VIRTUALIZATION-BASED SECURITY ANALYSIS
OF PRODUCTION SERVER SYSTEMS

A Dissertation in
Computer Science and Engineering
by
Shengzhi Zhang

© 2012 Shengzhi Zhang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2012
The dissertation of Shengzhi Zhang was reviewed and approved* by the following:

Peng Liu  
Professor of Information Sciences and Technology  
Dissertation Advisor  
Co-Chair of Committee

Sencun Zhu  
Associate Professor of Computer Science and Engineering  
Co-Chair of Committee

Bhuvan Urgaonkar  
Associate Professor of Computer Science and Engineering

Soundar Kumara  
Professor of Industrial and Manufacturing Engineering

Lee Coraor  
Associate Professor of Computer Science and Engineering  
Head of Computer Science and Engineering

*Signatures are on file in the Graduate School
Production server systems are critical resources in the era of network-centric warfare. With the rapid prevalence of E-Commerce, on-line gaming, social networking, logistics, and Cloud Computing, the demand on service continuity and availability is increasingly crucial to production servers or data centres. Any downtime or malfunction caused by vulnerability exploitation leads to productivity and profit loss. For instance, drivers (accounting for more than half of most commodity operating system kernels), especially third party drivers, could contain malicious code (e.g., logic bombs) and/or carefully designed-in vulnerabilities. Once got executed/exploited, such compromised drivers render the attackers the opportunity of leveraging drivers’ privilege to interrupt the intend-to-guard services. Hence, production server systems need undergo comprehensive security analysis to be sufficiently resistant to attacks to guarantee continuous service and correct execution.

In this dissertation, I propose a set of automatic security analysis mechanisms to help commodity systems to be resilient to vulnerability exploitation. These mechanisms finally help server systems automatically preserve service continuity and correct execution upon vulnerability exploitation. Specifically, an intrusion harm analysis system has been built to comprehensively evaluate the damage caused by attacks to the production server systems. It allows “imperfect” or vulnerable software to be deployed in trustworthy enterprise production environment. By providing automatic checkpointing and intrusion analysis however, the protected systems can obtain precise knowledge
of the damage that has been caused, which would enable the systems to do appropriate availability-integrity tradeoff in generating the recovery plan. As the significant increase of vulnerable drivers, a trustworthiness assessment of third party drivers is also proposed against kernel integrity manipulation, confidentiality tampering, and resource abuse. Then, only the outweighing drivers can be deployed in trustworthy production environment with the proposed operate-through system preserving the states of critical service applications against vulnerability exploitation.

First, I propose PEDA (Production Environment Damage Analysis) system to comprehensively analyze the harm of intrusion to production servers, by decoupling the onerous analysis work from the on-line execution. Once the system being compromised, the “has-been-infected” execution is analyzed during high fidelity replay on a separate instrumentation platform. The replay is implemented based on the heterogeneous virtual machine migration. The servers’ on-line execution runs atop fast hardware-assisted virtual machines (such as Xen for near native speed execution), while the infected execution is replayed atop binary instrumentation virtual machines (such as QEMU for instrumentation platform). From identified intrusion symptoms, PEDA is capable of locating the fine-grained taint seed by integrating the backward system call dependency tracking and one-step-forward taint information flow auditing. Started with the fine-grained taint seed, PEDA applies dynamic taint analysis during the replayed execution to provide the most fine-grained intrusion harm analysis.

Second, I present a novel driver evaluation approach, Heter-device, to fully analyze drivers’ trustworthiness before putting any trust on them. Heter-device relies on virtual platforms to emulate heterogeneous device (Heter-device) pairs (e.g., Intel 82540EM NIC
and Realtek RTL8139) for guest operating system replicas. Each replica loads heterogeneous drivers corresponding to the devices it runs on. Heter-device approach stands on the assumption that heterogeneous drivers should not have the same exploitable vulnerability due to their separate developing procedures. Thus they provide an implicit and complete reference model for each other when trustworthiness assessment is conducted via fine-grained auditing. By deploying Heter-device as a high-interaction honeypot with the synchronization points and monitoring “sensors”, I can closely compare the divergence of two replicas when the vulnerable driver is being compromised. Hence, multiple attack vectors of compromised drivers, including kernel integrity manipulation, resource abuse, and confidentiality tampering can be faithfully revealed.

Last, I present DRASP (Diverse Replica based Application State Preserving), a mechanism that leverages Heter-device architecture to help protected systems operate-through vulnerability exploitation. Once the proposed virtualized device diversity is deployed, the system replicas need to load different drivers as loadable kernel modules for the diverse devices. Again, the idea is that different drivers should have different vulnerabilities, in terms of the location or the details of the vulnerability. Hence, one driver vulnerability exploitation can at most succeed in one replica, with the other replica surviving. Once the successful exploitation tampers the applications’ code/data or compromises the applications’ metadata for commercial benefits, it can be detected through the proposed response or state validation. Afterwards, the application on the survival replica can immediately take over the workload to preserve the service continuity and accumulated state, while ensuring the correct execution.
Table of Contents

List of Tables ................................................................. xi

List of Figures ............................................................... xii

Acknowledgments ............................................................ xiv

Chapter 1. Introduction .................................................... 1
  1.1 Motivation ............................................................... 3
  1.2 Contribution ........................................................... 5
    1.2.1 PEDA .............................................................. 6
    1.2.2 Heter-device .................................................... 7
    1.2.3 DRASP ............................................................ 8
  1.3 Organization .......................................................... 9

Chapter 2. Background and Related Work ............................... 11
  2.1 Background .......................................................... 11
    2.1.1 Dynamic Binary Translation ................................. 12
    2.1.2 Backtracking Intrusion ..................................... 13
    2.1.3 Whole System Replay on Virtual Machine ................. 13
    2.1.4 Taint Analysis ................................................. 14
  2.2 Related Work ......................................................... 15
2.2.1 Memory Bug Finding and Forensics for Production Workload Systems ............................................. 15
2.2.2 Diversity Approach .................................................. 17
2.2.3 Protect Kernel “Belongings” from Driver Faults or Bugs . . 18
2.2.4 Application State Preservation ................................. 20

Chapter 3. Comprehensive Intrusion harm Analysis .......................... 22
3.1 Introduction ................................................................. 22
3.2 Problem Statement and PEDA Approach .......................... 25
  3.2.1 Problem Statement ................................................ 26
  3.2.2 PEDA Approach ................................................. 27
  3.2.3 Lightweight auditing phase ...................................... 28
  3.2.4 Intrusion root identification phase ............................ 29
  3.2.5 Infection diagnosis phase ....................................... 30
3.3 Design of PEDA System ............................................. 31
  3.3.1 Analysis Decoupling .............................................. 31
    3.3.1.1 Checkpointing ........................................... 32
    3.3.1.2 Non-deterministic events logging ..................... 34
  3.3.2 Fine-grained Intrusion Root Identification .................. 35
    3.3.2.1 Dependency Graph Generation ......................... 36
    3.3.2.2 Intrusion Root Identification ......................... 38
  3.3.3 Heterogeneous VM Migration ................................. 42
  3.3.4 Cross Layer Infection Diagnosis ............................. 43
### 3.3.4.1 Infection Analyzer

Page 44

### 3.3.4.2 Reconstruction Engine

Page 46

### 3.4 Implementation Issues of PEDA System

Page 47

#### 3.4.1 Checkpointing and Non-deterministic Event Logging

Page 47

#### 3.4.2 Translation Engine

Page 48

#### 3.4.3 Infection Analyzer and Reconstruction Engine

Page 50

### 3.5 Summary

Page 51

### Chapter 4. The Drivers Trustworthiness Analysis

Page 53

#### 4.1 Introduction

Page 53

#### 4.2 Threat Model

Page 56

#### 4.3 Heter-device Design

Page 59

##### 4.3.1 Heter-device Architecture

Page 59

##### 4.3.2 Heter-device Approach

Page 61

###### 4.3.2.1 Address-alias Correlation

Page 61

###### 4.3.2.2 Runtime Synchronization

Page 62

###### 4.3.2.3 Kernel Integrity Mediation

Page 65

###### 4.3.2.4 Confidentiality and Resource Consumption

Page 69

#### 4.4 Implementation

Page 71

##### 4.4.1 Heter-device Deployment

Page 71

###### 4.4.1.1 Software Diversity Architecture

Page 71

###### 4.4.1.2 Input Replication and Output Verification

Page 72

###### 4.4.1.3 Synchronization of Replicas

Page 73
4.4.2 Address-alias Correlation .............................. 73
4.4.3 Fine-Grained Driver Execution Mediation ............... 74
4.5 Summary .................................................. 75

Chapter 5. Survivability Analysis against Compromised Drivers .......... 77
5.1 Introduction ................................................ 77
5.2 Problem Statement and DRASP Architecture ................. 80
  5.2.1 Threat Model ........................................ 80
  5.2.2 Problem Statement ................................... 81
  5.2.3 DRASP Architecture .................................. 83
5.3 DRASP Design ............................................. 84
  5.3.1 Model and Rationale .................................. 85
  5.3.2 Synchronization of the front stage and back stage Replicas .. 86
  5.3.3 Response Validation for Intrusion Detection ............... 88
  5.3.4 State Validation for Intrusion Detection .................. 91
  5.3.5 Operate-through Intrusion Response and Extendability Dis-
         cussion .................................................. 94
5.4 Implementation ............................................ 96
  5.4.1 Reconstructing OS Semantics .......................... 97
  5.4.2 Synchronization ....................................... 97
  5.4.3 Validation of Responses and States ...................... 98
5.5 Summary .................................................. 99

Chapter 6. Evaluation ........................................ 101
List of Tables

5.1 Metadata Validation for Service-Application-Target Exploit ............. 92

6.1 Kernel APIs called by different functions in e1000 and 8139too. For each
synchronization function, the upper box contains the invoked kernel APIs
defined in System.map file of guest OS, while the lower box includes the
indirected invoked kernel APIs that are called by drivers through the
following procedure. Drivers call some other extern kernel functions (not
defined in System.map file) by including some .h files, and these functions
in turn invoke the indirected kernel APIs identified by us. ............... 111

6.2 Runtime Performance of Different Benchmarks ......................... 113
List of Figures

3.1 PEDA Architecture .................................................. 27
3.2 Lightweight Auditing Phase ........................................... 28
3.3 Intrusion Root Identification Phase ............................... 29
3.4 Infection Diagnosis Phase ........................................... 30
3.5 System Call Auditing Records ..................................... 36
3.6 Object Dependency Graph .......................................... 37
3.7 Object-level Intrusion Root Identification ........................ 38
3.8 Xen Devices Emulation ............................................. 42
3.9 Taint Propagation Flows ............................................ 45
3.10 IOAPIC Emulation by Xen and QEMU ........................... 50

4.1 Heter-device Software-based Diversity Architecture for Driver Assessment 59
4.2 Interaction between OS Kernel and NIC Driver through Interrupt or
    Interrupt Preemption ............................................. 63
4.3 Interaction between OS Kernel and NIC Driver through NIC Driver
    Methods .......................................................... 65

5.1 DRASP Architecture .................................................. 84
5.2 Mode Transition of Service Application ........................... 89
5.3 State Validation Procedure ......................................... 95

6.1 Baseline Throughput ................................................ 102
6.2 With Router Throughput ........................................ 103
6.3 Performance Degradation Caused by Checkpointing ........ 104
6.4 Fine-Grained Intrusion Root Identification ..................... 106
6.5 Whole System Infection Diagnosis .............................. 107
6.6 Baseline Throughput .............................................. 116
6.7 Coarse-grained Throughput ....................................... 117
6.8 Fine-grained Throughput ........................................... 118
Acknowledgments

First, I would like to express my foremost gratitude to my Ph.D supervisor, Prof. Peng Liu. Five years ago, a telephone conversation with Prof. Liu started my path to Penn State. Then with his enthusiastic inspiration, his patient explanation, and his great effort to make things clear and simple, I was always able to overcome any daunting obstacle through my Ph.D journey. He also opened the door to me of the philosophy of research, the attitude to research, and most importantly, the fun of research. I would also like to thank all my dissertation committee members: Prof. Sencun Zhu, Prof. Bhuvan Urgaonkar, and Prof. Soundar Kumara. Their constructive suggestions and insightful comments help to improve the readability and reduce ambiguity of this dissertation. Many thanks to them for always offering me patient help regarding questions and requests.

I want to express my appreciation to all who have helped and taught me immensely in the group. Dr. Lunquan Li welcomed me on my first day to Happy Valley in August 2007. Dr. Xiaoqi Jia and Dr. Yoon-Chan Jhi are instrumental in helping me to swiftly transit from an electrical engineer to a computer scientist. Dr. Kun Bai and Dr. Fengjun Li have been amazing help all the time. Discussion and collaboration with Xi Xiong always remind me when I was young and enthusiastic. Thank you all in the group for offering me the opportunity of learning the value of a good question and the value of a clear presentation. My sincere thanks also goes to the colleagues Dr. Wenjie Wang in Intel, Haishan Wu in IBM research Lab China, Dr. Kirk Schloegel in Microsoft, Dr.
Allalaghatta Pavan, Dr. Arvind Easwaran, Dr. Gabor Madl, and Dr. Varadarajan Srivatsan in Honeywell Platform Systems group. The collaboration with them broadens my vision and enriches my experiences.

Great thanks also to my wonderful parents for their understanding, patience and encouragement when they were most required. Their hard-working and optimism toward life are always the source of my courage towards unknown future. They deserve far more credit than I can ever give them. My final, and most heartfelt acknowledgement must go to my wife Jiezhu, who accompanied me for more than four years at Penn State and pursued her Ph.D there as well. Her love, encouragement, and most importantly, companionship have turned my journey through graduate school into a pleasure. She has my everlasting love.
Chapter 1

Introduction

Upon system being compromised, a dilemma faced by enterprise security engineers is whether to aggressively continue the service for business continuity or to conservatively shut down the servers for loss constrains. It can be even more complicated as whether to resume the service from a clean checkpoint regardless of accumulated system/services state or to pause the execution for a comprehensive clean-up to preserve as much state as possible. In this scenario, the right decision no doubt relies on a comprehensive intrusion harm analysis for the server systems, e.g., locating the intrusion “break-in” and identifying the intrusion “footprint” (infection and cascading effects caused by infection propagation). However, due to diversity of the vulnerabilities, creativity of the attackers and complexity of the server systems, this too familiar task continues to bother the security engineers for years as an onerous and error-prone work.

