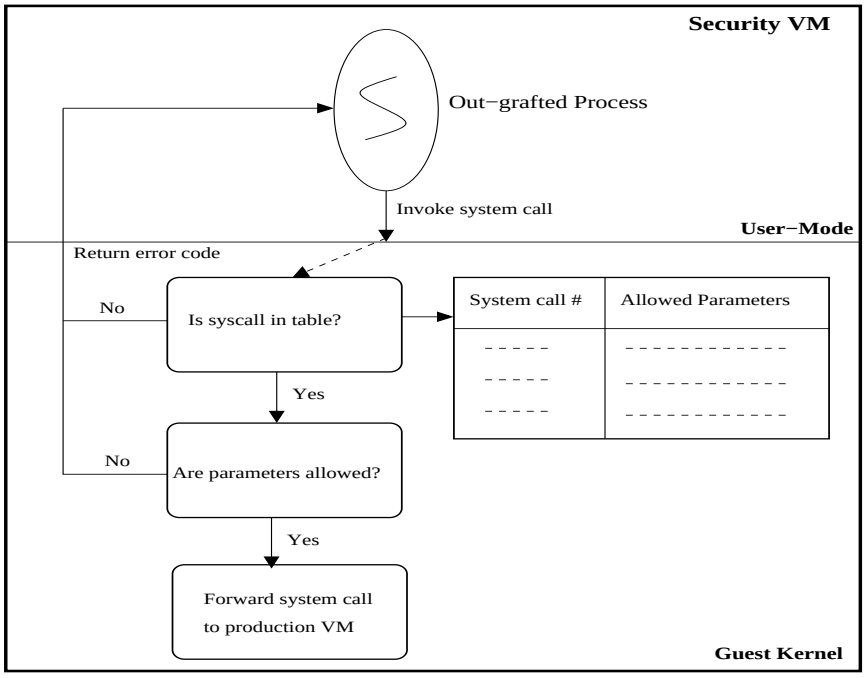

Figure 3.1: Out-of-VM system call policy enforcement

specified for the *systrace* tool [13]. We extend the loadable kernel module (LKM) in the security VM to implement system call policy enforcement, as illustrated in Figure 3.1. For each rule that permits a system call for an out-grafted process, we specify the system call number, allowed parameter values, and error code to be returned upon violation. Naturally, in real-world scenarios, this policy can be read from configuration files that are maintained by management tools.

We extend the out-grafting LKM to register a kernel hook function `enforce_syscall_policy()`. We modify the security VM kernel so that, before processing a system call, it invokes this hook function and passes in the system call number and parameters. Inside the hook function, the LKM checks the process id to see if it matches the out-grafted process. If the process id matches, the LKM proceeds to check the system call against the pre-defined rules in kernel memory. If the system call (and the specified parameter values) is permitted, the LKM places this information in the shared communication buffer (Section 2.3.2) in order to forward the call to the production VM's stub. If the system call has to be denied, we look up the error code and immediately return this value to the user-mode process.

An interesting situation arises when the system call parameter is a pointer to a user-mode memory location (e.g. the pathname parameter in `sys_open()`). For effective policy enforcement, the LKM must be able to access the entire pathname string in the process' memory. In some

cases, the user-mode memory corresponding to this pointer may not be allocated yet to the process, even though the memory region has been mapped and is valid in the process. However, we are still able to enforce policy even in such cases due to process out-grafting's page fault forwarding mechanism. Specifically, when the LKM attempts to read the user-mode memory for policy enforcement, a page fault is triggered and forwarded to the production VM. In the production VM, the kernel allocates the appropriate memory for the user-mode address. When this is complete, the LKM resumes its policy enforcement, and reads the parameter value in memory, which is now present.

3.3 Evaluation

In this section, we present the security analysis and describe our experiments with a prototype implementation.

3.3.1 Security Analysis

By enabling out-of-VM policy enforcement, our main goal is to provide isolation of the enforcement tool. In our design, we guarantee that the policy enforcement cannot be disabled or tampered with, since we leverage the isolation provided by the hypervisor. Naturally, a question arises if enforcing policies for untrusted out-grafted processes within the security VM's kernel introduces vulnerabilities into the security VM environment. In our current design, system call policy enforcement is implemented using simple pattern matching and does not introduce significant additional code. However, in cases where more complex policy enforcement is required, this can be implemented in a user-mode component to avoid introducing vulnerabilities in the kernel.

3.3.2 Effectiveness

To demonstrate the effectiveness of our prototype implementation, we evaluate out-of-VM system call policy enforcement for different applications. First, we illustrate the semantically-rich information available for out-of-VM policy enforcement by presenting results from running the *strace* on the `thttpd` web server. For this, we run `thttpd` inside the production VM and capture the sequence of system calls using *strace*, which is also running in the production VM. The results are shown in Figure 3.2. Next, we outgraft the `thttpd` process to the security VM and capture a similar sequence of system calls using *strace*, which is also running in the security VM. The results are shown in Figure 3.3. Based on these results, which demonstrate the availability of system calls and their associated parameters, we next perform the following experiments.

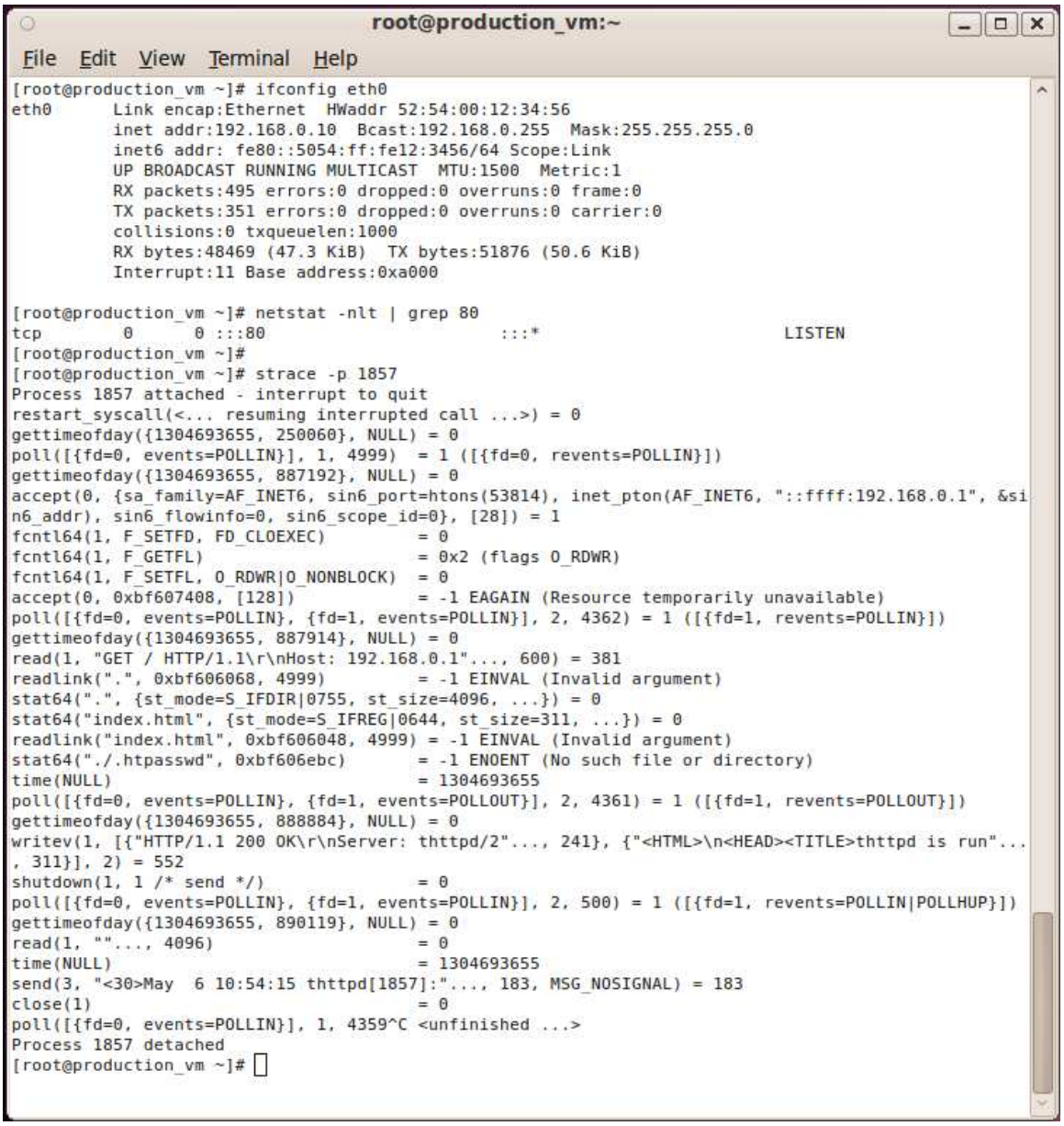

Figure 3.2: Semantically-rich information available for in-VM system call policy enforcement

First, we evaluated policy enforcement in our prototype by using a test process whose file system access is restricted. We configure the security VM with a system call policy which allows the out-grafted process to have access to all files under a "public" directory, but no access to files under a "private" directory. When the process attempts to open a file, we intercept the `sys_open()` call and check the filename and access permissions to ensure that it is allowed. We verify that the appropriate error code (indicating that access to the file is denied) is returned

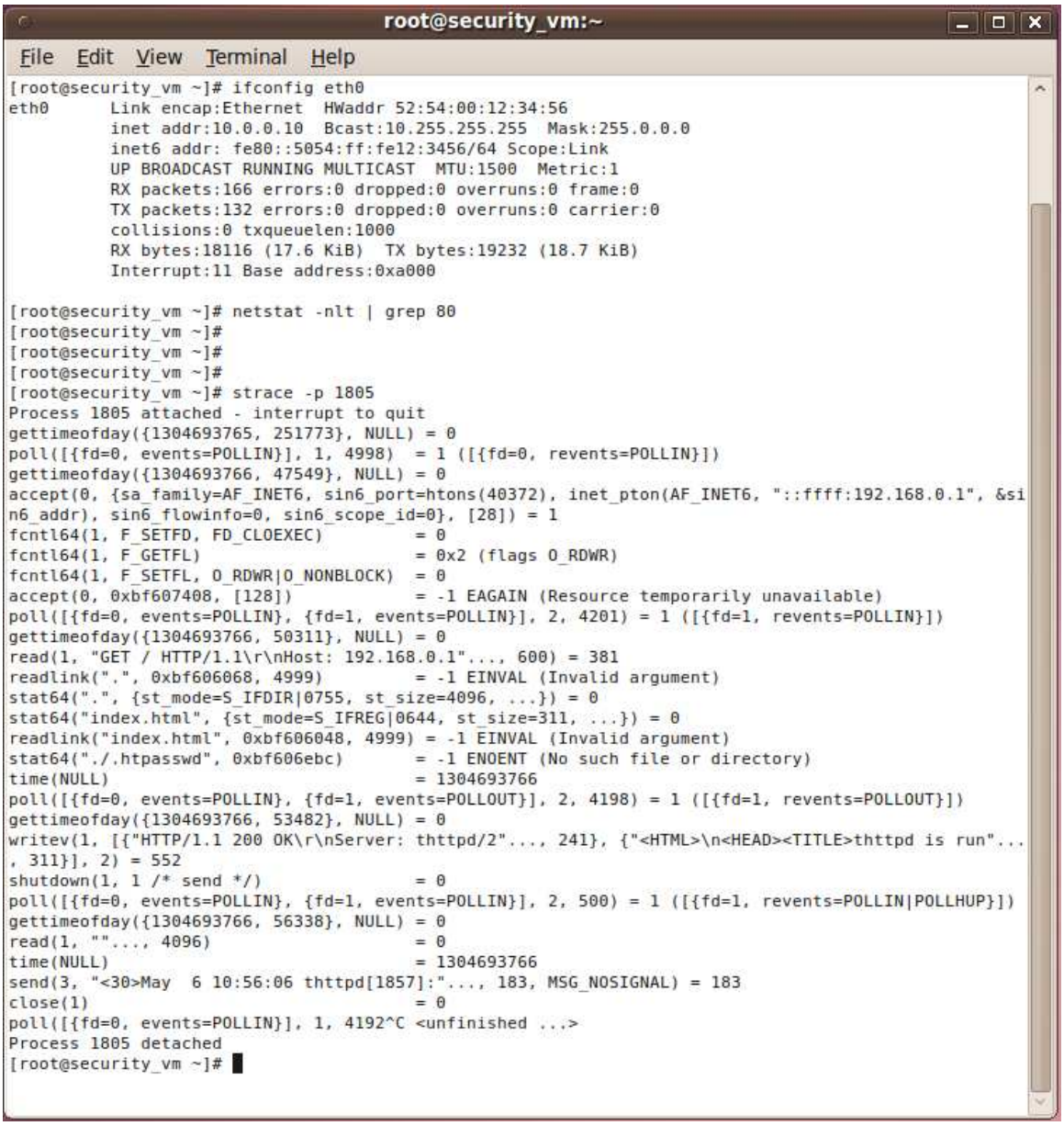

Figure 3.3: Semantically-rich information available for out-of-VM system call policy enforcement

inside the security VM, without having to forward the system call to the production VM. Next, we experimented with a web server policy based on systrace's sample policy for `httpd` [13]. For this, we used `thttpd` as the web server process which is out-grafted to the security VM. We modified the security VM kernel to maintain the system call policy (including allowable parameters for `sys_read()` and `sys_write()`. Next, we exercise the policy by opening a web

browser on a remote system and pointing it to the IP address of the production VM. The default page loads correctly. However, when we try to access a file that is not permitted, the security VM kernel returns the appropriate error code to `thttpd`. We verify this by ensuring an error gets displayed in the web browser. These experiments have demonstrated the effectiveness of extending process out-grafting to enable out-of-VM policy enforcement.

3.4 Discussion

While the system call interface provides a semantically-rich information for behavioral policy enforcement, it is previously known that interposing at this layer has drawbacks [39]. However, these limitations are applicable in general to system call policy enforcement and are orthogonal to our goal for enabling out-of-VM policy enforcement. Further, as presented in Chapter 2, the out-grafted process may be impacted by significant performance overhead while executing in the security VM. Future work can explore optimizations targeted at reducing this overhead. Our current design is well-suited for isolating non-critical untrusted processes which may tolerate performance overheads.

In this chapter, we have demonstrated that process out-grafting can be extended and applied to enable semantically-rich out-of-VM policy enforcement. By actively interposing on the system calls executed by an out-grafted process, we can immediately handle violating system calls without involving the production VM. As future work, we propose exploring techniques to redirect system call violations (such as reading from a sensitive file) to "shadow resources" or generating dummy values for such system calls. Thus, the out-grafted process can continue execution while sensitive resources in the production VM are still protected.

3.5 Summary

In this chapter, we have presented an extension and application of the process out-grafting techniques to successfully enforce out-of-VM policy enforcement. Our experiments with `systrace`-style policies show the effectiveness of our approach. Further, this has demonstrated that the key design techniques in process out-grafting are robust and extensible. In the following chapter, we present the VMsnare component of our framework, which enables capture of processes that violate system policy for live analysis.

Chapter 4

Trapping Intrusions from Production Environments with VMsnare

In the previous chapters, we have presented semantically-rich out-of-VM process monitoring and policy enforcement. Here, we present the VMsnare component of our framework, which builds on existing policy enforcement mechanisms and leverages process out-grafting to effectively capture live intrusions from production environments for detailed analysis.