As the attackers’ goal switches from just-for-fun to commercial benefit, commodity systems suffer from lots of potential exploitation. Guarding the core component, — the service applications, no longer suffices, because attackers are now trying to manipulate the applications through their running environment, — the OS kernel and/or device drivers. Nowadays, device drivers account for more than half of the source code of most commodity operating system kernels, with much more exploitable vulnerabilities than other kernel code [25]. This renders the attackers the opportunity to exploit the driver
vulnerability to leverage the kernel privilege to manipulate service applications as well as its running environment. Hence, analyzing the trustworthiness of third party drivers is critical to build a trustworthy production server system. Meanwhile, it is also highly desired to build a survival production system which is able to operate through attacks or compromise, to maintain service continuity and availability.

I propose a set of automatic security analysis mechanisms to help commodity systems to be resilient to vulnerability exploitation. These mechanisms finally help server systems automatically preserve service continuity and correct execution upon vulnerability exploitation. First, by providing automatic checkpointing and intrusion analysis, the protected systems can obtain precise knowledge of the damage that has been caused, which would enable the systems to do appropriate availability-integrity tradeoff in generating the recovery plan. Second, in order to protect service applications’ running environment, i.e., OS kernel, from compromised drivers, I propose to analyze the trustworthiness of third party drivers before deploying them into production server systems. The drivers are evaluated in multiple attack vectors, including kernel integrity manipulation, confidentiality tampering, resource abuse, etc. Then, the outweighing drivers with less “defect” can be deployed in trustworthy production environment, with the proposed runtime survivability system to achieve operate-through against vulnerability exploitation.
1.1 Motivation

Four factors motivated my research. Firstly, I observed that most, if not all, intrusion detection systems will lag behind the real intrusion occurrence, which indicates that the intrusion harm has already propagated across the server system when IDS alarms the intrusion. Hence, upon system being compromised, enterprise security engineers have no way but to shut down the servers and do a comprehensive clean-up for loss constrains. The “comprehensive clean-up” relies on a comprehensive intrusion harm analysis for the server systems, e.g., locating the intrusion “break-in” and identifying the intrusion “footprint” (infection and cascading effects caused by infection propagation). However, this basic yet essential task continues to bother the security engineers for years as an onerous and error-prone work. The recently proposed dynamic taint analysis can be applied to the servers’ on-line execution to ensure the fine-grained intrusion analysis, but it intuitively causes significant runtime overhead (about 10-40X [87], [73], [70], [68] and [47]). Obviously, running the on-line servers in that manner is not practical because business-critical production workload servers can’t tolerate such overhead. Thus, researchers’ attention is caught by how to do automatic fine-grained intrusion harm analysis for production workload servers with concerns of precision (without losing fidelity) and performance (without slowing them down greatly).

Secondly, most drivers, especially third party drivers, could contain malicious code (e.g., logic bombs) and/or carefully designed-in vulnerabilities. Once got executed/exploited, such compromised drivers render the attackers the opportunity of leveraging drivers’ privilege to manipulate system integrity and data confidentiality. Fully
assessing third party drivers before running them in most commodity server systems is challenging. First, static analysis of such drivers is not always possible due to the unavailability of their source code. Furthermore, carefully designed-in vulnerability or malicious code triggered by some specific logic are extremely difficult to be pinpointed during static analysis. Second, dynamic taint analysis (e.g., [87] and [27]) of driver code is generally infeasible, due to the unknown taint seed during assessment. Moreover, without an accurate reference model, tainting the entire code space of drivers can only reveal drivers’ behaviour, rather than distinguishing legitimate actions from malicious ones. Last, besides promiscuous attacks such as kernel integrity manipulation, some passive attacks, e.g., eavesdropping, launched by compromised drivers are more difficult to be captured.

Thirdly, with the rapid prevalence of E-commerce, MMO and social networking, the demand on service availability and continuity is increasingly crucial to production servers or data centres. Rather than compromising OS kernel as above, attackers begin to leverage driver code vulnerability to manipulate service core components, the applications [8]. In order to preserve the correctness, the only plausible way in such scenario currently is to restart the whole system as well as the running applications, regardless of any continuity or accumulated state. In fact, I find that the difficulty in detecting such attacks is bounded with the unique role device drivers are playing in a computer system. For instance, attackers could exploit the bugs in a NIC driver, and delicately tamper some incoming requests and outgoing responses. From either the service application’s or the system’s point of view, the driver functions well without any suspicious behaviour.
However, the correctness of the service application is maliciously manipulated, and exist-
ing monitoring or IDS in many cases cannot detect such compromise.

Lastly, for service applications dealing with E-commerce, MMO etc., any down-
time (due to integrity/correctness loss and recovery) leads to productivity and profit loss. Hence, fault tolerant systems ([63], [31], [57], and [69]) are widely deployed and software failure recovery systems ([62], [32]) are thoroughly studied. Further, even the service application is compromised, it is also feasible to preserve as much accumulated state and continuity as possible by means of the intrusion recovery systems ([62], [84], [94], [51]). However, stimulated by the significant commercial benefit, attackers begin trying to evade the existing auditing/recovering techniques by manipulating the service applications through the compromised kernel, e.g., driver code vulnerability exploitation as mentioned above. In such scenario, the survivability of production server systems, i.e., how to preserve the service applications’ accumulated state, continuity and correctness is crucial and challenging.

1.2 Contribution

To this end, I built PEDA system to comprehensively analyze intrusion harm to server systems, Heter-device system to fully assess drivers trustworthiness, and DRASP system to operate through exploitation with continuity, correctness and accumulated state preserving. Below are the details.
1.2.1 PEDA

I have built PEDA system [91, 90, 89], which enables the fine-grained intrusion analysis for production workload server systems. PEDA decomposes the complicated intrusion analysis work into lightweight logging, intrusion root identification, and infection diagnosis to provide comprehensive intrusion analysis with lightweight runtime overhead. PEDA decouples the onerous analysis work from on-line to off-line. During the on-line execution, only sufficient information for later replay and system calls for dependency graph construction are logged. Once intrusion occurs, PEDA invokes the fine-grained intrusion root identification to locate the fine-grained intrusion root at the memory cell or disk sector granularity instead of system objects (such as processes) granularity. The logging information is applied to replay the first-run execution with high fidelity, and taint analysis (starting from the fine-grained intrusion root) is applied to analyze the intrusion at the fine-grained granularity.

PEDA integrates a heterogeneous virtual machine migration framework, which enables the decoupled intrusion analysis to run on binary translation based virtual machine and the server production workload to run on much faster hardware-assisted virtual machine. I believe that the heterogeneous VM migration technique not only makes possible the comprehensive infection diagnosis of production workload systems, but also leads a way for heterogeneous virtual machine based OS replication to do better intrusion detection and prevention. To semantically understand the intrusion analysis provided by PEDa, I also implement dynamic reconstruction functionality to help pinpoint the system-object-level taint propagation. In this way, PEDa is able to provide
the cross-layer comprehensive infection diagnosis both at the binary and system object granularity. The implementation of dynamically reconstructing kernel data structures also leads a way for dynamic mapping between kernel address space and kernel data structures.

1.2.2 Heter-device

I have also built Heter-device system prototype [92], a novel driver evaluation approach, to fully assess drivers before putting any trust on them. Heter-device relies on virtual platforms to emulate heterogeneous device (Heter-device) pairs (e.g., Intel 82540EM NIC and Realtek RTL8139) for guest operating system replicas. Each replica loads heterogeneous drivers corresponding to the devices it runs on. The two replicas with heterogeneous drivers are synchronized at the exported function entry points, which are declared by OS kernel and implemented by each driver.

I start a fine-grained auditing of driver’s execution whenever kernel calls the corresponding driver functions. During driver’s execution, every jump, call or return to kernel or other kernel modules’ address spaces are logged for verification. The logs from heterogeneous drivers are parsed and compared to check any suspicious control flow redirection, e.g., one driver jumps to a kernel segment written by itself, while the other does not exhibit such behaviour. Moreover, any modification to key kernel data by drivers is recorded and verified against the heterogeneous drivers to check if it is a legitimate modification or a malicious manipulation.

The proposed approach also deals with passive attacks launched from compromised drivers, e.g., network card driver intercepting incoming/outgoing packets and
redirecting them to remote entities. The network outgoing packets of the two replicas are audited and compared to find mismatch. Additional amount of traffic on one replica against the other suffices an alarm of confidentiality compromise. Finally, abuse of kernel APIs, such as spin lock or kernel memory allocation requests, may cause CPU or memory starvation. Hence, any call to these resource request APIs from drivers is also verified against heterogeneous drivers.

1.2.3 DRASP

I leveraged Heter-device architecture to build an intrusion detection/recovery system, DRASP [93], for service applications against compromised drivers. By means of the corresponding driver diversity, I can dynamically validate the responses, critical memory regions, and persistent data of the service applications to swiftly detect malicious tampering of applications’ execution via driver vulnerability exploitation. My goal is to detect any malicious attack that exploits driver vulnerabilities, and in turn manipulates applications’ execution for commercial benefit. Hence, the mechanism is to audit the service-orientated commercial applications in terms of their responses, critical code/data on memory and persistent data.

I classify the harm to the service applications by the malicious exploitation as either transitional or persistent. By transitional, I mean that by manipulating the service application, the request processing routine or the state information the routine relies on are tampered. As a result, the response to the request diverges and the intrusion revenue is immediately observed. For instance, by manipulating the price tag, the attackers purchase things for free. During the transitional harm procedure, the responses generated by
the victim service application must have been tampered by attackers. Hence, I synchro-
nize the replicas at the granularity of per packet processing, and validate the responses
of the two service applications at the front tier proxy to detect such compromise.

By *persistent*, I denote a long-term-benefit-pursuing intrusion that achieves its
goal by tampering state information of the service applications. For example, by manip-
ulating the accounting records, the attacker’s property could be augmented significantly.
However, the service responses generated afterwards might not be tampered, since they
may not rely on the maliciously manipulated (application) state. To defeat the *persis-
tent* attack, I validate some critical memory regions (e.g., metadata, crucial code/data
segment) and persistent data of the applications (opened files by the service application
process) on the two replicas in an off-line comparison style. Once any validation policy
is violated, DRASP generates alarm reports for system administrator, who can decide
to transfer the service workload from the compromised replica to the survival replica.

1.3 Organization

In this dissertation, I describe the architecture and implementation of PEDA,
Heter-device, and DRASP, and evaluate their effectiveness and performance cost. Chapter 2 provides background on dynamic binary translation, backtracking intrusions, whole
system replay, and taint analysis. I also discuss related work on memory bug finding and
forensics for production workload systems, diversity approach, driver bug/fault protec-
tion, and application state preserving.

Chapter 3 describes PEDA, an intrusion analysis mechanism for production work-
load server systems. PEDA integrates analysis decoupling and heterogeneous virtual
machine migration to ensure the production system execution runs with lightweight overhead. Furthermore, PEDA involves fine-grained intrusion root identification and dynamic taint analysis to comprehensively and systematically analyze the intrusion harm propagation across the server systems.

Chapter 4 introduces Heter-device architecture, a novel cross-brand comparison approach to analyze the drivers in a honeypot or testing environment. Through hardware virtualization, I design and deploy diverse-drivers based replicas to compare the runtime behaviour of the drivers developed by different vendors.

Chapter 5 presents DRASP system prototype, which detects the driver code bug exploitation targeting service applications based on Heter-device architecture. Once intrusion being detected, DRASP preserves service continuity and correctness by transferring the workload to the back stage survival replica smoothly and transparently.

Following the chapters on PEDA, Heter-device, and DRASP, Chapter 6 analyzes the effectiveness of them at (1) profiling heterogeneous drivers in terms of interaction with OS kernel, (2) swiftly detecting driver code exploit to manipulate service applications and (3) comprehensively analyzing intrusion harm of production workload systems. I also evaluate the performance cost of using PEDA, Heter-device, and DRASP with common service applications. In Chapter 7, I discuss the lessons learned in implementing PEDA, Heter-device, as well as DRASP, and finally conclude.
Chapter 2

Background and Related Work

The goal of my work is to improve the commodity operating systems’ resilience to attacks, that is, a set of security analysis mechanisms that comprehensively analyze the intrusion harm, fully assess drivers’ trustworthiness, and swiftly operate-through the intrusion. These mechanisms are critical to build self-recovery commodity operating systems, which has become a goal of computer system designers recently, especially nowadays the commodity operating systems are confirmed to contain much more vulnerability than ever \cite{25} and \cite{8}. In this chapter, I first present up-to-date schemes that my proposed mechanisms involve. Then I discuss previous approaches on memory bug finding and forensics for production workload systems, diversity approach, protecting OS kernel from driver code vulnerability exploitation, and application state preserving recovery.

2.1 Background

First, I present dynamic binary translation technique, followed by discussing several up-to-date works, e.g., backtracking intrusions, whole system replay and taint analysis, which are widely used in analyzing the intrusions.
2.1.1 Dynamic Binary Translation

As the name indicates, binary translation means the translation of sequences of instructions from the source to the target instruction set. In some cases however, the target instruction set and the source instruction set may be the same to provide testing and debugging features such as instruction trace. Intuitively, static binary translation was first considered to convert all the source binary code into the code that can run on the target architecture before running. However, some parts of the executable may be reachable only through indirect branches, whose value is known only at runtime [2]. By contrast, dynamic binary translation typically looks for a short sequence of source code instructions and translates them into the target ones during runtime.

QEMU [19] is a dynamic binary translation based virtual machine, which generates host code from a piece of guest code without control flow redirection or static CPU state modification. Specifically, when QEMU first encounters a piece of guest code, it translates the code to host code up to the next jump or instruction modifying the static CPU state in a way that cannot be deduced at translation time. Thus, for each sequence of instructions (called translation block), guest OS executes without QEMU intervention unless interrupt occurs. At the end of each translation block, QEMU takes over the control and prepares for the next translation block. In order to improve performance, QEMU uses translation cache to hold the most recently used translation blocks. Furthermore, QEMU also implements translation block chaining for performance boost. In particular, every time when a translation block returns, QEMU tries to chain it to previous block if the value of simulated program counter is determined, e.g., after an
indirect jump. Hence, QEMU can chain those translation blocks together to save the overhead of context switch to QEMU emulation manager next time.

2.1.2 Backtracking Intrusion

BackTracker [52] traces the intrusion backward to automatically identify potential sequences of steps that occurred in an intrusion. Starting from a detection point, such as a suspicious file or process, BackTracker identifies the events and objects that could have affected that detection point and displays chains of events in a dependency graph. Administrators can then focus their detective work on those chains of events, leading to a quicker and easier identification of the vulnerability. In order to identify these chains of events, BackTracker logs the system calls that induce most direct dependencies between operating system objects (e.g., creating a process, reading and writing files), and constructs system object dependency graph dynamically. Once any intrusion or suspicious symptoms are detected, BackTracker first locates the symptom objects on the dependency graph, and then traces the chain backward to identify sequences of events that cause the detected symptoms. BackTracker uses several types of rules to filter out parts of the dependency graph that are unlikely to be related to the intrusion.

2.1.3 Whole System Replay on Virtual Machine

Virtual Machine replay is a relatively mature technique in the VM industry (e.g., VMware) these days. Revirt [37] is the first system that logs enough information to replay a long-term execution of the virtual machine instruction-by-instruction. Before starting the system, Revirt checkpoints the state by making a copy of its virtual disk,
which serves as the starting point for replay. During normal execution of the system, non-deterministic events, e.g. external inputs into the server such as network packets, keyboard inputs, timer interrupt etc., need to be recorded for redelivery during replay. The deterministic execution of the server systems (from the same initial virtual disk) and the non-deterministic events redelivery can ensure the high “fidelity” of the replay. However, the replay on another different VM raises some new issues, such as how to address the device emulation incompatibilities. Aftersight [26] is the first work that I can find talking about these heterogeneous VM migration issues. I will discuss the differences between PEDA and Aftersight in the related work.

2.1.4 Taint Analysis

Several works exist to help system administrators to do intrusion analysis/recovery, such as Repairable File Service [96], Backtracking [52], Rx [62], RETRO [51], and Taser [45]. All of them log system calls during execution, and use them to track the flow/dependency between system objects. Such kind of coarse-grained dependency tracking typically cannot capture the whole “footprint” of intrusion, because the attackers can craft programs with direct memory load/store instructions to evade the system call level auditing. Hence, dynamic taint analysis [87, 61, 28] has been well studied recently auditing each instruction to apply data flow dependency.

For each computational instruction involving a source operand and a destination operand, the destination operand will be tainted if the data derived from the source operand is tainted. Note the data could be stored in register, or fetched from memory. Furthermore, if the value used to compute the source operand address is tainted,
the destination operand should also be tainted. Last, the instruction fetched from a tainted memory address results in the taint of the destination operand as well. Only applying data flow taint policies will render attackers crafting a simple program to evade data flow dependency tracking. Consider the example if(taint == a) clean = a; else clean = a;. The taint will propagate to clean without the data flow dependency between taint and clean.

Therefore, control flow taint policies are also enforced to capture any taint propagation caused by control flows. Specifically, if the condition of the currently executing instruction is tainted or the destination address of the execution redirection is tainted, the following redirected execution should be tainted until the conditional branches’ merging point is identified by applying static analysis [38]. The tainted registers or memory locations will become untainted whenever they are overwritten by untainted values or constants. Note the special cases of assignment instruction such as follows to set register eax to zero: xor %eax, %eax;.

2.2 Related Work

Below, I will discuss the previous approaches closely related to PEDA, Heter-device, and DRASP systems in depth.

2.2.1 Memory Bug Finding and Forensics for Production Workload Systems

Aftersight [26] is the first work that I can find talking about analyzing execution of production workload systems. Being a generic technology for decoupling dynamic program analysis from execution, Aftersight decouples instruction level analysis from
the normal execution (of on-line servers) for a spectrum of purposes, e.g., bug finding and forensics. Aftersight records program execution and replays it on a separate analysis platform against a set of memory safety guarantee policies. Hence, it enables heavyweight analysis during replay to find serious bugs in large complex systems such as VMware ESX Server and Linux.

In contrast, PEDA focuses on the post-mortem intrusion analysis for production workload servers from intrusion root identification to fine-grained infection diagnosis. Thus, the problems faced by PEDA are to precisely locate the intrusion root objects to patch the vulnerabilities, and to reasonably associate the intrusion root objects with the fine-grained taint seed to start comprehensive infection diagnosis. Though Aftersight and PEDA share the same idea of decoupling analysis from normal execution, they aim at different types of analysis, thus dealing with different sets of design and implementation issues.

In addition to dealing with different analysis, PEDA also differs from Aftersight in the architecture design. Aftersight migrates guest server system from recording platform (VMware Workstation) to analysis platform (QEMU), while PEDA does it from Xen to QEMU. Since Xen relies on qemu-dm to emulate majority of devices, PEDA takes the approach of recording the external inputs to each device and redelivering them to the corresponding device during the replay. Because VMware and QEMU emulate I/O devices differently, Aftersight chooses to record all the outputs from each emulated device to CPU and to redeliver them to CPU during the replay to “bypass” the device emulation incompatibility. In order to avoid the significant runtime overhead introduced by the large amount of the device output logging, Aftersight adopts the approach of “replay
based replay”. In particular, Aftersight records external inputs to the device during normal execution, logs the device outputs to CPU during the first replay on the same recording platform, and finally replays the second recording for analysis on the analysis platform. Compared with Aftersight, the device emulation incompatibility elimination of PEDA is more straightforward and efficient for production workload server systems, though less generic.

Several other works exist to help system administrators to do intrusion analysis such as Repairable File Service [96], Intrusion Recovery [94], Backtracking [52], and Process Coloring [49]. All of [96], [94] and Backtracking log system calls during execution, and use them to track the flow/dependency between system objects such as files and processes. In [49], a novel process coloring technique is presented to trace worm break-in and contaminations. By intercepting system calls inside the VM, [49] can successfully prevent advanced attacks from tampering with logging facilities. With a little modification of kernel data structures, [49] records the color information of each process directly in its own process control block, which significantly reduces the size of system call log files.

2.2.2 Diversity Approach.

Software diversity approach for intrusion detection has been studied in several works, such as COTS [78], Behavioural Distance [43], Diversified Process Replica [21], and Detection of Split Personalities [18]. COTS Diversity [78] applies N-version programming into web servers to verify their interactions with the environment for any anomaly, e.g., HTTP responses from those web servers. Generally, [78] is ideal to detect
anomaly targeting web servers, especially for most promiscuous attacks. But for other attacks, such as denial of service attack, or resource abuse, comparing network packets cannot work, since such attacks do not involve network packet revision. On the other hand, comparing network packets is not always possible. For instance, if the payload is encrypted through IPsec, comparing the payload is meaningless.

Behavioural Distance [43] and Detection of Split Personalities [18] aim to detect intrusion or anomaly by comparing the system call sequences made by diverse applications. Diversified Process Replica [21] proposes non-overlapping processes address spaces to defeat memory error exploits. The seminal work N-variant [29] proposes address space partition and instruction set tag diversities to detect divergence caused by intrusions. Although such approaches are quite effective in detecting code injection related attacks, other types of exploitation, such as direct kernel object manipulation, kernel APIs abuse and confidentiality tampering can evade their auditing.

2.2.3 Protect Kernel “Belongings” from Driver Faults or Bugs

Generally, there are several fundamental ways to protect kernel from vulnerable driver code, e.g., isolation based protection, user mode driver based protection, and software based protection. I will discuss them in detail as follows.

**Isolation Based Protection.** Researchers proposed to isolate device driver module from other kernel address spaces, e.g., Nooks [75], Byte-granularity Isolation [23], Reference Validation Mechanism [82], and Device Driver Reuse and Isolation [55]. Nooks ([75], [74]) and Mondrix [83] pioneer the hardware-based isolation at either the page granularity (Nooks) or the fine-grained memory granularity (Mondrix) to prevent faulty
driver tampering with kernel data structures. Heter-device can provide page granularity protection, and offer more deployability since it requires neither the modification to driver or kernel source code, nor a special hardware support. Virtual machine technique has also been applied into the kernel protection from buggy drivers, e.g., [55] and [40] isolate drivers by running a subset of drivers in a separate OS/VM domain, thus achieving both driver reuse and isolation. Basically, Heter-device can provide the same deployability with an affordable performance overhead.

**User Mode Driver Based Protection.** User mode device driver framework has been proposed recently to de-grant the driver code’s privilege, thus ensuring the kernel’s integrity ([12], [13] and [54]). However, they either suffer from significant performance degradation ([15] and [58]), or require complete rewriting of driver code and modifications to the kernel ([13] and [54]). Concerning the performance issue, Microdrivers [42] present a novel approach to split driver code into both user mode and kernel mode execution, with only performance-sensitive code remaining in the kernel. Based on Microdriver, [22] protects kernel from vulnerable driver by mediating all data transfers from the untrusted user-mode execution to the kernel-mode execution to preserve kernel integrity. As noted in [22] however, it cannot protect kernel from malicious kernel mode driver code since the kernel mode execution is fully trusted. In addition, Microkernel (e.g., Nexus [82]) can verify driver’s behaviour against a safety specification leveraging hardware isolation. Although the reference validation mechanism in [82] is effective in isolating drivers on Microkernel architecture, it is generally not applicable to most commodity operating systems.
**Software Based Protection.** SFI [80] and XFI [39] can isolate kernel extensions with low runtime overhead, but they require instrumentation binary rewriting. SafeDrive [95] uses a novel type system that provides fine-grained kernel extension isolation to achieve memory error detection and recovery, while Heter-device focuses on malicious driver or exploited driver detection. The seminal work Decaf [64] separates performance-sensitive code and generates a customized kernel interface that allows the remaining code to be moved to Java programs in user mode. Termite [67] presents a concrete driver synthesis approach to produce a new driver implementation based on a formal device specification and OS interfaces.

### 2.2.4 Application State Preservation

DRASP is the first workable approach that can preserve the continuity, accumulated state and correctness of service application even in the scenario when driver code is compromised, independent on any device specific semantics or heuristics. Existing application recovery techniques ([62], [84], [51], [65] and [30]) can preserve the continuity and accumulated states only in the scenario when the application is compromised, rather the kernel. Most of them ([62], [84] and [51]) take periodical snapshot of the application state and log system events during routine execution.

Upon exploitation to the protected application, dependency tracking ([52], [62], [90]) and [91] is applied to analyze the intrusion “footprint” across the system. Rollback/roll-forward actions are performed to get rid of the intrusion harm, meanwhile the clean state of the application can be preserved with best effort. Failure-Oblivious Computing [65] is proposed to continue server execution through memory error exploit
by inserting checks that dynamically detect invalid memory access. [30] automatically
detects and repairs data structures that violate a data structure consistency specifica-
tion, thus enabling the program to continue executing productively even in the face of
otherwise crippling errors.
Chapter 3

Comprehensive Intrusion harm Analysis

3.1 Introduction

Post-mortem intrusion analysis of production server systems has been a too familiar task for enterprise security technicians for many years [41, 79], e.g., locating the intrusion “breakin” and identifying the intrusion “footprint”. Due to diversity of the vulnerabilities, creativity of the attackers and complexity of the server systems, this too familiar task continues to bother enterprise security units for years. They have to push their security technicians to take the technical training of enterprise intrusion analysis [4, 6] and to struggle with the tedious, error-prone work of comprehensive diagnosis of the victim server systems.

Hence, lots of automated analysis frameworks have been proposed by researchers, but these frameworks either lack completeness (doing system events dependency tracking) [96, 16, 53, 46, 14, 3, 52] or are not feasible to be deployed for production workload servers (doing dynamic taint tracking) [87, 73, 70, 47]. On one hand, system events dependency tracking, i.e., at a higher abstraction level such as system calls, is lightweight and can be deployed on production workload servers. However, it cannot provide precise or comprehensive intrusion analysis due to its inability to capture all the relevant system execution flows, e.g., the infection propagation traces across kernel. On the other hand, dynamic taint tracking instrumentation is desired when pursuing completeness in
intrusion analysis. However, it will incur considerable runtime overhead on the order of 10-40X \([87, 70]\), which is much beyond the tolerance of production workload servers.

To resolve the above conflict, I present PEDA (Production Environment Damage Assessment), a system that can provide cross-layer comprehensive intrusion analysis for production workload servers with lightweight runtime overhead. PEDA decomposes the onerous intrusion analysis work into three tasks: lightweight logging, intrusion root identification, and infection diagnosis. The first task records a small subset of whole system execution information with minimal overhead; the second generates system dependency graph and locates the fine-grained intrusion break-in to the server systems; the last does decoupled dynamic-taint-tracking based infection diagnosis on an out-of-band “system execution replay” platform. These three tasks are not separated but instead, correlated: Task 2 yields the taint seed(s) for Task 3 to do dynamic taint tracking, while the logging information collected by Task 1 is used by both Task 2 to generate system dependency graph and Task 3 to replay the “has-been-infected” execution.