4.1 Background

When we consider tools available for malware capture and analysis, honeypot systems have been traditionally deployed and used for this purpose. For example, a high-interaction honeypot hosts a commodity system with vulnerable services (that can be remotely exploited), and is deployed with monitoring software to record intruders' behavior [14, 37, 82]. By allowing intruders to completely take over the system while monitoring their behavior, honeypots improve our understanding of attacker motivations and techniques. The advances in system virtualization techniques on commodity platforms have been leveraged to effectively improve honeypot analysis capabilities [36, 50, 83, 88]. However, when using honeypots for intrusion analysis, administrators must still deploy and manage them independently. Attackers may also be able to evade honeypots, thereby reducing their usefulness [57].

On the other hand, malware detection and prevention mechanisms have been used effectively on production systems to block attacks. Well-known techniques such as system call monitoring have been used to prevent malware infections by confining untrusted processes and ensuring that they adhere to pre-defined policies [38, 41, 44, 74]. While malware prevention systems

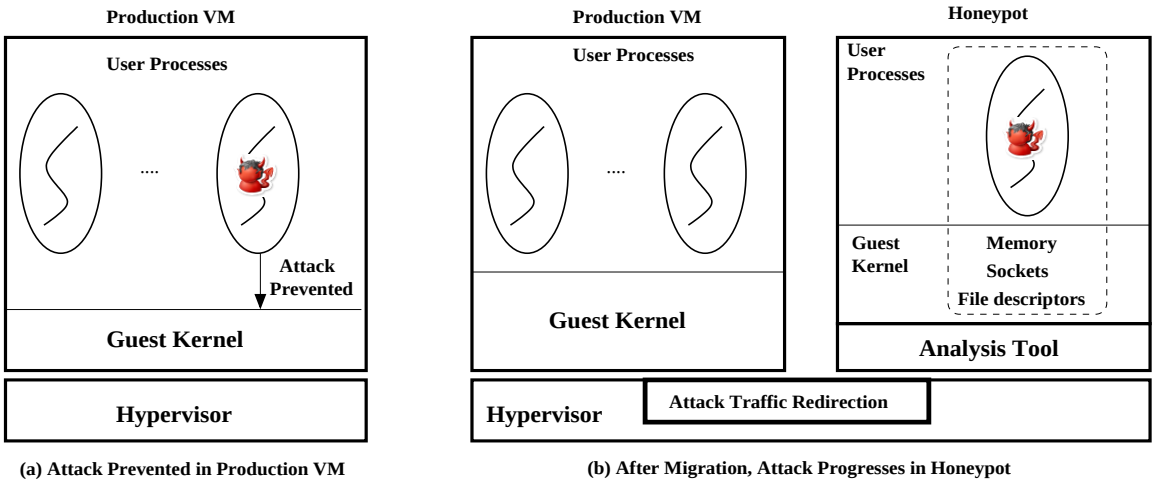

Figure 4.1: An overview of VMsnare

are effective in protecting production systems from attacks, if the intrusion were allowed to proceed (as it would in a honeypot), valuable information about the malware and intrusion can be collected for analysis. Naturally, it is undesirable to allow attacks to continue in production systems and target valuable user data and applications. In our framework, to facilitate detailed intrusion analysis, we investigate techniques to efficiently migrate the attack to an isolated environment. First, we observe that system virtualization has been leveraged for malware prevention in virtual machines running on a trusted hypervisor [42, 51, 75, 84, 86, 91]. As an example, the NICKLE system [75] leverages virtualization to defend systems against kernel rootkits by disallowing any unauthorized code execution. Such virtualization-based malware prevention systems can be naturally extended to trigger detailed analysis.

In this chapter, we present VMsnare which leverages virtualization to facilitate continuous and isolated execution of attacks that target production systems to enable their analysis. Similar to our terminology in Chapter 2, we use the terms "production VM" and "honeypot VM" respectively to represent the VM running a user's actual workload protected by anti-malware software and the VM that will act as a honeypot. The production VM makes full use of underlying hardware virtualization support and runs at optimal performance. The honeypot VM is hosted in an emulation environment that allows maximum flexibility for fine-grained and extensible analysis. Similar to previous honeypot systems [50, 83], VMsnare leverages virtualization for rich analysis of ongoing intrusions. However, in VMsnare, we do not require the hosting and management of independent honeypot VMs that are targeted for attacks and capture of intrusions. Instead, we extract intrusions that have been detected and prevented by the production VM's anti-malware system into a honeypot, thereby enabling live analysis of such intrusions.

A high-level overview of VMsnare is presented in Figure 4.1. We have developed two key techniques to enable VMsnare. While an attack is being prevented from infecting the production VM, our first technique *"Attack Preservation"* ensures that the attack state is preserved on the host system but destroyed in the production VM. This ensures that the production VM itself continues to execute free from infection. At the same time, the attack state is accessible to the underlying hypervisor and our second technique *"Live Analysis"* ensures that the attack state is quickly transferred to a honeypot VM running on the same physical host. Further, we ensure that, though the attack is now transferred to the analysis environment, it continues to execute without any disruption in execution and its network connectivity to external entities is maintained. In VMsnare, our focus is on attacks originating from user-mode processes, which can be either untrusted or compromised trusted processes. The attacking process is prevented from infecting the production VM, but can successfully infect the honeypot VM, while being monitored. The honeypot itself is hosted in a highly-instrumentable environment that affords great flexibility for detailed monitoring and analysis. Because the malicious process continues to execute in this isolated analysis environment, it cannot tamper with the production VM.

We have implemented a proof-of-concept prototype of VMsnare using the open-source KVM hypervisor and Ubuntu Linux 11.04 VMs. We have evaluated our prototype with a number of scenarios including a compromised web server accessing sensitive resources. Further, we have shown that the attacking process can be transferred into a highly-instrumentable emulation environment (QEMU) for detailed analysis. This allows integration with a number of existing tools such as system call monitoring [50], kernel malware analysis [76], and recording and replaying of entire attacks [83]. We also evaluate the performance impact of VMsnare and live intrusion analysis. Our evaluation with a number of standard benchmark programs shows that our prototype implementation incurs a small performance overhead for workloads in the production VM, while allowing the attacking process to execute in an emulation environment.

4.2 Design

4.2.1 Goals and Assumptions

VMsnare is a virtualization-based approach that enables secure observation of detected intrusions targeted at production systems, while still protecting the production systems. It is introduced to effectively enable capture and secure observation of the malicious behavior to further improve malware defense systems. The captured attacking process is then redirected to an instrumented honeypot environment for detailed analysis. In order to develop an effective system for intrusion analysis, we have the following primary design goals:

- *Attack Isolation* VMsnare should strictly isolate the malicious process from other processes running in the production VM as well as the underlying OS kernel in the production VM. Although we are allowing continued execution of suspect malware, the data and code integrity in the production VM must not be compromised.

- *Seamless Operation* Malicious processes typically maintain external network connections. When transferring a malicious process for analysis, all its resources must be kept alive and the process should be able to continue communicating with external entities using network connections that could have been opened in the production VM.

- *Flexible Analysis* In order to allow for meaningful analysis, VMsnare should enable the extracted process to run in a honeypot environment that is highly and flexibly instrumentable. We should enable monitoring the malware process' complete behavior.

We assume that the hypervisor hosting the production and honeypot VMs is trusted and it provides strong isolation between different VMs running on the same physical hardware. We also assume that the hypervisor is protected from any malware infections that target the VMs and that the underlying hardware is trusted. In our current VMsnare design, our focus is on process-level malware. We rely on a malware prevention system which provides system policy enforcement and triggers analysis on policy violation. We further assume that the malware prevention system and the guest OS are not compromised when the malware is detected.

When the malware prevention system is about to prevent an attack, in VMsnare, we identify and preserve the attack state and external resources (the process' network connections). We preserve this state in the physical system, while destroying it in the production VM. The preserved attack state is then transferred to the honeypot environment where the attack continues to progress without any disruption, while being monitored, thereby enabling live analysis.

4.2.2 Attack Preservation

VMsnare Triggers

VMsnare relies on an effective malware prevention system to be in place in the production VM. Live analysis of an attack will be triggered for attacks prevented by the anti-malware system, at the *break-in point*, when the attempted attack is detected and stopped. In VMsnare, rather than stopping the attempted attack from infecting the production VM, we transfer the attack to a honeypot environment for analysis. To ensure clean transfer of the entire attack, the *break-in point* must be such that no sensitive resources are compromised prior to this instance. Since malware prevention systems provide this guarantee, we can naturally extend them to trigger live analysis in VMsnare.

System call policies [38, 74] have been used extensively for attack prevention. For example, system call policies can be used to isolate untrusted processes from trusted resources and the underlying OS kernel. Such policies can also be used for enforcing that trusted or critical processes adhere to pre-defined known-good behavior. We can prevent execution of certain system calls by various processes, according to pre-configured policies. In VMsnare, we leverage system call policies as the malware prevention system and extend the policy enforcement mechanism to trigger live analysis. Specifically, when a process attempts to execute a system call that violates the system policy, this will be flagged as a *break-in point* and VMsnare ensures the attacking process is transferred to a honeypot for continued execution and analysis.

System virtualization has also been leveraged for preventing malware attacks on guest OS'. The presence of a higher-privileged hypervisor layer has been used to effectively prevent kernel rootkit infections as well [78, 91]. For example, [75] leverages virtualization to maintain two separate memory spaces for a VM's OS kernel code. It maintains an authenticated, protected copy of the running kernel code in memory and ensures that all kernel instruction fetches are transparently routed to this authenticated code. Thus, the attacking code will remain in memory, but will be blocked from executing in the VM. In VMsnare, we can naturally extend systems such as NICKLE to trigger live analysis and transfer the attacking process to a honeypot for analysis.

In general, malware prevention systems can be extended to trigger live analysis in VMsnare. At the *break-in point*, when an attack is about to be terminated, we preserve and migrate it to the honeypot for analysis.

State Identification

Once the *break-in point* is detected by the production VM's anti-malware system, our first step is to preserve the attack so that it can be transferred for live analysis. Specifically, we ensure that, at the instant when a malicious process is about to infect a system, the process execution is stopped and the state of the attacking process is captured. The attacking process, similar to any process running in the system, consists of OS-specific resources and the corresponding underlying physical resources in the system. After identifying these resources, we destroy them in the production VM and allow the hypervisor to "take over" this process. At a high level, the resources associated with a process include CPU register contents (at the instant the attack was prevented), the virtual memory addresses, memory content, file descriptors, and network connections. Resources such as virtual memory map to underlying physical resources in the system - the actual physical page frames that are allocated to the process. In order to preserve the attack, we need to identify this state information and remove them from the production VM. This preserved state is then transferred to the honeypot where the attack can continue to

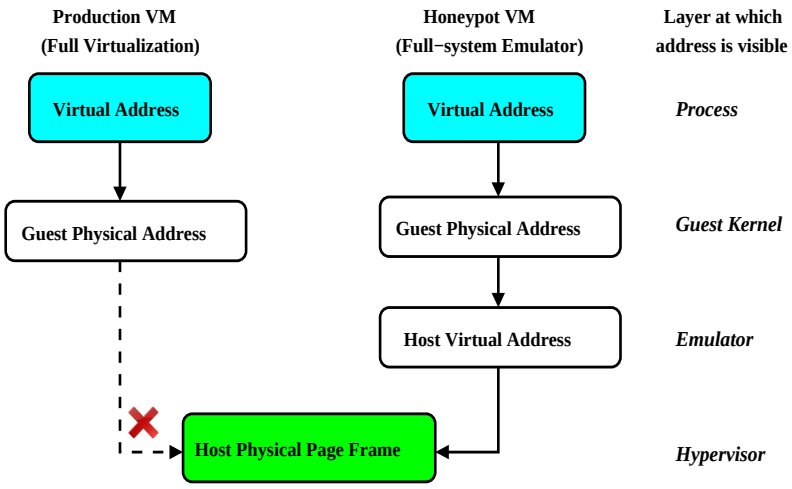

Figure 4.2: In VMsnare, hypervisor re-maps host physical page frame of guest physical address in production VM to host virtual address of honeypot emulator

progress, while being monitored (Section 4.2.3).

The production VM runs with full hardware virtualization support from the hypervisor, available in mainstream commodity CPUs [2, 49]. This allows the production environment to run with optimal performance. In order to identify the attack process' state and resources, one option is to leverage VM introspection techniques [42, 51, 67]. In VM introspection, the hypervisor examines the state of the VM that is available externally and interprets it to make meaningful observations. For example, the hypervisor has access to the VM's entire memory and can examine the guest kernel memory to determine the list of running processes or loaded kernel modules in the VM. However, VM introspection techniques have inherent limitations such as the semantic gap [51] and are sensitive to guest OS kernel changes [67]. Further, OSes maintain detailed state information for resources such as network sockets. If a hypervisor has to externally parse memory and interpret it to extract such level of detailed state, it will be tedious and likely lead to significant performance impact to the production VM. This is because the production VM has to be paused while the hypervisor is examining its memory pages to extract the attack state.

In VMsnare, in order to avoid the above shortcomings of VM introspection, we leverage the semantic knowledge available in the guest kernel to identify the attack state. As described in Section 4.2.2, attack preservation is triggered at the *break-in point* prior to which the production guest kernel and underlying system have not been compromised. (The VMsnare trigger can be raised by any number of sources such as system call monitors or other system integrity monitors. Our key techniques are independent of the actual malware prevention system and the VMsnare trigger can arise from either in-guest monitors or external monitors.) When the

anti-malware system detects a *break-in point*, the production VM guest kernel is notified of this attempted attack. Specifically, when a system call violation is detected for a process, the production VM guest kernel is notified. The kernel stops execution of the process and retrieves its entire state information by walking through the OS-internal data structures and places the state information in a memory buffer. It next invokes a hypercall and sends down this information to the hypervisor. Thus, the hypervisor receives the "internal state" information of the attack.

When the hypervisor gets this information, it proceeds to identify the corresponding external resources. The hypervisor walks through the underlying physical page tables of the production VM and identifies the physical page frame number corresponding to each virtual memory page belonging to the process in the VM. As mentioned previously, the production VM runs with hardware virtualization support including memory virtualization support [48]. The production VM guest kernel identifies the *guest-physical* page frame numbers and the hypervisor retrieves the corresponding *host-physical* page frame numbers. This is illustrated in Figure 4.2. At the same time, the hypervisor clears out the *host-physical* page's page table entry in the production VM's hardware page table. Thus, while preserving the physical attack state, we disconnect the malicious memory state from the production VM. The attack state is preserved for use in the honeypot VM (Section 4.2.3). The hypervisor keeps track of the host frame numbers that contain malicious state and it ensures that these are not mapped to any production VMs, or used for any purposes other than in the honeypot VM.

In VMsnare, after an attack is migrated to the honeypot VM, we ensure that its external network connectivity is maintained. During attack preservation, we identify the external network connections that are being used in the attack. For this, the hypervisor retrieves the network port numbers and remote IP addresses used by the attacking process. This information is provided to the hypervisor by the production system guest kernel. The hypervisor records the port numbers and IP addresses to ensure that no network traffic matching attack patterns is delivered to the production VM (Section 4.2.3). Thus, in VMsnare, we identify and preserve external network connectivity state for analysis, while disconnecting them from the production VM.