By decoupling the workload of intrusion root identification and infection diagnosis from the routine server execution, PEDA ensures “same-as-before” routine server system execution with minimum interference and runtime overhead. Moreover, only the replayed execution needs to be done on top of binary-translation based virtual machines, e.g., QEMU, to ensure the fine-grained infection diagnosis. PEDA integrates a heterogeneous virtual machine migration module to overcome the device emulation incompatibilities between off-line analyzing virtual machine and on-line hardware-assisted virtual machine.
Whether this approach can succeed depends on whether the fidelity of replay can be preserved. By “fidelity”, I mean during replay, the infection diagnosis can faithfully reveal what had happened on the server systems through the whole attack/intrusion/infection process. PEDA is able to replay the production workload server execution with high “fidelity” by combining the Revirt whole system replay technology [37, 86] with heterogeneous VM migration. Thus, during replay, dynamic taint tracking [87, 73] can be applied to reveal what had happened to the victim server system during the first run.

However, PEDA has to deal with two issues before applying the taint tracking technique. First, existing intrusion analysis techniques cannot provide the suitable “taint seed” for dynamic taint tracking. For instance, intrusion detection system (IDS) in many cases can only notify some intrusion symptoms instead of the intrusion root, and backtracking [52] can only identify the system-object-level intrusion root. In order to tackle this issue, PEDA “dips” further down to the raw memory or disk storage segment, identifies intrusion root there, and provides the fine-grained taint seed to dynamic taint tracking. Second, though dynamic taint tracking reveals comprehensive instruction-level infection diagnosis for the victim serve systems, the lack of semantics at the system object level makes it unreadable to human beings. Therefore, I develop the reconstruction engine to bridge the semantic gap by mapping each instruction flow dependency with system objects to provide both comprehensive and meaningful infection diagnosis for victim server systems.

In summary, I made the following contributions:

- PEDA is the first work that systematically investigates the fine-grained intrusion analysis problem for on-line production workload servers. PEDA decomposes the
complicated intrusion analysis work into lightweight logging, intrusion root identification, and infection diagnosis tasks, and accomplishes it with both the coarse-grained and fine-grained intrusion root identification and comprehensive infection diagnosis of the whole server system.

- I design and implement a heterogeneous virtual machine migration framework, which enables the decoupled infection diagnosis on a binary translation based virtual machine and the server production workload on a much faster hardware-assisted virtual machine.

- PEDA is the first system that can identify the fine-grained intrusion root at the granularity of memory cell or disk storage segment, thus bridging the “gap” between system call level backward tracking and instruction level forward tracking.

- To semantically understand the intrusion analysis provided by PEDA, I also implement dynamic reconstruction functionality to help pinpoint the system-object-level taint propagation. In this way, I am able to provide the cross-layer comprehensive infection diagnosis both at the binary and system object granularities.

- The implementation of PEDA is completely on the Virtual Machine Monitor or coordination host, which can be generally considered as a trusted platform for intrusion analysis.

3.2 Problem Statement and PEDA Approach

In this section, I first present the problem of fine-grained intrusion harm analysis for production workload server systems, and then give an overview of PEDA approach.
3.2.1 Problem Statement

**Fine-grained Intrusion Harm Analysis for Production Workload Server Systems.** Whenever the Intrusion Detection System (IDS) detects any intrusion symptom, such as system binary modification through integrity check, the problem for the proposed PEDA system is to first quickly determine the fine-grained intrusion root at the granularity of memory cell or disk sector, and then start the dynamic taint analysis from the identified intrusion root to provide comprehensive and semantically meaningful infection diagnosis.

**Acceptable Runtime Overhead.** Fine-grained Intrusion Harm Analysis for Production Workload Server Systems actually integrates both the fine-grained intrusion root identification and fine-grained infection diagnosis. Although both of them relies on dynamic taint analysis (usually 10-40X runtime overhead), PEDA approach should develop a feasible way to render the routine system execution with acceptable runtime overhead (e.g., lower than 5%). Otherwise, the intolerable runtime overhead would cause PEDA approach infeasible or impractical for production workload servers.

**Comprehensive and Semantically Meaningful Infection Diagnosis.** The infection diagnosis for production workload server systems should be as comprehensive as possible, since any minor infection on the mission-complicated server systems would propagate exponentially. So the difference between malware analysis and infection diagnosis turns out to be what to focus, whether on code behaviour or system-wide effects.
Therefore, we applied the whole system instruction flow auditing to satisfy the requirement of comprehensiveness, and the semantic gap between instruction flow and operating system semantics is properly mapped to satisfy the requirement of semantics.

### 3.2.2 PEDA Approach

Figure 3.1 PEDA decomposes the intrusion analysis work into three phases: lightweight auditing phase, intrusion root identification phase and infection diagnosis phase. I implement five functional components: lightweight logger, dependency tracking engine, translation engine, infection analyzer and reconstruction engine. Below, I briefly
describe each phase with several components working in tandem to fulfill the desired functionality.

### 3.2.3 Lightweight auditing phase

Figure 3.2 shows the lightweight auditing phase of PEDA system. During routine execution of the production servers, the lightweight logger will periodically take a checkpoint of the whole server system, including disk, raw memory, CPU registers, RTC, I/O devices, DMA and etc., which in whole serves as the starting point of the replay. Moreover, non-deterministic events (e.g., timer interrupts and external inputs into the server, such as network packets, keyboard inputs and etc.) between contiguous checkpoints are also recorded for redelivery during replay.

In this dissertation, the non-deterministic events involve time and external inputs. Time refers to the exact point in the instruction sequence at which an event takes place.
For example, to replay an interrupt, I must log the instruction at which the interrupt occurs. External inputs refer to the data received from a non-logged entity, and enter the processor via a peripheral device, such as a keyboard, mouse, or network card. The deterministic execution of the server systems (from the same initial state) and the non-deterministic events redelivery can ensure the high “fidelity” of the replay, which helps PEDA to reveal “what had happened” since the intrusion occurred. During the first-run execution of the server system, PEDA also needs to record all the system operations that could cause potential dependency between system objects.

### 3.2.4 Intrusion root identification phase

Figure 3.3 shows the intrusion root identification phase of PEDA system. Whenever some intrusion symptoms have been captured (e.g., some critical files are missing or suspiciously modified), the dependency tracking engine will generate system-object
dependency graph based on system operations logged during lightweight auditing phase. Then the engine traces the captured symptoms back throughout the system object dependency graph to quickly identify the system-object-level intrusion root, typically the network-oriented process. To provide the fine-grained taint seed to infection analyzer, I expand the functionality of dependency tracking engine. It can either do a one-step-forward auditing to locate the buffers containing the taint propagation data or audit suspicious network packets processing flow to identify the intrusion packets. The translation engine translates the logging information recorded from the much faster hardware-assisted VM to a format that the analyzing binary translation based VM can “understand”. All the work of dependency tracking engine and translation engine is done at the coordination host off-line.

3.2.5 Infection diagnosis phase

When the translation engine finishes the system states and non-deterministic events translation work, the first-run execution of server system is ready to be replayed
on the binary tracking instrumentation with high fidelity. Figure 3.4 shows the infection diagnosis phase of PEDA system. The infection analyzer treats the fine-grained intrusion root identified by dependency tracking engine as taint seed, and starts the fine-grained infection propagation analysis from it. PEDA applies dynamic taint tracking with both the data taint flow and the control taint flow to prevent some intended attackers crafting code that can evade the data flow auditing. Generally, the fine-grained taint analysis can only generate instruction flow dependency, which contains valuable information but lacks operating system semantics, so it is still infeasible to be the final infection diagnosis report for the security technicians. Therefore, I develop the reconstruction engine to bridge this kind of “semantic gap” between the instruction-level taint analysis and the meaningful OS semantics by dynamically mapping each instruction flow dependency with system objects. Through the coordination of the infection analyzer and reconstruction engine, the infection diagnosis report can be both comprehensive and full of semantics.

3.3 Design of PEDA System

In this section, I focus on the design of PEDA system by describing the details of analysis decoupling, fine-grained intrusion root identification, heterogeneous VM migration and cross layer infection diagnosis.

3.3.1 Analysis Decoupling

PEDA takes the idea of analyzing the intrusion during replay instead of the first-run for the following reasons. First, replay-based intrusion analysis can take over the
fine-grained analysis workload from the routine execution of production servers to ensure the performance of the server systems (considering the 10-40X overhead introduced by fine-grained analysis). Second, replay-based intrusion analysis can take advantage of the “already-happened” knowledge to reduce the workload of assessment. For instance, intrusion root identification can be done by logging system calls, generating dependency graph, and integrating IDS detected intrusion symptoms. During the first-run however, intrusion root cannot be easily identified. Last, by doing the intrusion analysis off the main production servers, PEDA offers the flexibility for production server system administrators to either aggressively restart (or continue running) the services for business continuity or just conservatively shut down the server for loss constraint. PEDA decouples the analysis work from the first-run of server system execution by analyzing the intrusion during replay, which requires recording the whole system state and non-deterministic events during the routine execution of server systems.

3.3.1.1 Checkpointing

PEDA periodically takes a snapshot of the whole server system to ensure that the replay shares the same initial state as the first-run. The different checkpoints enable the system execution replay to start at different time of the first-run. Thus, it is feasible and realistic to analyze a specific event on system, such as intrusion, with no need to replay the system execution from the very beginning of the first-run. The snapshot contains all hardware states, such as CPU registers, raw memory, disk, and I/O devices as well as DMA if any.
Pure stop-and-copy checkpointing involves halting the system execution, copying all hardware state to the log, and resuming the suspended execution. Such checkpointing mechanism has advantages in terms of simplicity but introduces service downtime which is proportional to the amount of hardware state (e.g., disk, memory, DMA, VGA, and etc.) of the running system. For the production servers with large disk storage and raw memory, this kind of checkpointing would cause quite some service downtime which is intolerable for the production workload servers. Rather, I observed that the amount of service requests during 2:00 am to 5:00 am is much more degraded than that during daytime for Amazon-style servers. Therefore, PEDA is designed to take the checkpoint infrequently (e.g., once per day during the service degradation period).

To further take advantage of the Amazon-style servers’ working style, PEDA trades off service response time with service downtime by using the pre-copy [77] checkpointing. In particular, PEDA initializes a pre-checkpointing phase, during which the whole disk and raw memory states are recorded. During this pre-checkpointing phase, the system continues execution, and I create and maintain shadow pages for any modification to disk or memory. After checkpointing the whole disk and memory, PEDA establishes a stop-and-copy phase, during which the system execution is suspended. During this stop-and-copy phase, all other device states, including DMA, vga, I/O devices and etc., are recorded into log. Moreover, the shadow pages created during the pre-checkpointing phase are committed to the originally recorded disk and memory, as well as the real hardware disk and memory. Finally, PEDA commits the end of checkpointing and resumes the execution of server system. By introducing a pre-checkpointing phase, the heavy
workload of checkpointing disk and memory is taken off from stop-and-copy phase, thus greatly reducing the service downtime.

3.3.1.2 Non-deterministic events logging

For the production workload servers, the non-deterministic events are mainly the service-requesting network packets, the administrator’s management keyboard inputs, and the I/O devices’ interrupts. The keyboard inputs happen quite infrequently for such kind of servers. Thus, it will only introduce little runtime overhead to the servers to log them directly using emulated keyboard of virtual machine. However, this is not the case for network packets. Typically, the production workload server deals with thousands of or even more service-requesting packets everyday. Hence, it will introduce intolerable overhead to log these packets by emulated NIC (network interface card) of virtual machine, because the NIC needs to perform an additional data transfer per packet.

PEDA successfully solves this problem by leveraging a router to replicate each incoming packet and to forward them to both the target server and the backend server. The backend server will log the contents of all these packets. Simultaneously, the emulated NIC on target server will only record the header identification information of each packet. During the intrusion root identification phase, the translation engine will associate each logged packet with its identification information. All the I/O devices’ interrupts to CPU are logged through the device emulation code of virtual machine. In order to exactly redeliver the interrupt during replay, I record the timing semantics at which the interrupt occurs. For instance, I log the time at which the keyboard input
arrives, and the instruction at which the corresponding interrupt is delivered to CPU. The time is logged by the unit of CPU clicks, while the instruction is logged using the program counter and the number of branches executed [20] by means of one provided hardware performance counter.

### 3.3.2 Fine-grained Intrusion Root Identification

Identifying the intrusion root is a critical procedure of intrusion analysis, since it determines where to patch the vulnerabilities and what to audit during analysis. I cannot simply rely on IDS to inform us the intrusion root, because intrusion symptoms detected by IDS often lag behind the real intrusion break-in. PEDA can identify the system-object-level (processes or files) intrusion root in a similar way as Backtracking [52] paper.

However, system-object-level intrusion root cannot be used by infection analyzer as taint seed, because dynamic taint tracking requires the taint seed at the granularity of memory cell or disk segment to ensure the precision of taint analysis. For instance, if considering a process as taint seed, then all the operations done by the process should be tainted and the whole address space of this process should also be tainted since it is compromised. This will generally result in the taint explosion throughout the server system with high false positive. Thereafter, the infection diagnosis will cover much more than the actual intrusion propagation, thus hurting not only the efficiency but also the precision (correctness) of infection diagnosis.

Therefore, PEDA implements one-step-forward-auditing and suspicious packet auditing to “dip” further down to the raw memory cell or disk storage segment to identify
3.3.2.1 Dependency Graph Generation

During the lightweight auditing phase, the lightweight logger records all the system operations that could cause system object dependency. These system calls are delivered to dependency tracking engine to generate dependency graph during intrusion root identification phase. I specify system object dependency as a source object, a destination object, and a specific time. For instance, if one process reads a file, then the file is the source object; the process is the destination object; while the time is defined as when the process issues the “read” operation. Since system objects are generally processes and files, I define two categories of system-object dependency: process/process dependency and process/file dependency. PEDA records the process id issuing the system call,
parameters of the system call, as well as the system call issuing time or sequence, and correspondingly associates them with source object, destination object and time.