When executing in the honeypot, the attacking process may require access to a file that is not present in the honeypot's filesystem. In our current design, while we do not provide redirection for disk I/O by the attacking process, we could extend the honeypot filesystem to generate fake data for those files that are absent. Another option is to leverage VM storage replication techniques to keep the production and honeypot VMs filesystems in sync. We discuss these in detail in Section 5.5. In our current design, we assume that any files needed by the attacking process are present in the honeypot filesystem (either locally, or via a shared disk). At this point, the hypervisor notifies the production VM that it has preserved the attack state.

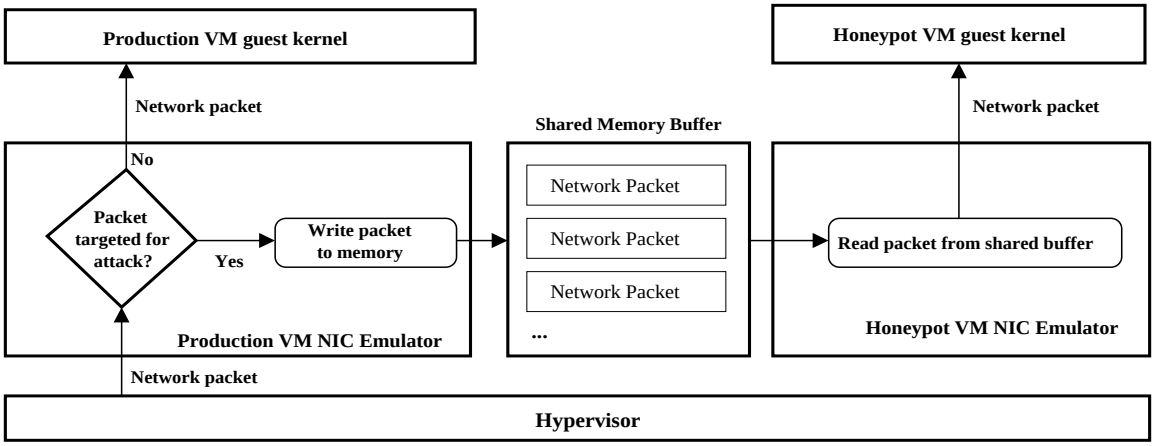

Figure 4.3: Incoming attack traffic redirection in VMsnare

The production VM kernel then destroys in-guest resources belonging to the attacking process.

4.2.3 Live Analysis

Our second technique transfers the preserved attack state to a honeypot environment where it can continue to progress while being monitored closely for analysis. As seen in Section 4.2.2, attack preservation ensures that the malicious state is available to the hypervisor.

In order to provide support for in-depth intrusion analysis that can be easily extended and to allow exernal analysis of the attack, we use a full-software emulator as our honeypot environment. Further, as the attack progresses in the honeypot, it can even infect the honeypot guest kernel, and we must not prevent this in order to enable effective analysis. Since we assume that the entire honeypot VM can become infected, we do not install monitoring software inside the honeypot that would also be subject to tamper and compromise by the attack [50]. We note that the honeypot VM can be always running on the physical system or it can be instantiated from a previously saved VM state at the time live analysis is triggered. The first goal of live attack analysis is to transfer the malicious state from the underlying physical resources into a form that can be used by the emulator.

Full-system software emulators typically run as a process on top of an underlying host OS or hypervisor. Similar to any other user-mode process, the emulator interacts with the underlying hypervisor using system calls. It provides complete hardware emulation implemented in software that can be modified and extended for detailed analysis. For example, internally, the emulator maintains and implements a complete memory management unit for the running VM, in software. When a guest OS running inside the emulator allocates a guest physical page frame, the emulator assigns one of its own host-allocated user-mode memory pages.

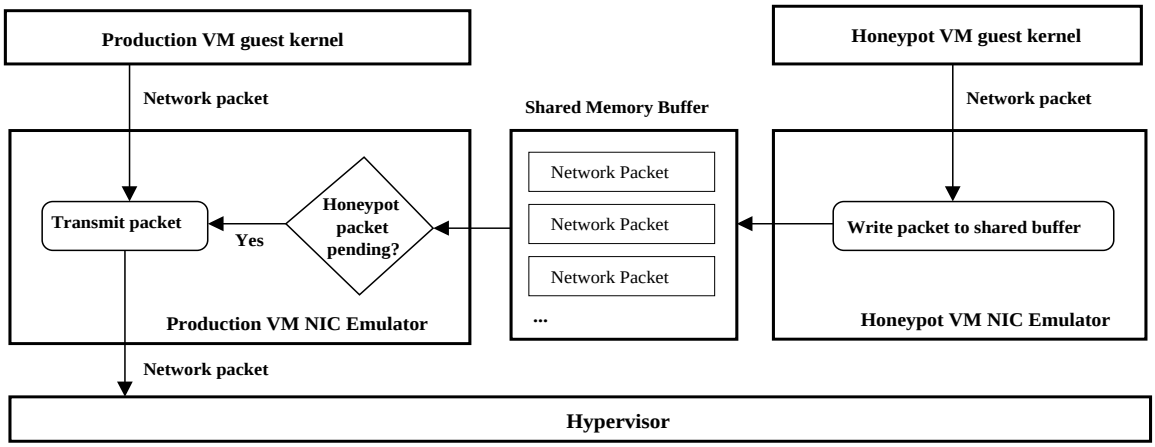

Figure 4.4: Outgoing attack traffic redirection in VMsnare

In VMsnare, after the attack state has been preserved and disconnected from the production VM, the hypervisor works in conjunction with the honeypot emulator to transfer the OS-maintained state information into the honeypot VM. The emulator registers a signal handler with the hypervisor that is invoked whenever the hypervisor has preserved attack state to be transferred. Upon receiving this signal, the emulator retrieves the state information from the hypervisor and invokes a virtual IRQ to the honeypot VM. The honeypot guest OS then creates a process context for the attack process and populates it with the internal state. Specifically, it sets the CPU context, allocates the guest physical page frame numbers for the virtual addresses, opens the required files, and restores network connections' state.

The next step is to transfer the attacking process' memory contents from the underlying physical memory pages (obtained from the production VM) to those allocated for the honeypot emulator. In order to avoid performing a complete memory content copy, we leverage the fact that both the production VM and honeypot VM are executing on the same physical hardware and we re-map the memory pages to the honeypot emulator process. This is illustrated in Figure 4.2. For each guest physical address corresponding to the attack process in the honeypot, the emulator obtains the corresponding host virtual address. The host virtual addresses are allocated to the honeypot by the hypervisor, and the emulator extracts the "guest physical *to* host virtual" mapping from its software MMU. In other words, for every memory page allocated for the attack process in the honeypot VM, we now know the user-mode page address of the honeypot emulator process. The honeypot emulator sends the process' complete memory mapping information to the hypervisor using a hypercall. Next, the hypervisor maps the emulator's host virtual addresses to the corresponding host physical page frame numbers containing the malicious state. Thus, the preserved malicious state information is efficiently transferred into

the honeypot. At this point, the attacking process resumes execution in the honeypot where it can successfully infect the underlying kernel and other honeypot resources.

Attack Traffic Redirection

When a transferred attacking process continues execution in the honeypot VM, it may need to communicate with external network entities. Malware, such as bots, typically communicate with external command and control servers to perform operations on the local system. After migrating the attack to the honeypot, we need to ensure that the process is able to continue using its network resources while retaining the network identity or IP address of the production VM.

In current system virtualization technologies for commodity hardware, hypervisors typically use device emulators to support virtualization of I/O devices. For example, when a VM requires network access, a virtual network interface (NIC) is presented to the guest kernel within the VM by device emulation software. The network device emulator is invoked whenever network packets are sent or received between the VM and external entities. In our design, we configure the honeypot VM such that it does not contain an externally-accessible network interface. Any external traffic is received by the production VM's NIC emulator which redirects attack traffic to the honeypot VM's NIC.

We modify the NIC emulator in both the production VM and honeypot VM to enable the attack traffic redirection, as shown in Figures 4.3 and 4.4. We create a shared memory buffer between the VMs that is used to temporarily store the redirected network packets. As described in Section 4.2.2, the hypervisor captures the network port number and remote IP address used by the attack process. We extend the production VM's NIC emulator to inspect each network packet received from the hypervisor. If the packet header matches the port number and/or remote IP address, then the emulator does not deliver it to the production VM. Instead, it copies the packet into the shared memory buffer. Thus, although the attack is progressing in the physical system, the production VM is still protected from the attacking traffic.

On the honeypot side, we modify the NIC emulator as follows. The virtual NIC does not send or receive network packets using the underlying hypervisor. Instead, it only uses the shared memory buffer for communication with external entities. Specifically, we modify the honeypot's virtual NIC to periodically check the shared memory buffer for any network packets that have been placed there by the production VM's virtual NIC. If such packets are present, we read it and send it to the honeypot guest kernel. Similarly, when the honeypot VM has an output network packet to transmit, the virtual NIC obtains this and copies the packet to the shared memory buffer. Whenever the production VM's virtual NIC sends out a packet from its guest kernel, it also checks this shared memory buffer for any packets placed there by

the honeypot virtual NIC. If such packets are present, they are read by the production virtual NIC and transmitted out using the hypervisor. To ensure that the production VM's network traffic does not suffer significant slowdown, for each production VM packet transmitted, we transmit only one honeypot packet. The honeypot packet transmission rate can be further tailored depending on specific system and workload needs.

Thus, the attack and attacking process continue to execute and progress even communicating with external entities while being monitored. By extending the production VM's NIC for attack traffic redirection, we ensure that the attacking process communicates with external entities using the same network identity as the production VM.

External Monitoring

As we previously described, in VMsnare, we use a full-system software emulator to host the honeypot environment for flexible monitoring and extensible analysis. These monitoring modules are plugged-in external to the honeypot VM by instrumenting the emulator. Previous systems have shown that software emulators can be instrumented for a variety of analysis including effective system call tracing [50, 55], kernel-mode profiling [76], and instruction-level monitoring. In VMsnare, we enable such techniques to be integrated with the honeypot VM emulator and leveraged for external monitoring, while not perturbing the attack process. The external placement of the monitor also protects it from compromise or tampering from within the honeypot VM.

Of particular note is VM record and replay systems [30, 36, 83]. In these systems, the entire execution of a VM is captured and logged. At a later time, the VM's execution can be replayed while running external analysis. VM record and replay techniques can be readily combined with the emulator so that when an attack is migrated to the honeypot, its complete execution can be recorded for later detailed analysis. VMsnare's design readily enables integration and deployment of these virtualization-based analysis systems.

4.3 Implementation

We have implemented a prototype of VMsnare using the open source KVM [5] hypervisor version 2.6.39. KVM is implemented as a kernel module that converts the Linux OS into a hypervisor. KVM enables VMs to run with hardware virtualization support for performance benefits. In our prototype, the host runs 32-bit Ubuntu Linux 11.04 with kernel version 2.6.38. The production VM runs with full virtualization support provided by KVM and the underlying Intel VTX-enabled CPU [48] and device support provided by QEMU. The honeypot VM runs in a full software virtualization environment using QEMU version 0.14.1 [10]. QEMU runs as a

user-mode process on top of Linux/KVM. In this section, we use the term "host OS" whenever we refer to general OS services that are used by QEMU from the underlying Linux kernel, to differentiate from virtualization-specific functions. We have tested with 32-bit Ubuntu Linux 11.04 as the guest OS in both the production and honeypot VMs. The honeypot QEMU is configured so that its virtual NIC is not accessible from the external network.

4.3.1 Attack Preservation

In our prototype, we have used a system call policy enforcer as the malware prevention system. System call policies can be enforced by either an in-guest kernel module or an external component supported by the hypervisor. For ease of prototyping, we choose an in-guest policy enforcer by extending the production VM kernel to monitor system calls executed by specific processes. When a monitored process attempts to execute a system call that violates the policy, it is flagged as a *break-in point* and live analysis is triggered. The guest kernel immediately freezes the violating process and proceeds to capture its state from OS-maintained data structures. Specifically, we have modified the guest Linux kernel to collect CPU register values (`pt_regs` structure), file table (`task_struct->files`) member, and network sockets (`struct socket`). We also note the network port numbers being used. To identify the process' virtual memory addresses and allocated memory pages, we read from the `mm_struct` and `vm_area_struct` structures. The guest kernel places the captured information in a memory buffer and invokes a hypercall to KVM, along with the guest physical address (GPA) of the buffer.

KVM retrieves the GPA and extracts the state information using the `kvm_read_guest()` function. We have modified KVM to then perform a page table walk of production VM's hardware page table, which in our implementation system is provided by Intel EPT [48]). For each guest physical memory frame number mapped to the attacking process' user-mode memory, KVM identifies the corresponding host physical page frame numbers. We use the `gfn_to_pfn()` function in KVM to retrieve the physical page frame numbers. Thus, we build a "guest virtual-to-host physical mapping" of the process' entire user-mode memory. At this point, we have preserved the attack state while disconnecting it from the production VM. KVM returns from the hypercall and the production VM can resume execution. The production guest kernel terminates the attacking process and destroys all the in-guest resources associated with this process.

Next, KVM sends a signal (using the Linux `send_sig()` function) to the honeypot VM process. This signal notifies the honeypot that an attack has been extracted from the production VM and the state is available in KVM. Upon receiving this signal, the honeypot VM proceeds to perform Live Analysis of the attack.

4.3.2 Live Analysis

To host the honeypot environment, we use the open-source QEMU full-system software emulator which offers flexible and extensible analysis capabilities [50, 83]. The QEMU honeypot can be always running or launched from a previously saved snapshot, depending on resource constraints of the underlying system. In our prototype, we set up the honeypot VM to be always running, on a separate processor core. The honeypot runs a guest kernel that is identical in version and configuration to the production VM. Particularly, it is configured with the same IP address as the production VM.

QEMU runs as a user-mode process on the underlying KVM/Linux hypervisor. Although the honeypot runs in full emulation mode and does not use KVM for virtualization, it still needs to communicate with KVM for attack migration. To enable this, we modified QEMU to open the `/dev/kvm` file and used this to issue IOCTLs for communication with the hypervisor. When the honeypot QEMU first starts, it issues an IOCTL to register itself with KVM and KVM records its process id. The honeypot QEMU process also registers a signal handler with the hypervisor.

When KVM receives the attack state from the production VM, it looks up the honeypot process id and sends a signal to it. As soon as the honeypot QEMU process receives the signal, a series of operations occurs. First, QEMU allocates memory and issues an IOCTL to `/dev/kvm` to retrieve the attack state. KVM copies the attack state to this memory location. Next, after QEMU has received the attack state, it has to notify the running guest kernel and transfer the attack state into the VM. This can be done by either injecting an IRQ into the honeypot guest or having the guest poll periodically for attack state availability. In our current prototype implementation, the honeypot guest kernel allocates in-guest memory for the attack state and uses an I/O port to communicate this to QEMU. It periodically monitors this memory to see if QEMU has copied the attack state into it. When KVM receives attack state from the production VM, it notifies the honeypot QEMU process and sends the data to QEMU, which in turn transmits the OS-maintained state to the honeypot VM. The honeypot guest kernel instantiates a process and populates it with the process state retrieved. The corresponding process data structures (as described in Section 4.3.1) are configured with state retrieved from the preserved attack.

Next, in the honeypot, we set up the memory regions for the process and the guest kernel allocates the *guest physical addresses* corresponding to the process' virtual addresses. These memory pages need to be populated with content from the malicious process. As described in Section 4.2.2, rather than copying data from the original physical memory pages, we re-map them into the honeypot. Since we host the honeypot VM in QEMU running completely in software emulation mode, there is no underlying hardware physical page table (i.e. EPT on

Intel platforms). Instead, the "physical memory" memory seen by the honeypot VM comes from the user-mode data pages allocated to QEMU by the host OS. In other words, for every *guest physical* address, there is a corresponding *host virtual address* in QEMU. The *host virtual address* is a user-mode address belonging to the honeypot QEMU process (Figure 4.2).