Taking the system call auditing records as input, the dependency tracking engine will generate the system object dependency graph during the intrusion root identification phase. A piece of system call auditing records is shown in Figure 3.5, while a segment of the dependency graph is shown as an example in Figure 3.6. Each node denotes a system object either as source or destination. The $\rightarrow$ denotes the dependency relationship between the source object and the destination object at time $t$. Specifically, I applied different policies to different system-object dependency categories in order to generate system-object dependency graph as below. Since the graph may grow quite large and produce false positive results on taint propagation, PEDA performs graph pruning to reduce the storage size and false dependency. For instance, I do not consider situations like independent process termination, irrelevant signals, or accessing dummy objects like stdin/stdout and /dev/null.
3.3.2.2 Intrusion Root Identification

When any intrusion symptom is detected by IDS, such as some files are missing or maliciously modified, the dependency tracking engine switches to the intrusion root identification mode. It traces the system-object dependency graph backward from the detected intrusion symptoms, and identifies the fine-grained intrusion root which can be treated as taint seed for instruction level taint tracking in the infection diagnosis phase. The system-object-level intrusion root identification is straight-forward, locating the detected intrusion symptom objects on the dependency graph, tracing the dependency chain back with timestamps, and eliminating uninfected objects from the graph as transformed from Figure 3.6 to Figure 3.7. Therefore, I can obtain the intrusion propagation flows with only the infected system objects as shown in Figure 3.7. For the production workload server systems with constrains of physical access, the intrusion break-in should mainly occur at the network-service-oriented applications. Therefore, I trace back the intrusion propagation flow and locate the very beginning network-oriented process, which should be the system-object-level intrusion root. Besides, system security
technicians can also specify a set of vulnerable services and ports to further refine this procedure. Furthermore, PEDA also records the intrusion break-in timestamp, the time when the intrusion root performs the first operation which eventually propagates to the detected intrusion symptoms.

To bridge the gap between the system-object-level intrusion root and the dynamic taint tracking, I develop a straightforward but effective method, one-step-forward-auditing, to locate the fine-grained intrusion root for infection analyzer. Once I identify the system-object-level intrusion root (generally a network-oriented process), I examine the system calls issued by this intrusion root process from the intrusion break-in timestamp. Generally, processes need to issue system calls to ask kernel to help them affect the “state” of other processes or files. So by examining the system calls issued by the intrusion root process, I can identify the processes and files whose state is affected by this intrusion root process. By manually analyzing the system call dependency graph, I can identify several intrusion chains, starting from the intrusion root process and ending at the intrusion symptoms. Furthermore, I locate the last merging node of those chains. No matter the node represents a file or a process, I find the set of system calls that establish the dependency flow from this merging node to the following files or processes along the intrusion chains.

By manually analyzing the parameters of these system calls (typically write, send, etc.), I can locate the buffers propagating the data flow from the intrusion root process to the affected processes and files. These buffers actively expand the intrusion from the merging node to the one-step-down processes or files, and then throughout the server system. Therefore, treating these buffers as the fine-grained taint seed for dynamic
taint tracking can provide the precise and comprehensive infection diagnosis results. As shown in Figure 3.7, Process 1 is the system-object-level intrusion root, and I locate all the buffers contributing to the intrusion propagation and treat them as taint seeds for infection analyzer.

There are several drawbacks of the one-step-forward-auditing method to identify the fine-grained intrusion root. First, it cannot provide any information regarding how the intrusion root object is compromised, so there is no way for the infection analyzer to trace the intrusion from the very beginning. Second, it relies on an implicit assumption that the backward tracking can extract at least all the one-step-away infected objects from intrusion root in the dependency graph. Then, the one-step-forward system call auditing can capture all the infection propagation out from the intrusion root object. This can be generally true, but some intended attackers aware of backward tracking can craft intrusion programs to evade such kind of auditing, thus rendering the one-step-forward-auditing vulnerable and imprecise.

Therefore, I provide an alternative way to identify the fine-grained intrusion root. I rely on the general belief that the intrusion should come from the network packets, thus try to associate the intrusion packets with the detected infection symptoms. If any packet finally “contributes” to the infection symptoms, I identify it as the intrusion root and provide the receiving buffer or storing disk sectors of this packet as the taint seed to infection analyzer. However, tracking everyday thousands of packets to the enterprise-level production workload server is generally infeasible. Since I have already identified the system-object-level intrusion root with timestamp, I have a rough idea of where (intrusion root object) and when (before the timestamp) the intrusion breaks into the
system. Thus, I only need to track the network packets sent to this intrusion root object (generally an application) before the time it propagates the dependency towards the one-step-away objects in the dependency graph. Furthermore, it is relatively feasible for system security technicians to refer to the production server firewall’s “whitelist” to filter out the packets from trustworthy remote identities. I can also start multiple packet-auditing instances simultaneously to further increase the auditing efficiency. Generally, system security technicians can use some open source packet filter tools to quickly and accurately identify packets received by one particular process during a specific period of time.

After a much smaller set of suspicious packets is identified, I use the following technique to decide the actual intrusion packets. The dependency tracking engine can leverage the translation engine to first replay the server system execution. During the high fidelity replay, I audit the data flow of those packets since they entered the receiving buffer by applying instruction-level taint tracking. If any packet is manipulated by the intrusion break-in application to hit the system-object-level intrusion propagation chain as shown in Figure 3.7, then the corresponding packet should be considered as the intrusion packet. Note that several packets instead of one may contribute to one single intrusion. Generally, when one intrusion packet has been identified, I can use the identification information contained in the header of that packet to swiftly locate other intrusion packets if any. Once all the intrusion packets are identified, I specify the buffers receiving these packets or the disk sectors storing these packets as fine-grained intrusion root, and provide them to the infection analyzer as the taint seed for the dynamic taint tracking.
Dynamic taint tracking needs to be implemented in the binary translation based VM. Thus, a direct way to do decoupled analysis is to let the server system run on top of such kind of VM during the routine execution. When the intrusion analysis is needed the recorded VM image is migrated onto instrumentation platform implemented in such kind of VM. However, the problem of this approach is the runtime overhead introduced by the VM binary translation, — 3-4X slower than the native execution, which is typically intolerable to the production workload server systems. PEDA allows the server systems with lightweight logger to run atop the hardware-assisted VM (such as Xen) to provide near native-speed during the routine execution, and enables the replay-based fine-grained infection diagnosis to be done atop the binary-translation based VM (such as QEMU) to capture the comprehensive infection effects. In particular, PEDA integrates the heterogeneous VM migration functionality to help the latter (binary-translation based VM) “understand” the system states or events recorded by lightweight logger in the former (hardware-assisted VM).

Typically, different virtual machines use different device emulation techniques, so “translating” the device state of one virtual machine to that of the other is not an easy task considering the amount of devices critical to system execution. One way to avoid

<table>
<thead>
<tr>
<th>Devices</th>
<th>qemu-dm</th>
<th>xen hvm</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIC, vga, keyboard, mouse, dma, timer, and etc.</td>
<td>CPU registers, memory, ioapic, apic, pic, pit, and pmtimer.</td>
<td></td>
</tr>
</tbody>
</table>
such kind of device states translation is to consider each device as a black box with input
data flow from outside and output data flow to CPU. From the CPU’s point of view,
the data flow from each device to CPU is merely a flow of raw data. Hence, it is feasible
to implement the decoupled analysis by directly recording the data flows out from the
devices to CPU and during replay redelivering them to CPU without caring about the
device emulation incompatibility [26]. However, this “bypass” of the device emulation
incompatibility is traded off by performance loss. Generally logging raw data out of
each device introduces much more overhead than just logging the external inputs to the
devices (considering the situation of a large amount of data access from disk).

With much concern of production workload server’s performance, PEDA takes
the idea of directly recording the external input to the devices and leverages the trans-
lation engine to eliminate the device emulation incompatibility. PEDA simplifies the
whole system translation/migration work by choosing Xen as hardware-assisted VM and
QEMU as binary translation based VM, because the device emulation of Xen-HVM relies
on the QEMU “device manager” (qemu-dm) daemon running backend in domain 0 to
provide I/O virtualization. However, several devices such as CPU registers, apic, ioapic,
pic and etc. are emulated by Xen-HVM itself as shown in Figure 3.8. Therefore, the
translation engine needs to translate the device states recorded by Xen itself to those
recognizable to QEMU.

3.3.4 Cross Layer Infection Diagnosis

The infection analyzer is implemented inside the binary translation based whole
system emulator, which provides an abstract view of a series of hardware such as CPU
 registers, memory, and other devices. Therefore, it can provide down-to-bytes comprehensive infection diagnosis by auditing the emulated hardware. However, the “semantic gap” [24] really exists due to the lack of the meaningful information of operating system semantics. Therefore, I also develop reconstruction engine to bridge the semantic gap by mapping each instruction flow dependency with system objects.

### 3.3.4.1 Infection Analyzer

During the high fidelity replay of the server system execution, infection analyzer applies the dynamic taint analysis to pinpoint the infection propagation throughout the server systems. By treating the fine-grained intrusion root provided by the dependency tracking engine as the taint seed and auditing each executed instruction, the infection analyzer generates a whole system fine-grained taint propagation graph. In particular, I applied two kinds of taint propagation policies for dynamic taint analysis: data taint flow and control taint flow as shown in Figure 3.9.

**Data Taint Flow** Similar to several existing systems using dynamic taint analysis [87, 61, 28], I audit data transfer of each instruction to apply data flow dependency. As shown in Figure 3.9(a), for each computational instruction involving a source operand and a destination operand, I will taint the destination operand if the data derived from the source operand is tainted. Note the data could be stored in register (e.g., the third instructions in data taint flow), or fetched from memory (e.g., the first instructions in data taint flow). Furthermore, if the value used to compute the source operand address is tainted (e.g., the second instruction in data taint flow), the destination operand should
Data taint flow

Control taint flow

Fig. 3.9 Taint Propagation Flows

also be tainted. Last, the instruction fetched from a tainted memory address should result in the taint of the destination operand (the fourth instruction in data taint flow).

\[
\text{if}(\text{taint} == a) \quad \text{clean} = a; \quad \text{else} \quad \text{clean} = a;
\]

Considering the above example, the \textit{taint} will be propagated to \textit{clean} regardless of the value of \textit{a} without the direct data flow dependency between \textit{taint} and \textit{clean}. Therefore, I also enforce control taint flow policies to capture any taint propagation caused by the control flows. Specifically, as shown in Figure 3.9b, if the condition of the currently executing instruction is tainted (e.g., the second instruction in the control taint flow) or
the destination address of the execution redirection is tainted (e.g., the third instruction in the control taint flow), I will start tainting the following redirected execution until the conditional branches’ merging point identified by applying the static analysis [38].

Untainting The tainted registers or memory locations will become untainted whenever they are overwritten by untainted values or constants. Note the special cases of assignment instruction such as follows to set register eax to zero.

\[ \text{xor} \%eax, \%eax; \]

### 3.3.4.2 Reconstruction Engine

I develop reconstruction engine to bridge the semantic gap by mapping each instruction flow dependency with system objects. There are in general two ways to achieve this goal. The first one is to leverage a kernel module inside the guest OS [87], which “securely” communicates with underlying component inside the virtual machine monitor about the OS level semantics. However, integrating the kernel module to server systems is vulnerable to intended intrusion. Curious readers may argue that the kernel module may reside in the system without functioning until it is actually needed during replay. I think this is infeasible because the high fidelity replay relies on the precondition that the server system’s execution is deterministic. Only running the kernel module during replay will incur the execution incompatibility between the first-run and replay, so I have no way to redeliver the non-deterministic events exactly as they were redelivered to the system.
Thus, I adopt the second way to implement the reconstruction engine, totally at the virtual machine monitor level without any awareness of the guest OS. This is also different from static analysis of raw memory and *sysmap* reconstruction [50, 81], because the reconstruction engine extracts the system object semantics directly from CPU registers and dynamically maps the kernel address space with kernel data structures. This renders the reconstruction work a little more difficult, but I could ensure the correctness of replay and enhance the security of server systems. Currently I can reconstruct all process runlists, files, process address space, limited kernel data structures related to scheduling and etc (for Linux 2.6.20). When auditing each executed instruction, the reconstruction engine audits the memory operation and helps to relate the memory with process/file or kernel data structure.

### 3.4 Implementation Issues of PEDA System

I implement PEDA prototype on qemu-0.9.0 and Xen-3.3.0 to demonstrate its capability of comprehensive intrusion analysis for the production workload server systems. The goal and design of PEDA system pose various challenges, so I discuss the implementation issues of PEDA system through the rest of this section.

#### 3.4.1 Checkpointing and Non-deterministic Event Logging

On Xen Domain 0 management console, I add a new command `xm checkpoint`, which is passed to *Xend* via XML *RPC*. *Xend* responses to this checkpoint request by coordinating with Xen hypervisor to record raw memory, CPU registers, interrupt controllers and flush cache to virtual disk. It also calls qemu-dm to log other devices
states such as NIC, vga, keyboard, etc., and audits all the following keyboard inputs as non-deterministic events. After issuing the checkpoint command, the router forwarding network traffic to the server starts to direct the following traffic to both the server and the coordination host. The coordination host is modified to be able to receive and record these redirected packets as non-deterministic events for replay. Moreover, the `checkpoint` command will activate the system events auditing functionality to record all the system calls of the server system execution. This `xm checkpoint` command can be manually input by system administrator periodically, or automatically run from a pre-crafted script. The whole system states from checkpoint and keyboard inputs plus system call records during system execution are transferred to the coordination host through Gigabit Ethernet.