To re-map the attack process' memory from the production VM to the honeypot VM, the honeypot VM first walks through its in-guest page table (from the virtual `cr3` register) and identifies the *guest physical address* or GPA for each allocated virtual address. The guest kernel then provides this information to QEMU. In turn, QEMU walks through its data structures maintained in the software MMU code and identifies the host virtual address (HVA) for each GPA. Specifically, we use the `phys_page_find()` function in QEMU for this.

After we construct the GPA-HVA mapping for each virtual address belonging to the process, the honeypot QEMU process issues an IOCTL to `/dev/kvm` sending this information. KVM then invokes page mapping kernel functions to map the host physical frame numbers (that were identified during Attack Preservation) to the HVAs corresponding to the attack process in the honeypot. Specifically, in the hypervisor, we use the `pgtbl_walk()` and other kernel functions that modify page table entries, to perform this re-mapping. At this point, the attack state has been recreated in the honeypot, its memory has been re-mapped, and the honeypot can resume execution to allow the attack to progress.

Attack Traffic Redirection

As described in Section 4.2.3, while the attack progress, it will typically need to communicate with an external entity over the network. Since the attack began in the production VM, in order to maintain seamless process execution, we retain the network identity of the production VM, even after attack migration. In VMsnare, we modify the honeypot QEMU such that it does not directly access the external network. All attack traffic targeted to the honeypot are transmitted through the production VM's network device emulator. QEMU offers a variety of networking device emulation options. In our prototype, we choose QEMU's TAP virtual NIC interface [11] and modify it to implement attack traffic redirection. First, we set up a shared memory buffer on the host system between the production and honeypot VMs. The shared memory buffer will be used to place redirected network packets from the attack traffic.

We have modified and extended QEMU's TAP emulation code in both the production VM and honeypot VM. In the production VM's TAP virtual NIC, when a packet is received from the hypervisor, we first examine the packet header to see if it matches the attack traffic pattern. Specifically, we modified the `tap_send()` function to examine the port number and remote IP address in packet headers. If these match the values that were recorded during Attack Preservation, then `tap_send()` does not queue this packet to send into the production

VM. Instead, it copies the entire packet into the shared memory buffer. In the honeypot VM's QEMU, we have modified `tap_send()` so that it no longer accepts packets from the hypervisor. Instead, this function is periodically invoked to check the shared memory buffer. If there is a packet for the attack process, `tap_send()` reads the packet, removes it from the shared memory buffer, and sends it to the honeypot guest kernel.

Similarly, we redirect network traffic sent out from the honeypot VM. The `tap_receive()` function in QEMU is used whenever a network packet has to be sent from the running VM. We have modified the honeypot VM's `tap_receive()` so that it does not transmit the packet using the underlying hypervisor. Instead, the packet is copied into the shared memory buffer. Whenever the production VM QEMU's `tap_receive()` has completed transmitting a packet from the production VM, it checks the shared memory buffer to determine if there is a packet available from the honeypot. If such a packet is available, `tap_receive()` retrieves the packet and transmits it using its own TAP device via the hypervisor. As described in Section 4.2.3, we transmit a maximum of one honeypot VM packet for every output packet transmitted from the production VM and this can be tailored according to specific performance requirements.

Thus, the attack continues to progress while still being able to communicate with external entities. At the same time, the entire attack (including any infections to the kernel) can be completely monitored and analyzed by instrumenting QEMU.

4.4 Evaluation

In this section, we first analyze the security aspects of VMsnare's design. Next, we present results from experiments using our VMsnare prototype, with real-world software and malware. Finally, we present a performance evaluation of our VMsnare prototype using standard benchmarks.

4.4.1 Security Analysis

A key goal in VMsnare is to enable analysis of an attack while still ensuring that the attacking process cannot infect the production VM. In VMsnare, as soon as a process is flagged as malicious (at the *break-in point*), it is immediately frozen and can no longer execute in the production system. We rely on existing malware prevention software (such as anti-virus) to detect the *break-in point*. Since we transfer the attacking process' resources into the honeypot, it can no longer execute in the production VM. Particularly, it cannot tamper with the underlying OS kernel or malware prevention software which continues to protect the production VM as before. We completely destroy the attacking process' resources in the production VM and the attack analysis is done independently in the honeypot VM, ensuring that the attack cannot

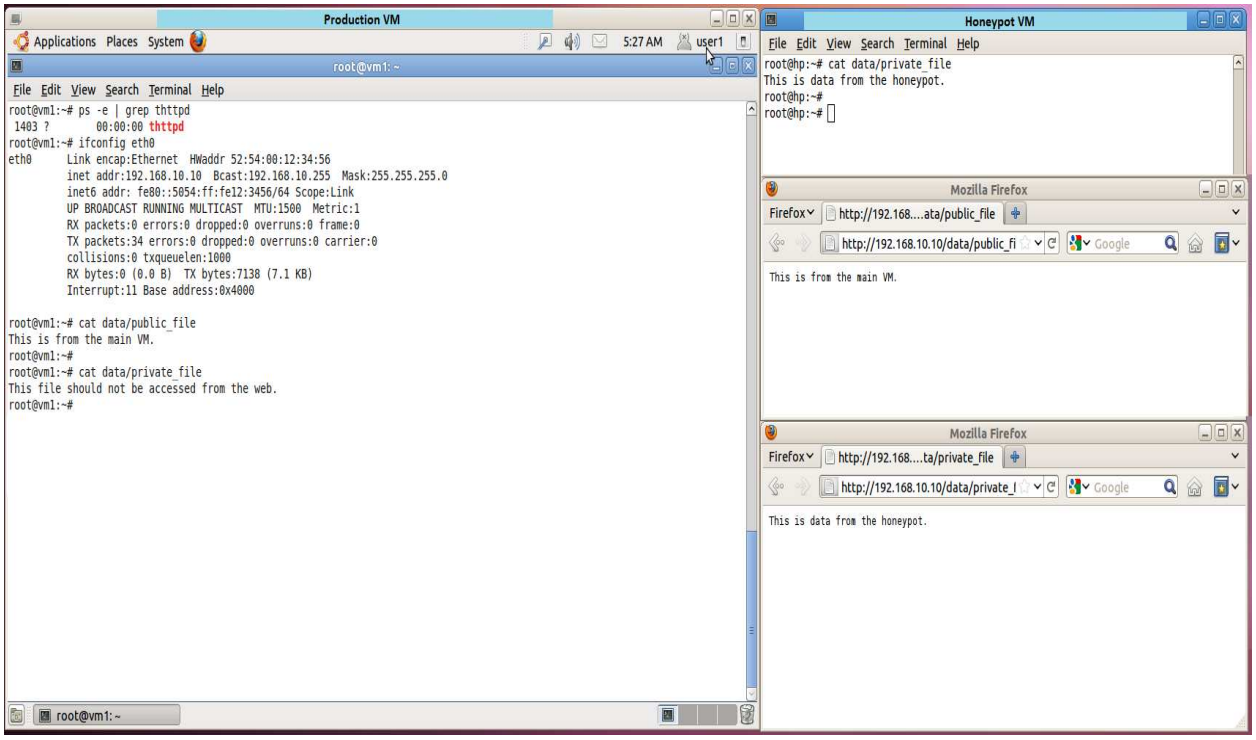

Figure 4.5: VMsnare transfers an attacking process from the production VM to the honeypot VM

tamper with production VM resources. After attack transfer, although the attack state is still present in the system physically, the production VM can no longer access it. While the state (such as malicious memory pages) is present in the underlying physical system, the hypervisor ensures that these are no longer mapped to or accessible by the production VM or any other VMs.

As live analysis progress, while the production VM is involved in network traffic redirection, our design ensures that this is done external to the production VM guest. Specifically, no components inside the production VM - either user-mode or kernel-mode - are used to redirect attack traffic to the honeypot VM. This is done entirely outside the VM by extending the network device emulation layer. Further, the modification that we make to the network emulator is verifiable in that we only check packet headers for criteria matching the attack source. Thus, the integrity of the production VM or its installed anti-malware software is not compromised in VMsnare although attack analysis occurs on the same physical host.

Next, we analyze threats to the honeypot VM as attack analysis progresses. At the point when the attack is transferred to the honeypot VM, the honeypot guest kernel is clean and indeed we leverage it during the transfer for ease of implementation. However, as soon as the

attacking process resumes in the honeypot, it can immediately compromise any other process in the same VM as well as the underlying OS kernel. This is desirable behavior in VMsnare since we enable full system analysis. As an example, if a malware process attempts to infect the honeypot OS kernel to install a kernel rootkit, we allow it to progress so that we can analyze the rootkit behavior as well. However, we ensure that the attack cannot tamper with any monitoring software by deploying them externally. Specifically, the attack is monitored by instrumenting and extending the honeypot emulator. Any logs generated by the monitors are also protected because they are stored externally.

We do not aim to provide transparent attack analysis capabilities in VMsnare. During attack analysis in the honeypot, a malicious process can detect that it is no longer executing in the production environment. Upon such detection, it can modify its behavior and even terminate itself. However, we ensure that even after such detection, it cannot tamper with the production VM or monitoring software since it is restricted to executing within the honeypot.

4.4.2 Effectiveness

User-mode Attack

In our first experiment, we demonstrate VMsnare's effectiveness in capturing a user-mode attack by a process running inside the production VM. For ease of demonstration, we choose *thttpd* to represent the attacking process since we can flexibly interact with it from a web browser running on a different system. Further, since *thttpd* naturally depends on network resources and network traffic, we are able to demonstrate our attack traffic redirection implementation. We first run *thttpd* in the production VM where we have configured a system call policy in the kernel, for files that can be accessed by *thttpd*. Specifically, we restrict the data files that can be opened by *thttpd* to those under a specific directory ("*public*"). Any attempts to open other files are flagged as malicious. While this policy is simple, it is representative of well-known system call policies for web servers (such as those defined for systrace [13]).

Figure 4.5 shows screenshots captured during this experiment. When *thttpd* is running in the production VM, we access a public file's URL from a web browser on the host. As seen in Figure 4.5, we access a data file under the "*public*" directory first from the browser and verify that it shows the contents from the production VM. Next, we point the browser to access a data file under a different directory "*private*". This is in violation of the system call policy configured for the *thttpd* process. When *thttpd* issues a system call to open a file under this directory, a break-in point is detected and the process is immediately flagged as malicious by the production VM kernel. The *thttpd* process' attack state is extracted and preserved by KVM and transferred to the honeypot (see Section 5.2). *thttpd*, now executing in the honeypot VM re-issues the system call to open the private file which completes successfully. A data file from

the "*private*" directory in the honeypot VM is sent to the browser client. We verify that these contents are displayed in the browser.

This experiment shows the effectiveness of VMsnare in maintaining external network connectivity for the attacking process. Although the browser client is still using its previous network connection and the production VM's IP address to access the private file, the data is returned from the honeypot VM. While *thttpd* is executing in the honeypot, we can leverage external monitoring software for detailed analysis. For example, the honeypot's QEMU can be extended with a system call tracer [50] to capture detailed information about the process' attack behavior.

Kernel Rootkit

In our next experiment, we demonstrate VMsnare's effectiveness in analyzing processes that infect the underlying OS kernel to install kernel malware such as rootkits. Kernel rootkits are typically used in conjunction with other malware to achieve stealthy behavior such as hiding malware processes, files, and network connections. Many kernel rootkits are routinely used to evade or even disable the installed anti-malware and monitoring software. To enable analysis of malicious processes that infect the OS kernel and install rootkits, we must be able to securely monitor the execution of the entire honeypot system, including its kernel-mode execution. In VMsnare, we transfer the entire attack to the honeypot VM and the analysis is performed externally by instrumenting the honeypot emulator code. Hence, we can naturally support secure whole-system analysis of kernel rootkits.

For our experiment, we use the SuckIT Linux kernel rootkit [71]. This rootkit infects the underlying kernel by writing to the `/dev/kmem` device which provides access to the OS kernel's memory regions. `/dev/kmem` is now recognized as primarily an attack vector and most Linux distributions do not provide this device in their default compilation options. Thus, when a process attempts to access this device, it will fail since it is absent. We set up a policy such that a break-in point is flagged at the instant when a process issues the `sys_open()` system call specifying `/dev/kmem` as the file name. In SuckIT, this system call is issued by the rootkit installer program and it is immediately transferred the attacking process to the honeypot for analysis. In the honeypot VM, we allow the system call to succeed since we compile the honeypot Linux kernel to include the `/dev/kmem` device. The rootkit installer program successfully infects the honeypot' OS kernel and we are able to exercise the rootkit's functionality.

Next, while the rootkit infection progresses in the honeypot VM, we can monitor its behavior for detailed analysis by instrumenting the honeypot VM's QEMU code. For example, the POKER [76] system effectively builds detailed execution profiles of kernel rootkits. POKER is designed to work in environments that can tolerate high overheads since it requires detailed instruction-level profiling, thereby being unsuitable for production environments. In our exper-

Table 4.1: Software packages used in our evaluation of VMsnare

Name	**Version**	**Configuration**
Host and Guest OS	Ubuntu 11.04	Linux-2.6.38
SPEC CPU 2006	1.0.1	integer suite
Apache	2.2.10	ab -c 3 -t 60
thttpd	2.25b	ab -c 1 -t 60

iments, although the rootkit initially targeted the production VM, we are still able to develop a profile for it. Since we transfer its execution into the honeypot VM, the production VM continues to run unaffected by any overhead introduced due to detailed profiling. Specifically, we can instrument the honeypot VM's QEMU with POKER's *combat-tracking algorithm* which tracks the memory addresses read and written by the rootkit's instructions. This data is then used to build a complete profile of the kernel rootkit's behavior such as determining which kernel objects are targeted by it.

4.4.3 Performance

When evaluating the performance impact due to VMsnare, we are mainly concerned with any slowdown that may be experienced by a workload running inside the production VM, when an attack is transferred to the honeypot for live analysis. Specifically, we are interested in evaluating the impact due to the production VM's network device emulator redirecting traffic to and from the honeypot. We use a Lenovo Thinkpad T420 system containing an Intel Core i7 processor with 2 cores running at 2.7 GHz and 4 GB RAM. The host and guest OSes are 32-bit Ubuntu 11.04 (Linux kernel 2.6.38). Both VMs are configured with 1 VCPU each. The production VM is configured with 1 GB memory and the honeypot VM is configured with 512 MB. Table 4.1 lists the detailed configuration used in our experiments. In all experiments, we pin the two VMs to run on separate physical cores (using the Linux `taskset` command) to avoid interference due to contending for time on the same underlying physical core.

First, we measure the performance impact to the production VM after an attack has been transferred for live analysis to the honeypot VM. To measure this, we first run the standard benchmark, i.e. SPEC CPU 2006 in the production VM while `thttpd` is running in the same VM. We generate traffic for `thttpd` by using the `ab` Apache benchmark tool. `ab` is run from the host system and it is also pinned to a different CPU core. Next, we measure the performance overhead to SPEC CPU 2006 in the production VM when VMsnare has been initiated to transfer `thttpd` to the honeypot. Between the two measurements, we do a cold reboot of the entire host system and restart the VMs to avoid any caching effects. Again, we run `ab` targeting the production VM's IP address, thus exercising the network redirection code and we observe

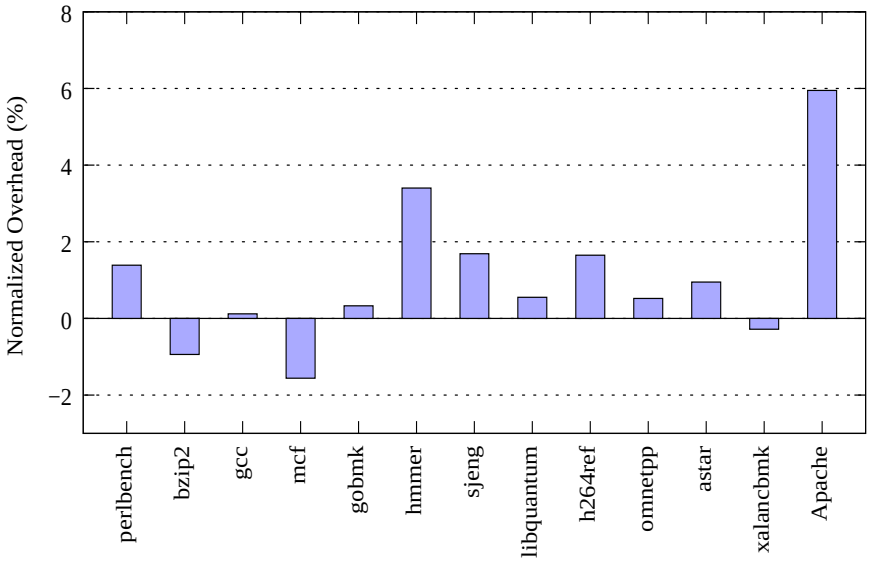

Figure 4.6: Production VM slowdown after VMsnare transfers an attack to the honeypot VM

the impact to the production VM due to this redirection. The normalized overhead is shown in Figure 4.6 and we note a maximum impact of 3.4% (`hmmer`). In some cases, we observe a slight speedup, as a side effect of moving a process from the production VM to a different CPU core.

Similarly, we also measure the impact to a network-bound process running inside the production VM. For this, we run the Apache web server in the production VM and generate traffic using the `ab` benchmark tool. At the same time, VMsnare is initiated and `thttpd` is transferred into the honeypot VM. As shown in Figure 4.6, we see that Apache experiences a slowdown of 5.94%.

Finally, we measure the overhead to the `thttpd` process after it has been transferred to the honeypot VM, when compared to its execution in the production VM. Since the honeypot VM is running in a software emulator without underlying hardware virtualization support, we naturally expect a significant slowdown. In our experiments, we confirm this expectation since we observe a 84% decrease in `thttpd`'s throughput.

4.5 Discussion

Our current prototype implementation of VMsnare has effectively demonstrated attack transfer from production VMs for live analysis. A natural extension of VMsnare would be to leverage VM record and replay techniques [36, 83]. Instead of analyzing the attack as it happens, we can record the entire execution of the honeypot VM. Later, the attack can be replayed for analysis

in multiple phases, while plugging in different kinds of external analysis modules.

In our current design, we leverage the production VM's guest kernel at the break-in point when we transfer the attack state to the honeypot and we only transfer state that is associated with the attacking process. At the break-in point, if the malware has already infected any other state, we cannot reliably transfer the process for effective analysis. For example, an attacking process may have already compromised a critical OS kernel data structure before the break-in point was detected. As an extension to our current VMsnare design, we could investigate techniques for transferring such infected kernel data structures to the honeypot. However, once the production VM's kernel has been compromised, even if we can transfer the attack transfer, it is difficult to guarantee its integrity. Separate recovery operations must be performed to completely restore the production VM.

In our current prototype, after attack transfer, while we implement redirection of network I/O, we do not similarly support redirection of storage I/O. When the attacking process executes in the honeypot, it may attempt to access certain files that are not present in the honeypot filesystem, but available in the production VM's filesystem. For example, these could be additional attack payload downloaded by the malware. Or, the process could perform read-only access to certain environment-specific files. Current VM storage replication techniques can be employed to keep the production VM's non-sensitive data in sync with the honeypot VM's disk. As future work, we propose exploring policy definitions that allows access from the attacking process for specific file locations. This will improve the analysis effectiveness of those attacks that would otherwise fail in our current prototype. To allow filtered access to selected files in the production VM, we plan to leverage system call forwarding techniques [84, 87].

Finally, we do not support interprocess dependencies when transferring the attacking process to the honeypot VM. For example, if there are shared memory regions or IPCs between the attacking process and other processes, we cannot effectively transfer them. Also, our current implementation supports attack transfer only for a single process. Most malware such as bots and rootkit installers operate independently from other processes in the system, and VMsnare can effectively analyze their malicious behavior. However, in order to support interprocess dependencies, our design can be extended to leverage application-level virtualization such as containers [66]. VMsnare's design can also be extended to transfer entire process trees to the honeypot for analysis.

4.6 Related Work

The additional level of indirection provided by a trusted hypervisor has been leveraged in several systems [40, 42, 51, 75, 91] to provide intrusion detection and prevention for the guest VM. Further, virtualization has been used in intrusion analysis by hosting honeypots in VMs

[50, 55, 83, 88] and effectively improving intrusion monitoring capabilities. In VMsnare, we leverage virtualization to provide improved intrusion analysis for live, ongoing intrusions that target production systems.

In Chapter 2, we presented process out-grafting techniques to demonstrate effective out-of-VM monitoring while addressing the semantic gap. Here, we highlight the unique contributions in VMsnare as an integral part of our framework. First, out-grafting only temporarily redirects the user-mode execution of a suspect process. In VMsnare, we completely migrate the attacking process to the honeypot VM and destroy it in the honey VM. Next, for out-of-VM monitoring, it is sufficient to identify only the architectural state of the suspect process. In VMsnare, in order to ensure effective analysis, we are also required identify the attacking process' OS kernel-maintained resources. Further, in VMsnare, we redirect network traffic for the attacking process so that it continues execution in the analysis VM, while retaining the production VM's network identity. Finally, in VMsnare, the honeypot VM runs in a full-system software emulation environment, which introduces subtle complexities in the memory re-mapping design (Section 4.2.3). This is different from leveraging out-grafting for out-of-VM monitoring, where both the production and security VMs run with full virtualization support and hardware-based memory management.

Sophisticated malware defense mechanisms are available for production systems. Mainstream anti-virus products incorporate both signature-based detection and behavioral monitoring. Systems such as NICKLE [75] protect the running guest VM from kernel rootkits. System call policy enforcement has been well-studied [41, 44, 74], for example, to enforce knowngood behavior and isolate untrusted processes. In VMsnare, we rely on these intrusion prevention systems to trigger the transfer of an attack into a honeypot for analysis. Virtualization has been previously leveraged for effective intrusion and malware analysis techniques [50, 83, 76]. In VMsnare, once we transfer an attacking process to the honeypot, such techniques which employ external monitoring extensions can be immediately plugged into the honeypot VM's emulator for analysis.

Finally, the mechanism of identifying the attacking process' state is similar to well-known process migration techniques [9, 61, 65, 66, 81]. However, VMsnare differs from these since we identify both the OS-maintained state and also the corresponding underlying physical state. In VMsnare, we leverage the fact that both the VMs are running in the same physical system and re-map physical resources (memory) from the production VM to the honeypot. Further, we also enable attack traffic redirection between the VMs.

4.7 Summary

We have presented the design, implementation and evaluation of the VMsnare component of our framework, which enables live intrusion analysis of attacks prevented in production systems. In particular, by preserving an attacking process' state from a production VM and transferring it to a highly-instrumentable honeypot environment, we enable fine-grained, extensible analysis. Further, by maintaining its network connections and redirecting its network traffic, we allow it to continue communicating with external entities for attack progression. Our evaluation with a number of performance benchmarks show its effectiveness and practicality. In the following chapter, we present the Timescope component of our framework, which facilitates detailed and time-traveling forensic analysis of honeypots.

Chapter 5

Time-traveling Forensic Analysis with Timescope

In the previous chapter, we presented VMsnare, which effectively enables capturing of live intrusions from production systems for analysis. The next step is to facilitate time-traveling forensic analysis of captured intrusions and derive valuable insight into attackers' techniques and motivations. In this chapter, we present the final component of our framework, Timescope, which leverages previous insights from VM-level deterministic record and replay systems to enable multi-faceted and extensible forensic analysis.

5.1 Background

To capture and analyze computer intrusions and malware infections [64, 82], honeypots have been widely used as an effective tool. For example, by running a commodity system, a high-interaction honeypot is typically designed to host vulnerable services (that can be remotely exploited), and in the meantime also contains additional monitoring software [15] to record intruders' behavior. By allowing intruders to completely take over the system and monitoring their behavior, we can better understand the motivations and techniques behind the intrusion. This is helpful as it will raise the awareness of network situation and lead to better design and development of next-generation intrusion detection systems (IDSs) and anti-malware software.

Forensic analysis of honeypots, though critical for the success of honeypot deployment, is still largely an ad-hoc, time-consuming process and ultimately affected by the type of data collected from honeypots. To better utilize honeypots and facilitate their forensic analysis, we argue that there is a need for a "time-traveling" capability in existing honeypots. By doing so, a security analyst will be given an opportunity to apply a new analysis method that may not be available during the time when the honeypot was deployed. Moreover, by repeatedly "traveling

back in time", multiple phases of analysis can be performed, either in parallel or sequentially. In the sequential case, one replay session can also be based on results from previous replay runs.

To "rewind" a honeypot's execution, an intuitive network-level approach would be to replay the captured network traffic targeting the honeypot system (since the honeypot is remotely compromised). However, due to inherent sources of non-determinism in modern systems and software, by simply replaying the captured network packets, we may *not* be able to obtain the same execution of the honeypot system. From another perspective, a number of system-level deterministic record and replay (R&R) approaches have been proposed for a variety of purposes, including fault tolerance [26], application debugging [1] and security analysis [30, 36]. Recording and replaying a VM is well-suited for honeypots since we can capture and reproduce the entire system's execution. However, most prior VM R&R systems are not suitable for high-interaction honeypots because either they do not support commodity OSes or require extensive OS-level customization, or they heavily rely on proprietary virtual machine monitors (VMMs) [1, 30]. Moreover, there is a lack of honeypot-specific forensic analysis modules that can take advantage of VM R&R capability.

As mentioned in Chapter 1, we have developed Timescope – a time-traveling high-interaction honeypot system designed for extensible, fine-grained forensic analysis. Leveraging previous insights from VM-level R&R systems, we have developed an open-source tool, hoping to engage the security community and benefit related research efforts that may require similar features. We have extended Timescope with a number of analysis modules: *contamination graph generator* (I), *transient evidence recoverer* (II), *shellcode extractor* (II), and *break-in reconstructor* (IV). These modules are applied only during honeypot execution replay sessions and placed externally so that the replay itself is not perturbed. By allowing the analysis modules to "travel back in time", it addresses key questions in honeypot forensic investigations, such as: "what are the contaminations or damages caused by an intrusion?"; "what intermediate evidence (e.g., files and directories), if any, has been erased by the attacker?"; "how is the attack launched?". We have implemented Timescope and these analysis modules based on the open-source QEMU VMM [10] and enabled multi-faceted, inter-related malware forensic analysis during multiple replay sessions. Our evaluation with a number of attack scenarios, including real-world worm programs and kernel rootkits, shows the practicality and effectiveness of Timescope to repeatedly and comprehensively analyze past intrusions. The experiments are enabled by repeatedly rewinding the honeypot's execution, *not* based on the log from one single run. In the remainder of this chapter, we present the high-level design of Timescope and describe the implementation of our prototype. The evaluation with a number of real-world malware shows the practicality and effectiveness of Timescope.

5.2 Design

To better analyze compromised honeypots, our goal is to design an extensible investigation system that is tailored for honeypot forensics. Specifically, the system is intended to greatly facilitate an analyst to effectively reveal various aspects of honeypot intrusions. In the following, we examine two main requirements for our investigation system.

- *Transparency* Our analysis system must work with a commodity OS without requiring any changes to the OS itself. This is needed because a high-interaction honeypot will run environments that are representative of production workloads. Also, due to the potential presence of multiple vulnerabilities in various services running in the honeypot and the need for monitoring attackers' behavior after the break-in, the system should allow for the capture of the execution of the entire honeypot, instead of a selected few applications.

- *Extensibility and Flexibility* The system needs to be extensible to support various analysis modules, each examining a particular aspect of an intrusion. In other words, it can flexibly yield itself for instrumentation during replays to enable in-depth forensic analysis. Further, any analysis module that is supported in this system should not perturb the deterministic execution in a replay session.

Certainly, our design should also meet the basic honeypot requirement in providing a "true" computer system to attackers and reliably recording the honeypot execution for later replay. For example, to maintain the reliability of logging, we need to avoid deploying any visible logging components inside the honeypot as they can be potentially compromised once the honeypot is taken over. And the collected log should not be placed within the honeypot. Also, the presence of the analysis system and various analysis modules should not be exposed to an attacker.

In Timescope, we assume that after compromising a honeypot, the attacker can obtain the highest privilege level inside the honeypot. We envision the scenario that an attacker exploits a vulnerability in a honeypot-hosted network daemon (or a client-side software such as the web browser) and then gains control of the system. After that, the attacker might deploy a rootkit to hide the intrusion. Due to the leverage of the virtualization layer for honeypot hosting, we assume a trusted VMM that provides necessary VM isolation (see Section 5.5).

The overall architecture of our system is shown in Figure 5.1. In essence, it involves the fundamental VM record and replay support. Note that such support can be applied at different levels in a running computer system such as for individual processes or the entire machine. Due to the need for transparently supporting honeypots on commodity hardware, we implement it at the system virtualization layer such that the execution of an entire honeypot VM can be captured and replayed. Among various virtualization techniques available (such as para-virtualization, hardware virtualization etc.), we choose *software-based full virtualization* and

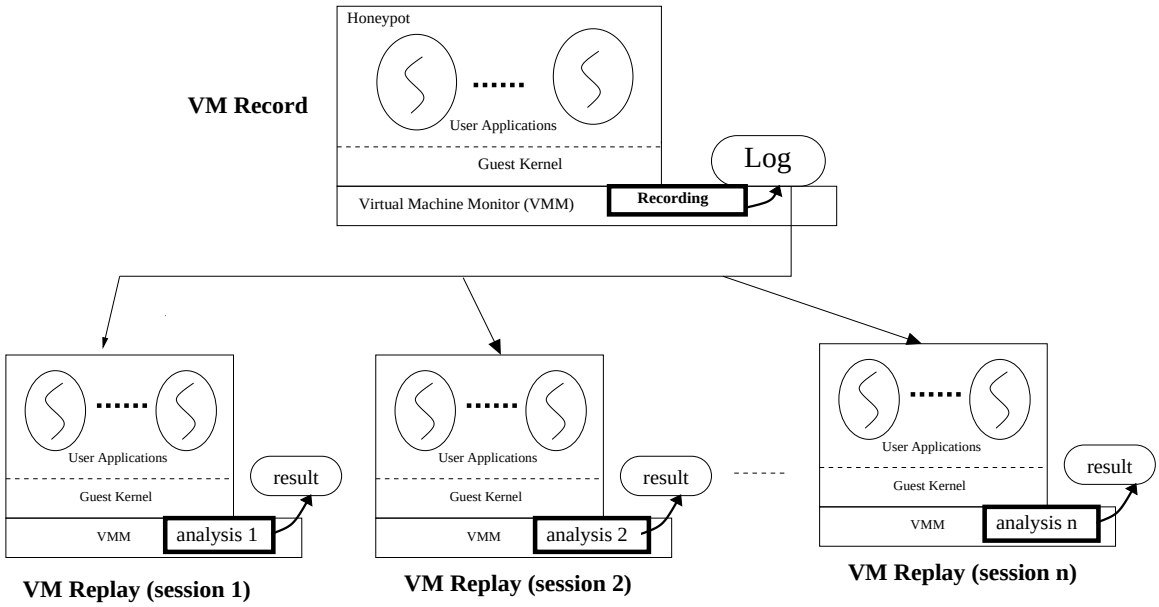

Figure 5.1: Timescope enables time-traveling forensic analysis of honeypots

leverage dynamic binary translation (implemented in VMware [18], VirtualBox[17], and QEMU [10]) which offers great flexibility for implementation of analysis modules. While it introduces high overhead over native system performance, this is acceptable for honeypot purposes.