### 3.4.2 Translation Engine

The pre-configuration can ensure that both Xen and QEMU share the same image file as virtual disk, emulate the same type of device for Quest OS such as X86 processor, rtl8139 NIC, and access the same amount of memory. However, the emulation implementation of the same device can still be different. For instance, considering the emulation differences of IOAPIC between Xen hvm and QEMU, I observed that the significant differences are the number of IOAPIC pins, and the definition of each redirectory entry as shown in Figure 3.10. In order to eliminate such kind of device state incompatibility, I refer to Intel IOAPCI datasheet [5] for the definition of each IOAPIC pin functionality, and manually match each redirectory entry at the source code granularity of Xen and QEMU.
I summarize several categories when matching each entry of Xen and QEMU devices as follows. (1) The same name with the same type, e.g., \textit{IOAPIC\_NUM\_PINS} in Figure 3.10. Both Xen and QEMU define it as a macro, but with different values. For this kind of incompatibility, I simply choose the smaller value and discard the redundant part. (2) The same name with different types, e.g., \textit{ioregsel} or \textit{id}. For these variables, their values are typically holdable by the smaller storage type. So I shrink the storage size to the smaller type. (3) Different name with different types, e.g., \textit{base\_address} in Figure 3.10(a) and \textit{irr} in Figure 3.10(b). For these variables, I match them together because I refer to the source code of both Xen and QEMU and are aware that they serve for exactly the same functionality for IOAPIC device. I cope with them in a similar way as (2). (4) Similar name with similar type, but significantly different contents, e.g., \textit{redirtbl} in Figure 3.10(a) and \textit{ioredtbl} in Figure 3.10(b). By referring to the source code of Xen, \textit{redirtbl} is defined as a union, and each functional byte is further specified by its member \textit{field}. In contrast, when referring to the source code of QEMU, \textit{ioredtbl}, each functional bit is defined as a macro. For such kind of incompatibility, I carefully refer to the Intel IOAPCI datasheet and the abbreviative definition to match each of them.

The above procedures need to be applied to each device emulated by Xen hvm itself in Figure 3.8. Note that even for the devices which Xen relies on qemu-dm to emulate, there are still format incompatibilities critical to the success of the heterogeneous virtual machine migration.
#define VIOAPIC_NUM_PINS 48
#define IOAPIC_NUM_PINS 0x18
#define APIC_LVT_TIMER_PERIODIC (1<<17)
#define APIC_LVT_MASKED (1<<16)
#define APIC_LVT_LEVEL_TRIGGER (1<<15)
#define APIC_LVT_REMOTE_IRR (1<<14)
#define APIC_INPUT_POLARITY (1<<13)

struct hvm_hw_vioapic {
    uint64_t base_address;
    uint32_t ioregsel;
    uint32_t id;
    union vioapic_redir_entry {
        uint64_t bits;
        struct {
            uint8_t vector;
            uint8_t delivery_mode:3;
            uint8_t dest_mode:1;
        } fields;
    } redirtbl[VIOAPIC_NUM_PINS];
};

struct IOAPICState {
    uint8_t id;
    uint8_t ioregsel;
    uint32_t irr;
    uint64_t ioredtbl[IOAPIC_NUM_PINS];
};

a. IOAPIC emulation from Xen  

b. IOAPIC emulation from Qemu

Fig. 3.10 IOAPIC Emulation by Xen and QEMU

3.4.3 Infection Analyzer and Reconstruction Engine

As a binary-translation based emulator, QEMU enables the implementation of instruction-level taint analysis for infection diagnosis. Each executed instruction is audited before QEMU translation block works to keep consistent with the view of Quest OS. Since I have located either the packet containing the intrusion root or the buffers propagating the infection, the infection analyzer can be configured to start working only when that corresponding packet arrives or that corresponding process issues the specific system call. This can greatly reduce the waiting time of the infection diagnosis report even though the whole system states is daily checkpointed.

Moreover, I implement the reconstruction engine to provide operating system semantics. For instance, to dynamically reconstruct process lists, I start from a structure
env defined by QEMU to emulate the processor for the virtual machine. From env, I can obtain all the registers such as tr to locate the kernel stack of the currently running process. At the bottom of the kernel stack resides the thread_info structure, which includes a pointer to the task descriptor (defined as task_struct in Linux). Through the task_struct, I can obtain all the information related to one process, such as the data structures describing the virtual memory, scheduling status, open files, and inter process communications. Furthermore, from the pointer tasks in the task_struct, I can locate all the processes on the Guest OS, and also their profiles. The reconstruction engine works in an on-demand style. With the help of other registers and the process profiles, it identifies all the file objects and part of kernel data structures (almost 50 %), which are sufficient to demonstrate the cross layer analysis results of two case studies in the evaluation section.

3.5 Summary

PEDA is a systematic approach doing post mortem fine-grained intrusion analysis for production workload servers. It helps security technicians swiftly identify the fine-grained intrusion root “break-in” to the server and precisely pinpoint the infection propagation throughout the server. PEDA effectively decouples the analysis work off the on-line server execution by novelly integrating the backward system call dependency tracking and forward fine-grained taint analysis. The proposed heterogeneous VM migration significantly reduces the runtime overhead of on-line server execution. To provide fine-grained seed to decoupled harm analysis, PEDA bridges the gap between backward system call dependency tracking and forward intrusion taint analysis.
by one-step-forward-auditing. The evaluation demonstrates PEDA’s advance over existing intrusion analysis systems in terms of efficiency and comprehensiveness. I believe that the comprehensive intrusion analysis functionality of PEDA system should have a profound impact on any system recovery framework.
Chapter 4

The Drivers Trustworthiness Analysis

4.1 Introduction

Drivers, especially third party drivers, could contain malicious code (e.g., logic bombs) and/or carefully designed-in vulnerabilities. Once got executed/exploited, such compromised drivers render the attackers the opportunity of leveraging drivers’ privilege to manipulate system integrity and data confidentiality. Even worse, some attackers have successfully stolen certification from benign third-party and easily obtained trust from the most cautious system engineers. For instance, mrxcls.sys, a driver digitally signed with a compromised Realtek certificate, may be viewed as trusted and loaded into industrial OS by system engineers. Once loaded, it injects malware Stuxnet into the victim OS, which in turn causes catastrophe in Siemens supervisory control and data acquisition industrial systems [9].

Fully assessing third party drivers before running them in most commodity server systems is challenging. First, static analysis of such drivers is not always possible due to the unavailability of their source code. Furthermore, carefully designed-in vulnerability or malicious code triggered by some specific logic are extremely difficult to be pinpointed during static analysis. Second, dynamic taint analysis (e.g., [87] and [27]) of driver code is generally infeasible, due to the unknown taint seed during assessment. Without an accurate reference model, tainting the entire code space of drivers can only reveal drivers’
behaviour, instead of distinguishing legitimate actions from malicious ones. Last, besides promiscuous attacks such as kernel integrity manipulation, some passive attacks, e.g., listening post, launched by compromised drivers are more difficult to be captured.

Previous research proposes to protect kernel integrity from drivers by confining the drivers’ execution context, e.g., Nooks [75], Gateway [72], HUKO [85], Device Driver Reuse and Isolation [55], and Mondrix [83]. Although these systems can effectively monitor drivers’ interaction with kernel functions or data, deploying such isolation approach to assess drivers would cause a large number of false positives or false negatives. For instance, both Gateway [72] and HUKO [85] rely on explicitly white-listed legitimate entry points for control transfer from drivers to OS kernel. However, such explicit and complete reference model is quite difficult to be established in practice. I observe that legitimate drivers indeed invoke kernel functions not defined in legitimate entry points occasionally, which results in false positive. Moreover, frequently invoking kernel APIs defined in legitimate entry points can also lead to Denial of Service attack due to resource starvation, which causes false negatives.

In this chapter, I present a novel driver evaluation approach, Heter-device, to comprehensively assess drivers against an implicit and complete model before putting any trust on them. Heter-device relies on virtual platforms to emulate heterogeneous device (Heter-device) pairs (e.g., Intel 82540EM NIC and Realtek RTL8139) for guest operating system replicas. Each replica loads heterogeneous drivers corresponding to the devices it runs on. Heter-device approach stands on the assumption that heterogeneous drivers should not have the same exploitable vulnerability due to their separated developing processes. So they provide an implicit and complete reference model for each other when
trustworthiness assessment is conducted via fine-grained auditing. Hence, by deploying Heter-device as a high-interaction honeypot, I can closely compare the divergence of two replicas when the vulnerable driver is being compromised and leveraged.

The two replicas with heterogeneous drivers are synchronized at the exported function entry points, which are declared by OS kernel and implemented by each driver. I start a fine-grained auditing of driver’s execution whenever kernel calls the corresponding driver functions. During driver’s execution, every jump, call or return to kernel or other kernel modules’ address space are logged for verification. The logs from heterogeneous drivers are parsed and compared to check any suspicious control flow redirection, e.g., one driver jumps to a kernel segment written by itself, while the other does not exhibit such behaviour. Moreover, any modification to key kernel data by drivers is recorded and verified against the heterogeneous drivers to check if it is a legitimate modification or a malicious manipulation.

I also deal with passive attacks launched from compromised drivers, e.g., network card driver intercepts incoming/outgoing packets and redirects them to remote entities. Thus, the network outgoing packets of the two replicas are audited and compared to find mismatch. Additional amount of traffic on one replica against the other suffices an alarm of confidentiality compromise. Finally, abuse of kernel APIs, such as spin lock or kernel memory allocation requests, may cause CPU or memory starvation. Hence, any call to these resource request APIs from drivers is also verified against heterogeneous drivers. By placing the synchronization and monitoring “sensors” in Heter-device, the honeypot can faithfully reveal multiple attack vectors of compromised drivers, including kernel integrity manipulation, resource starvation, and confidentiality tampering.
I target a honeypot or testing environment; accordingly, I implement Heter-device framework based on open source QEMU [19] project for the following reasons. First, QEMU facilitates the Heter-device architecture by providing heterogeneous device emulation options for several types of devices, e.g., sound card (sound blaster 16 or Gravis ultrasound GF1), Ethernet network card (Intel 82540EM NIC or Realtek RTL8139), video card (Cirrus Logic GD5446 Video card or Standard VGA card with Bochs VBE extensions), etc. Furthermore, it enables the fine-grained auditing of driver’s execution through binary translation blocks. Specifically, since each jump of register \( eip \) generates a new translation block in QEMU, I can simply monitor \( eip \) at the beginning of each translation block to capture the driver’s execution context, rather than auditing every instruction. Last, although the overhead of QEMU is significant, providing good performance is not so critical in either honeypot or testing environment.

4.2 Threat Model

In this chapter, I assume that the device drivers are untrusted, either with vulnerabilities that can be exploited locally or remotely, or inherently malicious. Furthermore, I only focus on the exploitations that are carefully designed and crafted by attackers. Otherwise, crashing the target system will definitely draw system engineers’ attention to trace such consequence back to the root cause. Last, as the base of Heter-device, I assume that heterogeneous drivers should have different vulnerabilities or different malicious code, in terms of where and what the vulnerabilities or malicious code are. Hence, at least one driver can serve as a criterion to verify and alarm the other compromised driver’s execution.
It is generally believed to be challenging to verify the behaviour of untrusted drivers in an efficient and robust way due to at least the following reasons. First, drivers in most commodity OS have exactly the same privilege as kernel and run in the same address space as kernel. Thus, any kernel module performing auditing or monitoring tasks may be manipulated by the compromised driver. Second, the attackers may leverage the compromised driver to tamper arbitrary OS components (e.g., function pointers, file metadata, system call table, etc.), to accomplish their intrusion goals. Hence, it is also quite difficult, if not impossible, to pinpoint the comprehensive auditing points covering all possible damages/harms that could be caused by compromised drivers. Last, even if it is possible to censor drivers’ execution efficiently and comprehensively, lacking a complete reference model makes the verification of drivers’ behaviour challenging. Significant false positive or false negative is expected.

In this chapter, I propose a novel approach, Heter-device, using driver-diversity-based replica as a complete and implicit reference model, to assess drivers in the following attack vectors:

**Control Flow Manipulation.** The control flow transition from driver to kernel is tampered by compromised drivers, e.g., jumping to a specific address in the middle of kernel functions, making suspicious kernel function calls to modify critical registers, etc.

**Key Kernel Data/Code Manipulation.** Compromised drivers tamper with kernel code, static global variables, or key dynamic data specified by kernel developers or system engineers, e.g., system call table, interrupt descriptor table, double linked list pointers in process control block, etc.
**Confidentiality Manipulation.** Compromised drivers intercept bypassing information or access sensitive files, and send them out through network to remote unknown entities. For instance, compromised NIC driver intercepts all the incoming/outgoing packets and redirects them to attackers’ machine.

**Resource Starvation.** Compromised drivers abuse critical resources and incur denial of service, e.g., dominating CPU by locking interrupts or exhausting memory by endless allocation request.

Since I assume OS kernel is fully trusted, I don’t verify the control flow transition from OS kernel to driver code, nor audit the driver’s data accessed by OS kernel. Furthermore, the function parameters and stack data passed between OS kernel and drivers are not verified currently, which could be leveraged by compromised drivers to tamper kernel integrity in certain ways. For instance, when calling a certain kernel API, attackers can launch a return-oriented attack to jump to other kernel functions through carefully crafted parameters. Such attack can indeed evade the auditing of Heter-device, but I believe that currently it requires significant manual efforts of attackers\(^1\). Finally, since Heter-device relies on underneath virtual platform to emulate heterogeneous devices for guest OS replicas, I assume the virtual machine monitor is in the trusted computing base. Exploiting bugs in virtual machine monitor, such as [11], and then controlling the guest OS are not in my scope.

---

\(^1\)The most recent work [48] fully automates the instruction sequence construction that can be used by an attacker for malicious computations. However, the side-effect of the construction time (2009 ms) and the runtime overhead (135 times slower) will cause significant divergence on the logs of the two replicas, which will be caught by Heter-device as CPU resource abuse.
4.3 Heter-device Design

In this section, I first describe the novel virtualized device diversity approach to efficiently produce driver-diversity-based OS replicas, then present Heter-device approach to evaluate drivers in multi-aspects.