Timescope operates in two different modes: *VM record* logs the honeypot's execution and periodically takes a number of snapshots (or checkpoints that contain processor, hardware devices and memory states); *VM replay* starts from a chosen snapshot, then re-executes or rolls forward using the collected log to deterministically reproduce the execution. Note that most events in the system are deterministic (e.g. memory loads/stores and arithmetic operations). As such, they do not need to be logged. Instead, the system will just re-execute these events in the same way during replay as it did during VM record.

More specifically, if we abstract the entire guest as a simple VM process, its execution is influenced by the input it receives from external entities (such as I/O devices) and the response (including the run-time environment) from the underlying hypervisor (such as asynchronous I/O, timers, and virtual interrupts). Note that emulating guest VMs as processes will still introduce non-determinism in the VM itself and this should be addressed as shown in Section 5.3. To interact with external entities, it eventually uses the services also provided by the hypervisor (either through certain I/O operations or hypercalls). As a result, during a VM record phase, we can just collect these influence factors or non-deterministic inputs in a log file. During the VM replay phase, we can re-execute the same sequence of instructions with the same input from the collected log and reproduce its execution, including the detailed attack

sequence in the honeypot's past execution. Certainly, when the non-deterministic inputs are collected, we also need to record the related timestamps, which is not based on wall clock time but on the virtual time lapsed since the honeypot started its initial execution. We point out that the output to external devices or peripherals will not affect VM replay and hence need not be saved. In fact, the output can be reconstructed as a by-product during the VM replay. This has the benefit of reducing the log volume – which is especially the case when the honeypot happens to run some I/O-intensive workloads.

5.2.1 Analysis Modules

With the development of companion analysis modules in Timescope, an analyst can travel back in time and investigate an attack when it *is happening*. Analysis sessions can be started from different snapshots to perform specific data collection within different time windows or use results from previous analysis sessions. In the remainder of this section, we describe four representative analysis modules that can be flexibly plugged into the analysis system during a replay session. One example attack scenario is that of a vulnerable service (e.g., the Apache web server) running in a honeypot that is compromised. After the compromise, the attacker escalates his privilege to root and installs a kernel rootkit. We assume the intrusion is observed by an administrator who notices some suspicious activity of the honeypot and denotes this detection point by *DP*. Then, we launch analysis sessions with these modules sequentially, each running in its own replay session.

Contamination graph generator (module I) The goal of this analysis module is to help obtain a high-level view of attackers' behavior by developing a contamination graph. As pointed out in [55], this graph allows us to identify which process was potentially exploited that led to the detection point. For this, the necessary logs can be collected by instrumenting the VMM's dynamic translation layer to intercept and log all system calls made by all processes running inside the honeypot. Note that these system calls are captured in the replay session, not the record session! Along with each system call, we also record the virtual CPU time to identify when the call was intercepted and the address of the instruction that is causing the system call. This analysis module helps address the question: "What time window in a honeypot's execution is interesting for further analysis?" Note the narrow-down of the time window for detailed analysis is helpful to perform targeted forensic analysis. As a result, with the generated contamination graph, we can identify a starting (ST) and ending (EN) points and the execution within [ST, EN] warrants further investigation.

Transient evidence recoverer (module II) Given a time window, this analysis module aims to recover attack evidence that may be erased during the intrusion. For example, during an intrusion, it is likely that the attack may create temporary files (that contain intermediate

computation results) or manipulate some system state for various malicious purposes (e.g., opening a backdoor). As part of the investigation process, it is extremely helpful to uncover all files that may be erased or manipulated and inspect the recovered content to better understand the attacker's motivations.

Shellcode extractor (module III) In certain attack scenarios, there is also a need to identify and extract the injected shellcode in memory. Note the shellcode is typically transient and will not be saved in the disk. Yet, it is the first attack code executed after successfully exploiting the vulnerability in the honeypot. In this analysis module, we aim to keep track of the untrusted network input and identify the set of data that is being executed as code. And this set of data is considered as the shellcode. It is also possible that a DP might be generated due to the shellcode execution, (e.g., an abnormal entry of logging a */bin//sh* process creation from the Apache web server). In our implementation, we leverage existing efforts on dynamic taint analysis [63] and more details will be presented in Section 5.3.2. Further, during a secondary run, this analysis module scans each incoming network message that is read by the exploited process to identify the timestamp when the shellcode is injected as well as the very moment the shellcode is about to execute. This allows us to precisely locate the time window of the code injection attack and aids in further analysis to reconstruct and understand the vulnerability that was exploited.

Break-in reconstructor (module IV) The goal of this module is to perform fine-grained analysis to understand how the execution of malicious, injected code hijacks control flow and tampers with any system resources or objects in process and kernel memory. Specifically, in the case of kernel rootkits, when the injected malicious code executes, this module generates a log of all memory reads and writes along with the memory contents. This collected log can then be analyzed offline, in combination with the binary of the kernel being compromised, to develop a profile of the injected code's execution. With this module, we can "zoom in" to monitor and analyze the execution of the injected attack code and apply in-depth fine-grained analysis techniques. Thus, Timescope re-creates temporary memory states and enables selective application of heavyweight techniques such as execution profiling and improves their efficiency.

Finally, we note that forensic analysis is essentially an iterative process. Based on results from previous phases, it is often the case that one may want to re-run another analysis but with a different time window of the honeypot's execution. Timescope greatly facilitates such analysis with its extensibility.

5.3 Implementation

We have implemented Timescope based on the open-source QEMU version 0.12.3 [10]. As mentioned earlier, due to the lack of a suitable open-source record and replay implementation,

we have to implement it from scratch. On top of that, we further implement four honeypot-specific analysis modules. The dynamic binary translation architecture in QEMU and its readily available source code make it convenient for our implementation. Our development environment is a 32-bit x86 Ubuntu 9.10 running Linux kernel 2.6.31-20.

5.3.1 QEMU Record & Replay

In QEMU, a VM runs based on the emulated computer hardware and I/O devices. Also for each running VM, there is a corresponding user-level process on the host system. At its core, QEMU translates guest instructions in batches (basic blocks) and the resultant host instructions are known as translation blocks (TBs). The translated TBs are then executed from a "main loop". For optimized execution, it employs a technique known as "direct block chaining" [10], where TBs that have been previously translated can be re-used. When a device emulator module in QEMU requires the attention of the CPU, it asynchronously calls a function to signal that an interrupt is pending to be serviced. This causes the main execution loop to exit and service the I/O. QEMU also uses a host timer (by default, the real-time clock) to break periodically from the main loop and perform actions such as refreshing user displays and updating virtual time.

This design brings an interesting observation in our implementation: if we enable R&R for the QEMU process, we can achieve the R&R for the emulated VM. We believe this design choice is different from previous VM-based R&R approaches [30, 36, 33]. However, from another perspective, QEMU itself is a regular but complex user-level program whose design introduces *non-determinism* in the execution of a guest OS. *This violates our requirement for a deterministic R&R!* Specifically, its execution is influenced by external sources of non-determinism such as host OS timer facilities, device interrupts and asynchronous I/O.

To make the QEMU execution deterministic, we need to capture all external inputs. For this, we use a technique known as *function interposition*. We notice that the interface to access these external inputs is the glibc library, which is loaded dynamically and all glibc symbols are resolved at run-time. As a result, we can provide a wrapper for all functions that will be used by QEMU to intercept these dependent glibc calls. During the VM record run, these function calls first invoke the corresponding function in glibc and then record the values of its output parameters and return value. During the VM replay run, the wrapper functions simply return the previously recorded output parameters and the return value from the R&R log.

There is also a subtle issue related to time in QEMU. In particular, QEMU issues the "rdtsc" instruction to read the timestamp counter from the host hardware. For our purpose, we replace the code to this instruction with an equivalent wrapper function. QEMU's default behavior is optimized for performance - the virtual CPU's instructions are executed in a highly optimized loop and exceptions (such as device interrupts) are processed asynchronously. This causes non-

deterministic guest OS behavior. Instead for our implementation, we configure QEMU such that one instruction will be executed in a fixed period of virtual time. Moreover, I/O interrupts are checked and serviced only at the end of a TB's execution. With that, there is no need to rely on the host timer, which is a major source of non-determinism in the original QEMU system.

By addressing the QEMU-inherent non-determinism and logging the external input, our implementation enables deterministic VM record and replay. Also from our implementation experience, there are additional details that are worth mentioning. For example, to support R&R checkpoints, we use the built-in VM snapshot feature in QEMU, but modify it to save and retrieve VM state images to and from a separate file on the host filesystem. Also, our current implementation disables the asynchronous I/O support in QEMU which leads to additional performance penalty (Section 5.4), but makes our implementation easier since it only requires a single thread of execution to be recorded and replayed. Note that this limitation is not inherent in our approach and can be effectively eliminated [22]. Finally, during a replay session, all output from the virtual honeypot to the serial port is allowed to pass through, so that an analyst can "view" the honeypot's execution progress.

5.3.2 Analysis Modules

To demonstrate Timescope's time-traveling analysis capabilities, we have implemented four analysis modules. These modules all operate outside the virtual machine honeypot. Further, they all execute in replay sessions thus enabling time-traveling forensic analysis. The modules we developed examine different aspects of an intrusion, including contamination graph generation, transient evidence recovery, shellcode extraction, and break-in reconstruction.

Contamination graph generator This analysis module typically runs immediately after a suspicious detection point (DP) has been identified. In particular, we replay the VM execution and apply virtual machine introspection techniques [50] to collect all system calls invoked by all processes running inside the honeypot. At a high level, whenever the honeypot executes an *int 0x80/sysenter* instruction, it indicates that a system call is being requested by a process within the honeypot. By examining the honeypot's virtual registers, the system call and corresponding arguments can be identified and reported. This process may further involve examination of the honeypot's memory and interpretation of the name of the running process that invoked the system call and other system call arguments. We point out that the interception and interpretation of guest system calls at the VMM level has been implemented in a few other systems [34, 50]. It is interesting to note that all these techniques operate in a live system. Timescope instead travels back in time and operates in a replayed "live" system.

Transient evidence recoverer As described in Section 5.2.1, given a starting time (ST) and ending time (EN), this analysis module aims to capture all file write activities, copy these files (including the modified content) out, and save them on the host filesystem. By doing so, one can tell the list of files that have been modified or removed by a particular process and all deleted files can still be recovered for later analysis. For this, we first keep track of the open file descriptors within the time window between ST and EN. In particular, our implementation extends the system call interception in the first analysis module: Whenever a *sys_open()* system call is being invoked within the time window, we retrieve the file name and when the corresponding call returns, we obtain the file descriptor. We also track *sys_close()* during this time window to discard file descriptors that are no longer valid. The list of file names and descriptors is maintained on a per-process basis. When a *sys_write()* is intercepted, the size and address of the buffer being written is retrieved from the EDX and ECX registers of the virtual CPU respectively. The entire buffer is then retrieved from the VM physical memory and stored to a corresponding file (referred to as *recovered file*) in a specified directory on the host file system. The name of the recovered file contains the process name that is writing to it and the file descriptor. If this file was opened within the specified time window, we store the name of the file along with the data. Thus, by looking at a recovered file, one can tell the name of the file in the honeypot that is being modified, the corresponding file descriptor and the data that was written. If the file was opened before the ST time window, our current implementation will search through the system call log generated in the first analysis module. Therefore, we are able to selectively create past states of the honeypot's execution.

Shellcode extractor Shellcode is typically the first attack code executed in an intrusion. However, the shellcode itself is not saved in the disk and hence will not be captured by previous analysis modules. In our implementation, we leverage dynamic taint analysis techniques [63, 73] to extract the attack code from memory. Specifically, all incoming network packets (via the virtual NE2000 device in our current implementation) are tagged as tainted. And the taint information will be propagated based on the instructions that operate on the tainted input. Further, by instrumenting the *call*, *jmp*, and *ret*, we can monitor the illegal use of the tainted data. In particular, if any of the targets in call, jmp or ret are tainted, we know the attack code is about to execute. And the execution of tainted data will be collected as the shellcode for analysis. In our implementation, we extract the malicious code by collecting a trace of tainted instructions that are executed by QEMU. Using the addresses of these instructions and the running process (as per the CR3 register), we further record related context information about the shellcode, such as the name of the compromised process. Once the shellcode has been identified, this module can be re-run in a secondary analysis session to identify at which point in the exploited process' execution, this data was injected and understand how the data triggers the vulnerability. For remote code injection attacks, we monitor the returns from the *sys_read()*

calls made by the exploited process and compare the buffer that is read into memory. When a match is found, the corresponding timestamp value and contents of the buffer are stored to a file on the host OS. This needs to be executed only up to the point when the first shellcode instruction is ready to execute.

Break-in reconstructor Once the injected malicious code has been identified and extracted, this module generates an instruction execution trace. The trace will be considered a working exploit against the vulnerability that leads to the honeypot break-in. In our implementation, we further perform execution profiling of the identified malicious code. For example, in our experiments with kernel rootkits (Section 5.4.2), we leverage it and apply the combat tracking technique described in PoKeR [76] to profile rootkit execution within a given time window ([ST, EN]). In particular, for a subset of instructions identified thus, all memory reads and writes and their contents, are recorded in a log on the host OS. Then, with the combat tracking technique, the kernel rootkit's execution profile can be obtained to reveal how kernel objects and control flow have been tampered with. Note that while the experiment is conducted in the context of kernel-level code injection, it can be readily extended to user-level code injection as well.

5.4 Evaluation

This section presents experimental results from our prototype implementation of Timescope. We demonstrate the accuracy of our R&R implementation and time-traveling forensic analysis capabilities. We also measure the performance overhead introduced by Timescope.

5.4.1 R&R Accuracy

To evaluate the accuracy and effectiveness of our prototype R&R implementation, we took two measures. *First*, during a replay session, in each system call wrapper function, our prototype performs a self-check to make sure that the requested system call number and its input parameters always match the next one stored in the R&R log. Our experiments confirmed the correctness of our prototype. Note this self-checking process is costly in terms of performance and thus it is present only in debug builds of the prototype. *Second*, during several tests of VM runs and their corresponding replay sessions, we collect all instructions (organized as basic blocks) executed by the honeypot and save them in two separate log files. By literally performing a file comparison between the two, we verify that the same instructions are executed in the same order, thus yielding deterministic replay.

5.4.2 Time-traveling Analysis

To demonstrate the effectiveness of our prototype, we have launched four synthetic attacks. The first one intentionally tests our second analysis module by verifying the recovery of an intermediate file with randomly generated content. For the rest, we utilized real-world malware, including a worm (Slapper [28]) and two rootkits (adore-ng [58] and SucKIT [71]), to understand their behaviors and test all developed analysis modules. Here, we summarize three of them.