4.3.1 Heter-device Architecture

Figure 4.1 shows the Heter-device architecture with the front stage replica and the back stage replica running as Guest OS atop the same Host OS. The device diversity is produced by virtual platform to emulate heterogeneous devices for the two VM replicas. The virtual platforms can run unmodified commodity OS by giving the Guest OS the illusion that it runs on top of “real” hardware. Thus, the guest OS will load corresponding drivers for the hardware devices that it regards as “real”. In this way, the virtualized device diversity approach gains the same security benefits as costly
hardware diversity in a much cheaper manner. For instance, software diversity approach enables two replicas to run on two separated emulated platforms, one with Intel 82540EM NIC (Network Interface Card), sound blaster 16 (sound card), Universal Host Controller Interface (USB controller), the other with Realtek RTL8139 NIC, Gravis ultrasound GF1 (sound card), Intel Open Host Controller Interface (USB controller).

The virtualized device diversity idea is inspired by both the hardware-based diversity approach and the sweeping deployment of virtual platforms (e.g., VMware, Xen, KVM, QEMU, etc.) in the production server environment. In this dissertation, I call the diverse devices with different models but performing the same functionality, e.g., Intel 82540EM NIC and Realtek RTL8139 NIC, as a pair of heterogeneous devices. As a result of pairs of heterogeneous devices emulated by virtual platform, the guest OS kernel of each replica will load heterogeneous drivers correspondingly, e.g., e1000 or 8139too kernel modules\(^2\). Except heterogeneous drivers, the guest operating systems on the front stage VM replica and the back stage VM replica are exactly identical in terms of kernel version, installed applications and services, other loaded modules, start-up scripts, etc.

The external input should be redirected to both the two replicas. I implement the input replication at the Host OS, totally transparent to the Guest OS replicas. Basically, every external input from network, keyboard, and mouse triggers both the device emulation modules of the two virtual machines. Since the input data from virtual disk is initialized by guest OS replicas, it does not need to be replicated. The output from the two guest OS replicas is intercepted and recorded by virtual machines. For

\(^2\)I focus the discussion on Linux operating system in this dissertation, but Heter-device is easily transported to other operating systems through reasonable efforts.
instance, the network output traffic from each replica is audited and verified to capture any confidentiality tampering through network. During evaluation, I ensure that only the output from the front stage replica is sent out, while the output from the back stage replica is discarded, to guarantee the correctness of communication context.

4.3.2 Heter-device Approach

There exist several challenges to assess drivers based on Heter-device architecture, so I abstractly present the system design to tackle these challenges in the following.

4.3.2.1 Address-alias Correlation

Through pre-configuration, both the front stage and back stage OS replicas can load root symbols (defined in System.map in Linux) into the same memory address. However, other dynamic kernel data may be loaded into different addresses, even if the data represents exactly the same semantics. For instance, with the same kernel version and configuration, the two OS replicas store the process descriptors of their root processes in the same address (pointed by the root symbol init_task). By traversing the double linked list of all the processes on the two replicas, I observe exactly the same process list, including kernel threads. However, the memory addresses storing all the other process descriptors, except the root process descriptor, do not match. Such address-alias of the same kernel data on the two replicas is prevalent and poses challenge to the auditing of kernel function calls and key kernel data accessed by evaluated drivers.

To tackle this challenge, I propose to correlate the address-alias kernel semantics of the front stage and back stage replicas. First, I need to reconstruct kernel semantics from
raw physical memory of each replica respectively. Due to the challenge of reconstructing dynamic kernel data, such issue catches researchers’ continuous attention recently, e.g., [50], [17], [36], [56], [35], etc.

Heter-device efficiently integrates both the out-of-VM and in-VM approaches to comprehensively reconstruct kernel semantics. All kernel exported function pointers can be referred to from root symbol definition (System.map file), which is identical for both the replicas through default configuration. Some key kernel data can also be referred to in a similar way, with additional effort of recursive identification of kernel data structures. Regarding dynamic data of both kernel and drivers, I insert a fully trusted kernel module into both the front and back stage replicas, which notifies underneath virtual platforms about the allocation/reclaim of kernel memory and loading/unloading of kernel modules. With the reconstructed semantics, address-alias correlation recursively maps the addresses of the same kernel semantics, either function pointers or kernel data structures. Hence, it makes possible the efficient auditing and verification of heterogeneous drivers’ execution.

4.3.2.2 Runtime Synchronization

Running the front stage and back stage replicas at large may incur “out-of-band” comparison of heterogeneous drivers’ execution on the two replicas. Although I delivered the replicated external input to the two replicas at the virtual machine monitor level simultaneously, the corresponding interrupt to CPU on each replica may not be “simultaneous”. Thus the actual processing of the interrupt on the two replicas may still be
“out-of-band”. Researchers have proposed interrupt-redelivery approach for deterministic replay, e.g., [37], [86], and [90], which could be leveraged by Heter-device to apply the exact-replay-style synchronization. However, due to the heterogeneous driver diversity introduced by Heter-device, synchronizing such diverse replicas poses quite realistic challenges, such as different instruction execution sequences.

I observed that although the implementation of heterogeneous drivers is different, they offer the same function interfaces to OS kernel. Such layered design of most operating systems implies that OS kernel only needs to know how to invoke the device driver’s methods, rather than to understand the detailed implementation of driver’s methods. Figure 4.2 and Figure 4.3 show the interaction among NIC driver, OS kernel, and other kernel modules. Besides function call returns, the control flow transition to NIC driver code must be through NIC interrupt handler as in Figure 4.2 or NIC driver function
calls as in Figure 4.3. For instance, NIC driver (Linux version) has totally 18 methods, with 8 fundamental (e.g., open, stop, hard_start_xmit, etc.) and 10 optional (e.g., poll, set_mac_address, change_mtu, etc.), indicating the operations that can be performed on this network card.

OS kernel declares the corresponding driver function pointers and initialize them during the loading of driver modules. These driver functions are the only entry points for the control flow to transit from OS kernel or other kernel modules to this driver. Since OS kernel is fully trusted, it will not redirect control flow to arbitrary driver addresses except driver function call returns. More importantly, these entry points are identical for heterogeneous drivers despite different implementation details of the driver functions. Address-alias of the entry points on the two replicas can be resolved by correlating the addresses of them together. Thus, the two replicas can be synchronized by the entry points of the same device driver functions.

Hence, the auditing and verification of drivers’ execution can be triggered by sensors monitoring the entry points. Specifically, when OS kernel’s execution encounters an entry point, i.e., OS kernel calls a driver function, the context of current execution is recorded on the two replicas separately. Then, all the following instruction sequences of the two replicas are audited respectively, until the return to the previously logged context. In particular, the entry point of the interrupt handler function deserves special attention, since nested interrupts (new interrupt comes during the processing of previous interrupt) may happen sometimes. Hence, each entry to driver’s interrupt handler

---

3Driver may also call kernel APIs during its execution, so the return to OS kernel address space does not suffice the end of driver’s execution. Instead, only the return to the caller’s execution context indicates the end.
Fig. 4.3 Interaction between OS Kernel and NIC Driver through NIC Driver Methods

function is sequenced, and strictly matched to the corresponding return. In this way, the driver’s execution can be identified apart from OS kernel’s execution.

4.3.2.3 Kernel Integrity Mediation

Compromised drivers can leverage the ultimate privilege to manipulate kernel integrity, e.g., hijacking control flow or tampering kernel data. Below, I present the approach of auditing such kernel integrity that could be tampered by compromised drivers based on Heter-device architecture.

Control Flow. For benign OS kernel and drivers, the control flow transitions among them can be well regulated by confining exported functions. For instance, Figure 4.2 and Figure 4.3 show that the control flow transition from OS kernel to drivers can be made by calling functions exported by drivers. Besides call return or hardware interrupt, the transition from benign drivers to OS kernel is through functions exported by OS
kernel itself or other kernel modules which may call exported kernel APIs on behalf of the calling driver. Since I trust OS kernel, OS kernel only calls functions exported by drivers to transit the control flow. However, compromised drivers may directly jump to any address inside kernel or other modules to continue execution. Moreover, they can inject malicious code into OS kernel memory using DMA, and subvert function pointers on stack to the injected code, to hijack the control flow transition.

Existing researches, e.g., Gateway [72] and HUKO [85], prevent such control flow integrity manipulation by isolating address spaces of OS kernel and drivers. Legitimate entry points for execution transition from drivers to OS kernel are explicitly listed, e.g., system root symbols in Linux System.map). However, I observe that legitimate drivers may invoke some kernel APIs not defined in their legitimate entry point set. Rather, invoking kernel APIs frequently in their legitimate entry point set can also lead to denial of service attack through resource starvation. Furthermore, compromised drivers may inject malicious code into their own stack or heap to launch attack without violating control flow transition policies.

The runtime synchronization facilitates the fine-grained auditing of drivers' execution, from the call to driver’s function to the corresponding return to caller. During the auditing of driver’s execution, every jump or call out of driver’s code address space is logged. These calls of kernel APIs or other kernel modules should be verified against the two replicas. Since the implementation of the heterogeneous drivers is different, strict verification of the sequence of their OS kernel API calls would always fail. However, to provide the same functionality, during assessment I observed a set of specific kernel APIs
is frequently called by heterogeneous drivers. For instance, the kernel API calls made by NIC converge at irq locking/unlocking and memory allocation/deallocation.

I expect system engineers to manually analyze and verify the logged kernel API calls made by heterogeneous drivers within each specific driver function. Based on my experience, most system engineers (even those not quite familiar with OS kernel) have a sense of which set of kernel APIs are relevant in a specific driver function based on the cross-checking reference model. In addition, it is also relatively easy for them to capture some outlier kernel API calls through the pairwise comparison for further verification. For example, one volunteer system engineer at the first glance, pointed out that rtl8139_open makes a kernel_thread call, while e1000_open does not. Although such behaviour is finally verified as benign, I believe that such significant variance deserves further verification.

Besides the outlier kernel APIs, the kernel APIs that are called with an extremely high frequency deserve further verification as well. For instance, endlessly calling resource request kernel APIs will cause resource starvation as discussed in Section 4.3.2.4. Repeatedly calling prepare_to_wait kernel API will put all the runnable processes into sleep. Such malicious behaviour can be easily captured through the comparison of the amount of each kernel API calls within a specific driver function against heterogeneous drivers. Furthermore, benign drivers typically will only write data, rather than code, into their own stack/heap, or DMA-mapped kernel memory. Thus, any jump/call to the address within driver’s stack/heap, or DMA-mapped kernel memory is strictly verified to check if such behaviour is general for the heterogeneous drivers to provide desired functionality.
If not, further verification should be given to the driver that exhibited such suspicious behavior.

Data Integrity. Drivers have the privilege to modify any kernel control-relevant or control-non-relevant data. Such modification by drivers should be strictly verified, rather than forbidden, since some of the modification might be legitimate to provide some desired functionality. For instance, previous Linux kernel does not export set_current_state API for drivers to change the state of a certain process. Instead, drivers have to directly set the state of current running process by current->state = TASK_INTERRUPTIBLE. However, compromised drivers may take advantage of those exported kernel APIs to hijack control flow, e.g., manipulating kernel control-relevant data, such as system call table, IDT, etc. A particular process can also be hidden by tampering kernel control-non-relevant data, such as pointers of double linked list processes.

Hence, I propose to verify the modification to key kernel data by heterogeneous drivers to capture any malicious manipulation. Beginning from the call to driver functions, copy-on-write is triggered on the memory storing the key kernel data on each replica, until the corresponding return to caller. Hence, the modification to key kernel data can be accounted to the corresponding drivers. However, driver’s execution may be disrupted by a preempted interrupt, which is handled by OS kernel and corresponding interrupt handler. The modifications during the preempted interrupt handling should be accounted to the driver that implements the interrupt handler function. These modification logs of each driver function are verified against heterogeneous drivers for any malicious manipulation.
I observe that most kernel data integrity manipulation is accompanied with control flow hijacking, which can be identified as discussed in control flow manipulation. Regarding pure data integrity manipulation, kernel developers or security engineers can provide a list of critical kernel data based on empirical experiences or referring to kernel critical data profiling [81]. Generally, when the amount of key kernel data to be verified becomes large, the runtime overhead and the false positive of Heter-device verification will become significant. Hence, I propose to select a subset of kernel integrity critical data as the verification candidate, e.g., system call table, IDT, critical function pointers in process descriptor, double linked list pointers, etc.

4.3.2.4 Confidentiality and Resource Consumption

Passive attacks, such as confidentiality tampering or resource abuse, are challenging to be identified or detected. Unfortunately, compromised drivers can leverage their ultimate privilege to maliciously intercept any data flow or repeatedly request any critical system resource, thus tampering confidentiality or crashing the system. Below, I discuss the methodology of relying on Heter-device architecture to capture such passive attacks during driver assessment, and present the framework I integrated into the approach.

*Confidentiality.* Compromised drivers can intercept bypassing data, call transmission (*hard_start_xmit* in Linux) function in network card driver, and send out to remote machines. Through control flow auditing and verification of heterogeneous drivers, the additional call to network card driver functions can be captured and identified as suspicious as discussed in Section 4.3.2.3. However, if network card driver itself is compromised, such data interception can be done totally within its own execution context,
without any call to functions in other kernel modules. Although data flow auditing and verification of heterogeneous drivers can be added to capture the data interception, it would incur significant runtime overhead.

In this chapter, I only focus on confidentiality leakage through network, that is, the intercepted data is transmitted to remote machines through network interface card. I assume that servers’ running environment has strict physical access restrictions. Thus, it is out of my scope that the compromised drivers intercept the data and write it to local disk, which is then fetched through local access. Based on the assumption, I monitor and verify the network output of the front stage and back stage replicas for any confidentiality leakage. When deploying Heter-device architecture, OS kernel and service applications on the two replicas are identical, and the incoming packets to the emulated network cards are exactly replicated. So the output from the two replicas should be kept in rhythm unless anomaly happens. Hence, the output from the two replicas are matched with the combination of receiver’s IP and packet sequence number. The additional traffic for information leakage from the compromised replica can be captured and alarmed.