Experiment 1: Intermediate evidence recovery In the first experiment, we show the ability of Timescope to re-create past, non-predictable temporary state and retrieve the content from a replay session for comparison. Specifically, we intentionally create a program that will generate an intermediate file with 1 MB random data. The file will be uploaded to a remote server and then immediately deleted. In the experiment, the run of this program is captured in a VM record session. In a replay session, we aim to uncover the content of the intermediate file using the second analysis module and compare with the copy saved in the remote server. Our manual verification indicates the uncovered file has the same *md5sum* from the server copy.

Experiment 2: Slapper worm analysis In this experiment, we demonstrate the time-traveling analysis capabilities of Timescope for a code injection attack (Slapper worm). Particularly, we setup a Timescope Redhat Linux 8.0 honeypot (in our isolated lab network) running a vulnerable Apache server (version 1.3.22), along with *mod_ssl* support that has a buffer overflow vulnerability exploited by Slapper. From another physical machine, we launch the Slapper worm and direct it to infect the honeypot. On the honeypot, we detect the presence of the worm by monitoring processes running and notice the process ".bugtraq". At this point, we pause the honeypot VM and retrieve the R&R log.

Our analysis is performed using multiple replay sessions using the previously described analysis modules (Section 5.2.1). We start a replay session with the first analysis module and using the results, we apply the backtracking algorithm [55], to generate a contamination graph (Fig. 5.2) of the Slapper infection. In this graph, an oval represents a process; a rectangle represents a file. The numbers in each oval represent the virtual timestamp at which the system call to execute the corresponding process was intercepted. The graph illustrates how the suspect process ".bugtraq" came to exist and shows that the *httpd* (Apache) process was compromised to spawn a shell process. We point out our analysis result is consistent with other Slapper analyses [50, 69].

Next, we start another replay session with the second analysis module. This replay session focuses on a time window specified by two virtual timestamp values (ST - when the "/bin//sh" process is spawned; EN - when the ".bugtraq" process is launched). Our results show that there are 8 files that have been written to (including the entire decoded Slapper source code), and their contents are stored externally as part of analysis results. Using the third analysis

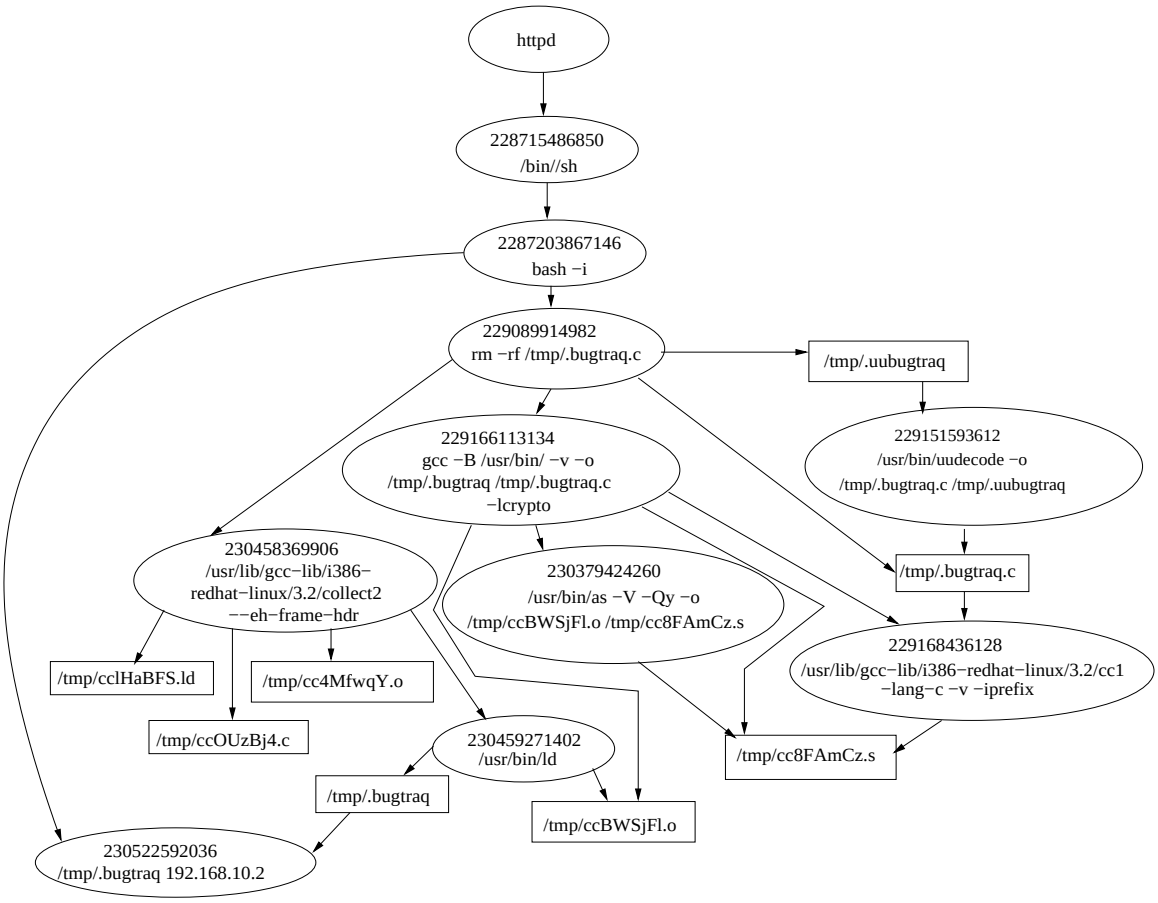

Figure 5.2: The contamination graph of Slapper worm reconstructed from a Timescope-based replay session

module, we extract the injected shellcode in memory that invoked the "/bin//sh" process. For this, we extract the address of the instruction in the *httpd* process that caused the *sys_execve()* to spawn the shell process. We execute a Timescope replay session to collect the instruction trace of the honeypot and search for this instruction address (and the process memory layout identified by the CR3 value). Then, we can identify the basic block of instructions that causes this shell to be spawned.

Using a secondary run of the shellcode extractor, we further identify the timestamp when the malicious code was injected into the process. With that, we identify a new time window for further analysis - from the time this injection occurred in the vulnerable process until the time the shell process was spawned. With the new time window, we execute another Timescope replay session with the second analysis module and we can interestingly identify two files that are modified (including 2 log entries written to Apache's error log). As reported in [69], such

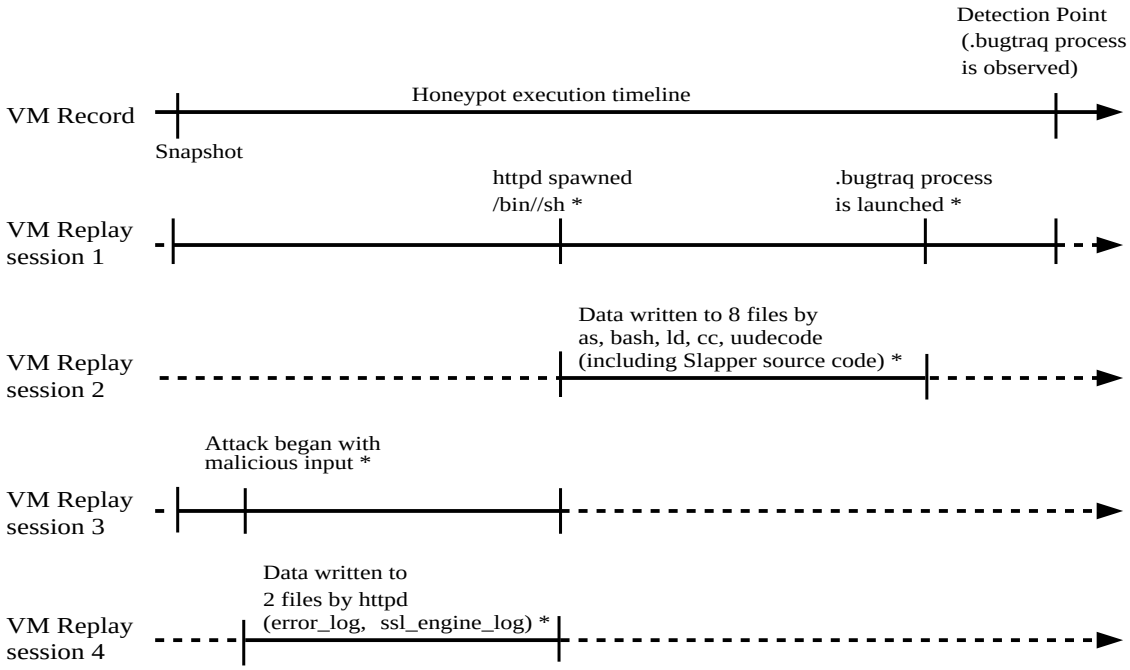

Figure 5.3: Timescope-based multi-phase time-traveling forensic analysis of Slapper infection. The replay sessions are run only for the time window indicated by the solid regions in the execution timeline. Results obtained during a replay session are indicated by asterisks.

behavior is related to the nature of the vulnerability exploited by Slapper. Putting it all together, Fig. 5.3 shows a more complete picture of the Slapper worm infection. Specifically, it depicts various events of interest along a timeline in the honeypot's execution.

Experiment 3: SucKIT rootkit analysis In this experiment, we aim to demonstrate how Timescope's replay-based forensic analysis techniques can be used to analyze intermediate memory states in the honeypot. For this, we use the SucKIT kernel rootkit to attack a honeypot VM. Presuming the scenario of a compromised root password, we launch this attack by logging remotely to the honeypot VM (running in an isolated lab environment), downloading the rootkit and executing a script to install it. To analyze this attack, we run a replay session with the first analysis module and notice the root login and the subsequent commands that were executed (with the *sys_execve()* system call). A subset of the log is shown in Fig. 5.4. In particular, we notice the command "install" run by the attacker and that it opens the file */dev/kmem* which, gives complete write access to the root user to write to arbitrary locations in the kernel memory. To highlight a subset of the execution profiling analysis, consider the lines indicating that the kernel memory is being overwritten as shown in Fig. 5.4. These lines indicate kernel memory being overwritten from the ranges 0xc7024400 to 0xc70261cb. Hence, to perform execution profiling, we use the fourth analysis module and generate a log of memory

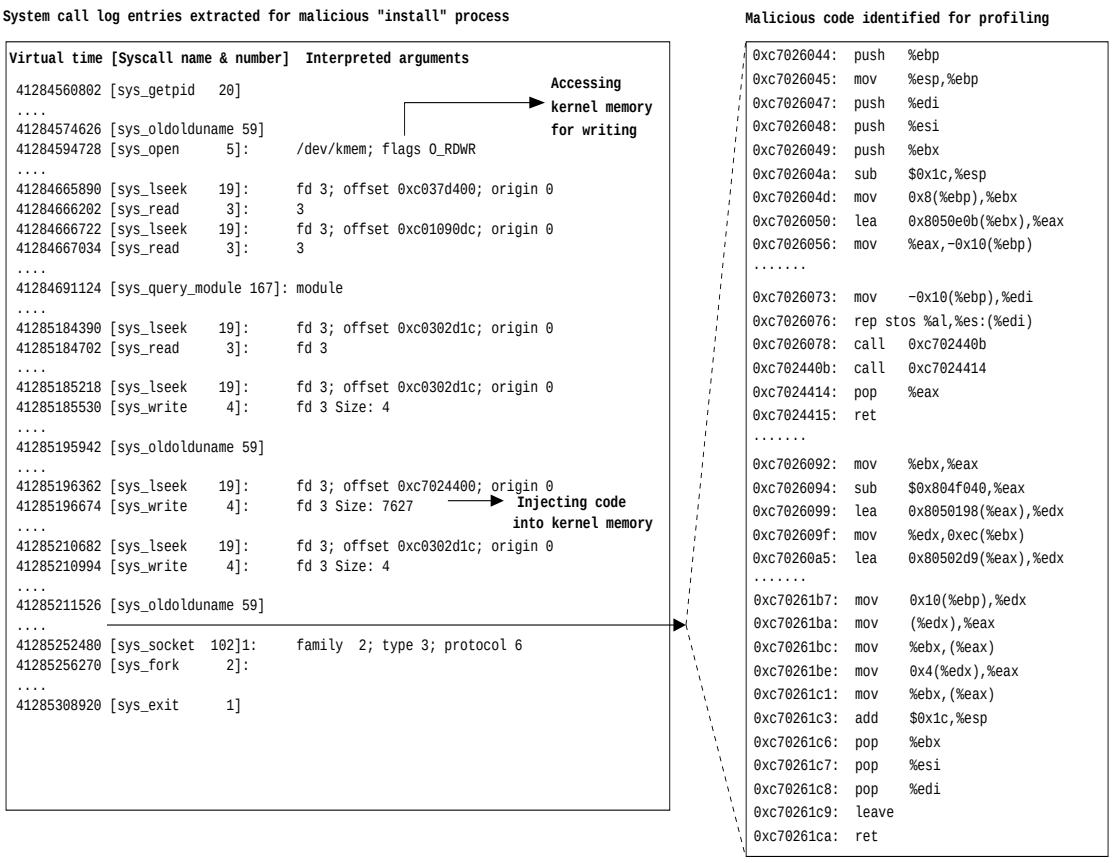

Figure 5.4: SucKIT rootkit analysis using Timescope

reads and writes and their contents for instructions fetched from addresses in these ranges when the processor is in kernel mode. Fig. 5.4 shows a subset of the instruction trace extracted in this range that is analyzed in detail. We can then run the log through PoKeR's combat-tracking algorithm to identify the set of kernel objects being manipulated by the SucKIT rootkit. One interesting observation we would like to note in Fig. 5.4 is that the "install" user process is issuing a *sys_oldolduname()* system call, when in reality, the rootkit overwrote the address of this system call handler in the kernel multiple times to use it for allocating kernel memory, injecting rootkit code in the kernel space, and hijacking kernel control flow. By combining different analysis modules in our system, we are able to understand the purposes of tampering with these data structures in the kernel memory.

5.4.3 Performance

After demonstrating the accuracy and effectiveness of our prototype, we then measure its performance overhead. In particular, as we are less concerned with the overhead during a replay

Table 5.1: Performance overhead in a Timescope VM record session

Benchmark	Configuration	Relative performance with QEMU
nbench	Default	0.97x - 1.39x
gzip	Compress 250 MB file	1.05x
ApacheBench	ab -c3 -t60	1.62x

session, we mainly measure the recording overhead. All the experiments were done with the Timescope honeypot running on a Dell Precision T1500 system with an Intel Core i7 2.8 GHz CPU and 4 GB physical memory. In our measurement, we ran three different benchmarks - Linux nbench [6], ApacheBench [3] and gzip. The configurations of these benchmarks as well as the results are summarized in Table 5.1. Each test was run 10 times and the averages are used to assess the overhead, compared to the default QEMU 0.12.3.

From the table, our evaluation indicates that recording introduces low overhead for the computation-intensive nbench - this is as expected, since most of the execution does not involve external interaction (or involvement of the recording layer). The slowest one in this suite is the "Assignment" test with a relative performance of 1.39x. In a couple of other tests, a minor speedup is noticed, which is due to the variation of different runs. Our evaluation indicates that recording introduces low overhead for the computation-intensive nbench (0.97x - 1.39x of the default QEMU performance). This is as expected, since most of the execution does not involve external interaction (or involvement of the recording layer). For the gzip test, we generated a 250MB file with random data and compressed it, and find that gzip performs at 1.05x of the default QEMU performance. In the case of ApacheBench, it performs at 1.62x of the default QEMU - this is a largely I/O-driven workload, hence the recording software is capturing large amounts of system activity. Though the performance overhead may seem high for normal production systems, we consider it is acceptable for honeypot purposes. From another perspective, the performance overhead is introduced due to certain simplifications we made in the implementation - e.g. disabling asynchronous I/O which could be addressed using other techniques [22].

5.5 Discussion

In this section, we describe current limitations in our prototype and possible solutions to mitigate them. Our honeypot analysis system shares certain limitations with other VM-based intrusion detection systems - the presence of the VMM can be detected by an attacker. However, recent tests have shown that only a small percentage [29] of malware currently perform such checks. Also, with the popularity of virtualized platforms, they may also appear attractive

to existing malware. Moreover, recent work [22, 54] shows promising ways to detect the change in a malware's execution in a virtual environment from a native one and adapt accordingly the underlying VMM layer to handle such difference. In this case, R&R could be leveraged to resume the malware's execution from the point of detection, with the VMM code now adapted to avoid detection and resume the malware analysis. Similar to other VMM-based security research efforts [36, 40, 42, 50, 51], we assume a trustworthy VMM and this is supported with recent progress in improving the hypervisor security [56, 62, 90].

Our analysis modules can also be further extended. For example, it will be helpful to develop extensions with the ability of launching a "go live" session during a replay. That is, instead of executing based on input from the log, the VM resumes real execution from a checkpoint state during a replay. Also, another example will be the development of "what-if" analysis modules that could alter certain input to the VM or its state and determine its effects. This will prove useful for developing and testing defense mechanisms. However, this will require a "live" session of the honeypot and would possibly need network packets to be replayed, depending on the kind of attack.

5.6 Related Work

In traditional host-based high-interaction honeypots, monitoring software (e.g., Sebek [15]) is introduced into the honeypot environment and the logs generated from it are used for forensic analysis. As another example, the Forensix [43] system targets answering various queries related to an intrusion by collecting detailed system information and enabling the resultant log for fast retrieval of queried data. Such systems are limited in re-creating past temporary state (such as memory state) and applying new data collection mechanisms. From another perspective, to address the issue of tamper-resistant forensic analysis while still collecting semantic-rich information, honeypots can be installed as virtual machines and the monitoring software operates at the VMM or hypervisor layer (e.g., by leveraging virtual machine introspection techniques [42, 50]).

Further, the use of virtualization significantly improves deployment and management of honeypots and many honeypot systems have leveraged virtualization to monitor and analyze new attacker techniques [52, 88, 89]. In Timescope, by using VM-based R&R and introducing forensic analysis modules in the VMM layer, one can rewind the honeypot's execution and examine past states of the honeypot in a transparent and non-perturbing manner.

The use of R&R has been proposed previously for a variety of purposes. For example, application cloning [25] aims to capture an application's execution and replay it to its clone on another machine. Aftersight [30] presents the general case for decoupled intrusion detection analysis so that production workloads' performance are not impacted by heavyweight analysis

techniques. Similarly, Crosscut [31], allows replay logs to be "sliced" along time and abstraction boundaries. Both Aftersight and Crosscut implement the record feature based on a proprietary VMM, which significantly limits the capability to customize existing forensic analysis modules or prevents the development of new ones. ReVirt [36] presents a similar VM-based R&R system, but requires a heavily para-virtualized guest OS kernel for the R&R capability. Argos [73], originally developed for capturing zero-day attacks with system-wide taint analysis, has been extended for VM R&R. By leveraging and extending the insights from these R&R systems, we have additionally developed a number of R&R-empowered interdependent investigation modules for honeypot-specific forensic analysis (four of them have been demonstrated in this document).

Meanwhile, it is worth mentioning that our approach to implement R&R is different from most previous ones, i.e., the host-based virtualization approach taken by QEMU will introduce non-determinism in the VM systems. Accordingly, we have to address such non-determinism to enable desirable R&R for honeypot analysis purposes. Also, our analysis modules are tailored for use in multiple-stage forensic analysis of honeypots. The development and deployment of a series of interdependent forensic analysis modules are helpful to construct a comprehensive picture of an intrusion. As a result, our system helps to address key questions in honeypot forensic analysis: "At what point in the execution of a honeypot should we retrieve its state for forensic analysis?" "Should a broader or narrower time window of the honeypot's execution be considered for further analysis?" "What data structures in memory were tampered by the intrusion?"

5.7 Summary

Honeypots are a valuable tool for intrusion and malware infection analysis. In this chapter, we have presented Timescope, a honeypot record and replay system that greatly enhances existing ways to perform forensic analysis of honeypots. Particularly, by allowing (potentially new) analysis methods to "travel back in time", Timescope offers great flexibility in the types of intrusion analysis that can be done. We have developed a QEMU-based prototype and four representative analysis modules. Our evaluation with a number of synthetic honeypot attacks has demonstrated its effectiveness by repeatedly rewinding the honeypot's execution and comprehensively revealing various aspects of honeypot intrusions.

Chapter 6

Conclusion and Future Work

In this dissertation, we have presented and described the *process out-grafting framework* which effectively elevates virtual machine introspection in the areas of semantically-rich monitoring, policy enforcement, intrusion analysis, and forensic analysis. By leveraging a key natural abstraction provided by operating systems, namely the process, our approach has addressed long-standing limitations in prior out-of-VM systems, including the semantic gap problem. Next, we have extended process out-grafting to enable semantically-rich out-of-VM policy enforcement. By developing the VMsnare component of our framework, we have enabled live analysis of ongoing intrusions captured from production environments. Finally, with the Timescope component of our framework, we have demonstrated multi-faceted and extensible forensic analysis of captured malware infections. Our prototype implementations and numerous experiments have demonstrated the practicality and effectiveness of our approach.

Based on the insights gained when developing the process out-grafting framework, we propose three directions for future research.

- **Kernel-mode Security** While process out-grafting effectively addresses several limitations in prior out-of-VM approaches, our current approach does not provide support for kernel-mode monitoring. Kernel-mode monitoring is a critical aspect of system security since it enables protection of the OS kernel and enforcement of process policies defined by the kernel (e.g. process privileges). We propose future work to investigate approaches whereby the guest OS can explicitly leverage the underlying trusted hypervisor to enforce kernel-mode security.

- **Improving Anti-malware Resource Utilization** We have effectively used the process out-grafting techniques for improving various aspects of system security. Fundamentally, we have shown that the user-mode execution of a process can completely occur outside of the VM it belongs to, on an independent processor core. We hypothesize that this property

can be extended and exploited to transfer heavyweight anti-malware functionality (such as behavioral monitoring) onto an independent processor core, thereby improving resource utilization in multi-core systems. We propose future research in this area to study the improvements gained in system performance when leveraging process out-grafting and defining appropriate resource utilization policies. For example, depending on policy, the monitoring of some processes in a VM can be always be transferred to a different core in the system.

- **Cloud Computing** The widespread availability of system virtualization technologies on commodity hardware has led to the popularity of cloud computing, where remote computing resources are made available on-demand. Especially in public cloud environments, strong security and isolation between different customers' workloads is a prime concern. The research presented in this dissertation has immediate applicability to today's cloud computing environments in the area of monitoring and intrusion analysis. We propose extending the process out-grafting approach to address other aspects of cloud computing security such as protecting the execution of security-sensitive applications.

REFERENCES

[1] The Amazing VM Record/Replay Feature in VMware Workstation 6. http://blogs.vmware.com. [last accessed: May 2010].

[2] AMD Virtualization Technology. http://www.amd.com. [last accessed: May 2012].

[3] Apache HTTP Server Benchmarking Tool. http://httpd.apache.org. [last accessed: May 2011].

[4] Kaiten Bot. http://packetstormsecurity.org/irc/kaiten.c. [last accessed: May 2011].

[5] Kernel Virtual Machine. http://www.linux-kvm.org. [last accessed: May 2011].

[6] Linux/Unix nbench Benchmarking Tool. http://www.tux.org/mayer/linux/bmark.html. [last accessed: May 2010].

[7] McAfee Threats Report: Fourth Quarter 2010. http://www.mcafee.com. [last accessed: May 2011].

[8] Microsoft Hyper-V Server 2012. [last accessed: December 2012].

[9] MOSIX - Cluster and Multi-cluster Management. http://www.mosix.cs.huji.ac.il. [last accessed: May 2011].

[10] QEMU. http://www.qemu.org. [last accessed: May 2011].

[11] QEMU Networking. http://wiki.qemu.org/Documentation/Networking. [last accessed: May 2012].

[12] Server Consolidation with VMware ESX Server. http://www.redbooks.ibm.com. [last accessed: May 2012].

[13] Systrace - Interactive Policy Generation for System Calls. http://www.citi.umich.edu/u/provos/systrace. [last accessed: May 2012].

[14] The Honeynet Project. http://www.honeynet.org. [last accessed: May 2010].

[15] The Sebek Project. http://projects.honeynet.org/sebek/. [last accessed: May 2010].

[16] UPX: The Ultimate Packer for eXecutables. http://upx.sourceforge.net. [last accessed: May 2011].

[17] VirtualBox. http://www.virtualbox.org. [last accessed: May 2010].

[18] VMware Inc. `http://www.vmware.com`. [last accessed: May 2012].

[19] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[20] AMD. AMD-V Nested Paging. *AMD White Paper*, 2008.

[21] Ahmed M. Azab, Peng Ning, Emre C. Sezer, and Xiaolan Zhang. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *Proceedings of the 25th Annual Computer Security Applications Conference*, 2009.

[22] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, 2010.

[23] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and The Art of Virtualization. In *ACM SIGOPS Operating Systems Review*, volume 37, 2003.

[24] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of the 15th Annual Conference of the European Institute for Computer Antivirus Research*, 2006.

[25] Philippe Bergheaud, Dinesh Subhraveti, and Marc Vertes. Fault Tolerance in Multiprocessor Systems Via Application Cloning. In *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems*, 2007.

[26] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault Tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[27] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *ACM Transactions on Computer Systems*, volume 15, 1997.

[28] CERT/CC. CERT Advisory CA-2002-27 Apache/mod_ssl Worm. *`http://www.cert.org/advisories/CA-2002-27.html`*.

[29] Xu Chen, Jon Andersen, Z. Morley Mao, Michael D. Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks*, 2008.

[30] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proceedings of the USENIX 2008 Annual Technical Conference*, 2008.

[31] Jim Chow, Dominic Lucchetti, Tal Garfinkel, Geoffrey Lefebvre, Ryan Gardner, Joshua Mason, Sam Small, and Peter M. Chen. Multi-stage Replay with Crosscut. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2010.

[32] Tzi cker Chiueh, Matthew Conover, Maohua Lu, and Bruce Montague. Stealthy Deployment and Execution of In-Guest Kernel Agents. In *Proceedings of the Black Hat Technical Security Conference*, 2009.

[33] Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T. Chong. ExecRecorder: VM-based Full-system Replay for Attack Analysis and System Recovery. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[34] 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.

[35] 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.

[36] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-machine Logging and Replay. *ACM SIGOPS Operating Systems Review*, 36, 2002.

[37] Loras Even. Honey Pot Systems Explained. *http://www.sans.org*, 2000.

[38] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. The Evolution of System-Call Monitoring. In *Proceedings of the 24th Annual Computer Security Applications Conference*, 2008.

[39] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, 2003.

[40] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[41] 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.

[42] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, 2003.

[43] Ashvin Goel, Wu Feng, David Maier, Wu Feng, and Jonathan Walpole. Forensix: A Robust, High-Performance Reconstruction System. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops*, 2005.

[44] 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.

[45] Robert P. Goldberg. Survey of Virtual Machine Research. In *IEEE Computer*, volume 7, 1974.

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

[47] Fanglu Guo, Peter Ferrie, and Tzi-cker Chiueh. A Study of the Packer Problem and Its Solutions. In *Proceedings of the 11th International Symposium on Recent Advances In Intrusion Detection*. 2008.

[48] Intel. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. In *Intel Technology Journal*, volume 10, 2006.

[49] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide, Part 1 and Part 2. 2010.

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

[51] 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.

[52] Xuxian Jiang and Dongyan Xu. Collapsar: A VM-based Architecture for Network Attack Detention Center. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[53] 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.

[54] Min Gyung Kang, Heng Yin, Steve Hanna, Steve McCamant, and Dawn Song. Emulating Emulation-Resistant Malware. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, 2009.

[55] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[56] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles*, 2009.

[57] N. Krawetz. Anti-Honeypot Technology. In *IEEE Security and Privacy*, volume 2, 2004.

[58] LWN. A New Adore Rootkit. *http://lwn.net/Articles/75990*.

[59] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference*, 2007.

[60] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. A Framework for Behavior-Based Malware Analysis in the Cloud. In *Proceedings of the 5th International Conference on Information Systems Security*, 2009.

[61] Dejan S. Milojíčić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process Migration. In *ACM Computing Surveys*, volume 32, 2000.

[62] Derek G. Murray, Grzegorz Milos, and Steven Hand. Improving Xen Security through Disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008.

[63] James Newsome and Dawn Song. Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.

[64] Stephen Northcutt and Judy Novak. *Network Intrusion Detection: An Analyst's Handbook (2nd Edition)*. New Riders Publishing, 2000.

[65] Mark Nuttall. A Brief Survey of Systems Providing Process or Object Migration Facilities. In *ACM SIGOPS Operating Systems Review*, volume 28, 1994.

[66] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *ACM SIGOPS Operating Systems Review*, volume 36, 2002.

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

[68] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, 2008.

[69] Frederic Perriot and Peter Szor. An Analysis of the Slapper Worm Exploit. *www.symantec.com/avcenter/reference/analysis.slapper.worm.pdf*.

[70] Nick L. Petroni, Jr. and Michael Hicks. Automated Detection of Persistent Kernel Control-flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[71] Phrack. Linux On-the-fly Kernel Patching Without LKM. *http://www.phrack.org/issues.html?id=7&issue=58*.

[72] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. In *Communications of the ACM*, volume 17, 1974.

[73] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: An Emulator for Fingerprinting Zero-Day Attacks. In *Proceedings of the 1st ACM European Conference on Computer Systems*, 2006.

[74] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[75] 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.

[76] 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.

[77] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006.

[78] 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 the 21st ACM Symposium on Operating Systems Principles*, 2007.

[79] 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.

[80] Jonathan M. Smith. A Survey of Process Migration Mechanisms. In *ACM SIGOPS Operating Systems Review*, volume 22, 1988.

[81] Jonathan M. Smith. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. *Berkeley Lab Technical Report*, 2002.

[82] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Professional, 2002.

[83] Deepa Srinivasan and Xuxian Jiang. Time-traveling Forensic Analysis of VM-based High-interaction Honeypots. In *Proceedings of the 7th International ICST Conference on Security and Privacy in Communication Networks*, 2011.

[84] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. Process Out-Grafting: An Efficient "Out-of-VM" Approach for Fine-Grained Process Execution Monitoring. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[85] Abhinav Srivastava and Jonathon Giffin. Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In *Proceedings of the 11th International Symposium on Recent Advances In Intrusion Detection*, 2008.

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

[87] Richard Ta-Min, Lionel Litty, and David Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[88] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *ACM SIGOPS Operating Systems Review*, volume 39, 2005.

[89] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Sam King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites

that Exploit Browser Vulnerabilities. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security*, 2006.

[90] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[91] 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.