Resource Consumption. Compromised drivers can launch various resource abuse attacks, and even cause denial of service due to resource starvation. Acquiring/releasing interrupt lock, allocating/freeing memory, etc., are benign operations for most drivers to provide desired functionality. However, such legitimate operations may be leveraged by compromised drivers to launch CPU or memory starvation attacks. Certainly, it is infeasible to restrict these kernel APIs from drivers, because benign drivers may not work or malfunction. Heter-device captures such resource abuse attack by strictly auditing and verifying the resource request kernel APIs issued by heterogeneous drivers. Although
different implementation of heterogeneous drivers may cause variance in system resource consumption, I believe significant variance must indicate suspicious driver, at least inefficient implementation of the driver. System engineers can easily set up a threshold of such variance to alarm resource abuse. Based on my experience, such variance threshold can be set from 5% to 15% based on various context for reasonable false negative and false positive.

4.4 Implementation

In this section, I present the implementation of Heter-device framework. I begin with Heter-device architecture deployment based on QEMU open source project, followed by address-alias correlation for the heterogeneous drivers based replicas. At last, I describe the implementation of the fine-grained mediation of heterogeneous drivers’ execution.

4.4.1 Heter-device Deployment

4.4.1.1 Software Diversity Architecture

Instead of deploying the replicas on costly real heterogeneous devices platforms, I implement the Heter-device architecture using QEMU, with one virtual machine as the front stage replica, and the other one as the back stage replica. I configure the virtual machine (QEMU) to emulate heterogeneous devices for the two replicas, i.e., one with Realtek RTL8139 NIC and Gravis ultrasound GF1, the other with Intel 82540EM NIC and sound blaster 16. Although other heterogeneous devices options are also available,
e.g., USB, video card, etc., I believe heterogeneous network cards and sound cards are sufficient to demonstrate the proof-of-concept of Heter-device.

The disk image file is replicated for the two replicas to ensure the same guest operating system (with kernel version 2.6.15), service applications, configurations, startup scripts, etc. Moreover, the two replicas are configured with the same amount of memory and networking model. Hence, the only difference between the two replicas is heterogeneous drivers, which interact with underneath heterogeneous devices. During the assessment of heterogeneous drivers, Heter-device serves as “honeypot” to trigger either the inherently malicious drivers or remote exploitation to drivers’ vulnerabilities.

4.4.1.2 Input Replication and Output Verification

To implement the external input replication to the two replicas, I insert a small piece of replication code into the host operating system kernel. Whenever there is any external input, i.e., keyboard, mouse, network packet, to the front stage replica, the inserted code on host OS kernel replicates the input and notifies both the two replicas for incoming events. Since the virtual machine I use (QEMU) behaves as a user process on host OS, the notification can be done either by signal or bit masking based on the context. In contrast, the network output from each replica is logged by the emulated network card of each virtual machine. On host OS, I implement a verification process examining the logs from the two replicas. In particular, it extracts the destination IP, sequence number information from each packet and matches the corresponding packets from the two replicas. A threshold of the amount of unmatched packets can be predetermined, to alarm any confidentiality leakage.
4.4.1.3 Synchronization of Replicas

I rely on both virtual machine and guest OS support to synchronize the front stage and back stage replicas at the granularity of driver function calls. I assume that host OS runs on multi-core hardware platform, thus each virtual machine is configured to run on a dedicated core for maximum CPU capacity. I craft a trusted kernel module to monitor the loading of heterogeneous drivers on each replica. For instance, it audits the procedure of initializing functions declared by OS kernel, e.g., the interrupt handler function and other functions registered by the driver. The memory addresses of these functions are sent to underneath virtual machine through a secure channel. Only the functions implemented by both heterogeneous drivers are selected as synchronization points for the two replicas. During runtime, the value of register eip is monitored on both replicas to capture the synchronization points, as discussed in Section 4.4.3.

4.4.2 Address-alias Correlation

I focus the implementation on Linux OS and start from a set of root symbols in System.map file. Since operating systems on the front stage and back stage replicas are with the same kernel version and configuration, the symbols and their addresses in System.map are exactly the same. Then I apply a CIL module [60] on the source code of guest OS kernel to automatically extract type definitions of kernel data structures. Finally, beginning from each correlated root symbol, I recursively reconstruct memory semantics based on type definitions of kernel data structures, and correlate the same data structure with address-alias.
In particular, in order to correlate *task_struct* of each process, I start from the data structure *CPUState* defined by QEMU to emulate the processor for virtual machine. All the CPU registers can be referred to through the instance of *CPUState: env*. From the register *tr*, I locate the kernel stack of the currently running process. At the bottom of kernel stack resides the *thread_info* structure, which includes a pointer to the *task_struct* of the corresponding process. By traversing the double linked list processes through the *task* pointers on the two replicas, I can obtain all the process descriptors and correlate their addresses together.

4.4.3 Fine-Grained Driver Execution Mediation

QEMU is a binary translation based virtual machine, which facilitates fine-grained auditing of guest OS execution. Instead of instruction-by-instruction translation, QEMU implements translation block to improve performance. Specifically, QEMU generates host code from a piece of guest code without control flow redirection or static CPU state modification. Thus, for each translation block, guest OS executes without QEMU intervention unless interrupt occurs. At the end of each translation block, QEMU takes over the control and prepares for the next translation block.

The translation block mechanism provides a perfect mediation approach for drivers’ execution. I can audit the program counter at the beginning of each translation block, which represents a control flow redirection, including *return*, *jump*, or *call*, etc. In this way, the entry into or the leave from driver’s code section can be recorded efficiently without monitoring every executed instruction. However, QEMU also implements translation block chaining for performance boost. In particular, every time when a translation
block returns, QEMU tries to chain it to previous block, thus saving the overhead of context switch to QEMU emulation manager. The translation block chaining indeed poses challenges for the control flow redirection mediation, since QEMU emulation manager may miss some redirections, i.e., some transitions between OS kernel and drivers.

In order to tackle this challenge, I trace down through the translation block chain whenever QEMU emulation manager begins a new translation block. QEMU defines *Translation Block* data structure for each translation block, where I can locate the program counters of this block and the next one along the chain. Hence, I can traverse the chain till the end to audit and record the program counter at the each control flow redirection. However, key kernel data cannot be recorded in this way since detailed execution context has not been established yet during the pre-traversing of translation block chain. In order to preserve the performance and key kernel data integrity, I mark the memory regions storing those key kernel data as non-writeable. Any attempt to write to the memory will be trapped to QEMU manager, and validated against the heterogeneous drivers. Currently, I assume that key kernel data always involves some static code, data, critical function pointers, etc.

4.5 Summary

In this chapter, I present a novel diversity based honeypot, Heter-device, to assess the trustworthiness of drivers from multiple aspects, including kernel integrity manipulation, resource starvation and confidentiality tampering. Heter-device relies on virtual platforms to emulate heterogeneous devices for guest operating systems, and correspondingly produce driver-diverse replicas. Fine-grained auditing and verification of
the heterogeneous drivers’ execution are conducted to capture any control flow integrity manipulation. Moreover, any modification to key kernel data by drivers is recorded and verified against the two replicas to detect kernel data integrity manipulation. Last, CPU or memory request and network output of heterogeneous drivers are also verified for any resource abuse or confidentiality tampering. The diverse replicas are deployed as honey-pot to audit and verify the heterogeneous drivers’ execution by placing synchronization and monitoring “sensors”. I also propose automatic address-alias correlation, a subset of kernel data for default integrity protection, and a set of policies to defeat resource abuse and confidentiality tampering.
Chapter 5

Survivability Analysis against Compromised Drivers

5.1 Introduction

With the rapid prevalence of E-commerce, MMO and social networking, the demand on service availability and continuity is increasingly crucial to enterprise servers or data centres. For service applications, downtime (due to integrity/correctness loss and recovery) leads to productivity and profit loss. Hence, fault tolerant systems ([63], [31], [57]) are widely deployed and software failure recovery systems ([62], [32]) are thoroughly studied. Further, even the service application is compromised, it is also feasible to preserve as much accumulated state and continuity as possible by means of the intrusion recovery systems ([62], [84], [94], [51]). However, stimulated by the significant commercial revenue, attackers begin trying to evade the existing auditing/recovering techniques by manipulating the service applications through the compromised kernel. With the kernel privilege, the successful exploit can tamper the critical data/code, even the metadata of the service application process unscrupulously and undetectably.

Nowadays, device drivers account for more than half of the source code of most commodity operating system kernels, with much more exploitable vulnerabilities than other kernel code [25]. This renders the attackers the opportunity to exploit the driver vulnerability to leverage the kernel privilege, since the drivers execute with kernel privilege in most commodity operating systems. With the unrestricted access to the whole
(kernel/user) memory address space, attackers can manipulate critical code/data or even the metadata of the service application process, besides launching denial of service attack by incurring driver fault for commercial benefits. In order to preserve the correctness, the only plausible way in such scenario currently is to restart the whole system as well as the running applications, regardless of any continuity or accumulated state. Checkpoints may have been taken for the running applications; unfortunately, due to the unpreparedness and difficulty in detecting attacks from compromised drivers, the security officer often does not know which checkpoints are clean and which are infected. As a result, attacks launched from compromised drivers can greatly hurt service availability and continuity.

In fact, I find that the difficulty in detecting such attacks is bounded with the unique role device drivers are playing in a computer system. For instance, attackers could exploit the bugs in a NIC driver, and delicately tamper some incoming requests and outgoing responses. From either the service application or the systems point of view, the driver functions well without any suspicious behaviour. However, the correctness of the service application is maliciously manipulated, and existing monitoring or IDS in many cases cannot detect such compromise.

In order to preserve the correctness and state the service applications against compromised drivers, in this chapter, I present DRASP (Diverse Replication based Application State Preserving), a novel approach re-using heter-device architecture to produce operating system replicas. By means of the corresponding driver diversity, I dynamically validate the responses, critical memory regions, and persistent data of the service application to swiftly detect malicious tampering of applications’ execution via driver
vulnerability exploit. The software diversity for intrusion detection idea has been studied in several works, such as COTS [78], Behavioural Distance [43], and [18]. All of them aim to detect intrusion or anomaly by comparing either the web server responses or application system call sequences on diversity based replicas. A seminal work N-variant [29] proposes address space partition and instruction set tag diversities to detect intrusions.

Once the proposed virtualized device diversity is deployed, the system replicas would need to load different drivers (e.g., as loadable kernel modules) for the diverse devices. Again, the idea is that different drivers should have different vulnerabilities, in terms of the location or the details of the vulnerabilities. Hence, one driver vulnerability exploitation can at most succeed in one replica, with the other replica surviving. Once the successful exploit tampers the applications’ code/data or compromises the applications’ metadata for commercial benefits, it can be detected through either response or state validation. Afterwards, the applications on the survival replica can immediately take over the workload to preserve the service continuity and accumulated state, while ensuring the correct execution.

I aim to detect any malicious attack that exploits driver vulnerabilities, which in turn manipulates applications’ execution for commercial benefits. Hence, the proposed mechanism is to audit the service-orientated commercial applications in terms of their responses, critical code/data on memory and persistent data. I classify the harm to the service applications by the malicious exploit as either transitional or persistent, and construct validation policies correspondingly, which are discussed in Section 5.3.1. Once any validation policy is violated, DRASP generates alarm reports for system administrator,
who can decide to transfer the service workload from the compromised replica to the survival replica for operate-through intrusion response.

To my best knowledge, DRASP is the first workable approach that can let applications operate through attacks launched from compromised device drivers. DRASP can dynamically detect and respond to driver-vulnerability-oriented exploits targeting commercial applications without relying on device specific semantics or heuristic. DRASP is flexible and very easy to be deployed, without any modification to the driver code, compilers, or operating systems. In addition, this architecture can be immediately applied to the existing fault tolerant systems, which often run replicas and verify their responses to client requests through comparison (e.g., [76], [88] and [66]).

5.2 Problem Statement and DRASP Architecture

The ultimate goal of DRASP is to preserve the correctness, continuity and accumulated states of any service application even in the scenario when the driver is compromised. Below, I summarize the problems in the scenario when attackers tamper the service application execution by exploiting the driver vulnerability, and then overview DRASP architecture.

5.2.1 Threat Model

I assume the Guest OS kernel extensions, especially the device drivers, are generally untrusted. The attackers can craft exploit code to specific driver vulnerability. Once succeeded, they leverage the privilege of the driver to unscrupulously and undetectably manipulate the service application for commercial benefits. Though this compromise
procedure is indirect and probably more effort-consuming, it can generally evade the existing intrusion detection and/or recovery techniques for the service applications, e.g., [62], [84]), and [51]. I argue that the attackers probably might prefer such indirect compromise procedure due to at least the following reasons. First, service application code is produced, maintained, and updated by experienced programmers within the service provider enterprise, and potentially by the community of the customers. However, the driver code development and maintenance generally involve much fewer resources. Hence, it is relatively easier for the attackers to find out the driver code vulnerability and figure out the plausible exploit. Second, with the compromised driver, the attackers have uncensored access to the whole system, rendering them multiple undetectable ways to achieve their goal. In contrast, after compromising the application, the attackers’ behaviour may be audited by existing application intrusion detection/recovery module. Last, by integrating forensic analysis for service application code and existing state of art techniques ([62], [71] and [51]), attackers know that exploiting the application will be quite challenging and have a good chance to not only be detected immediately, but also cause the application to be rolled back to clean state. They would rather go underneath to the driver code for stealthiness, effectiveness and efficiency.

5.2.2 Problem Statement

I consider two fundamental challenges to be tackled when attackers exploit driver code vulnerability to manipulate service application’s execution.

(1) How to detect the driver vulnerability exploitation in an efficient and swift way, such that the corresponding intrusion response can be made appropriately?
This is in general a quite difficult issue, even we know that the target of the exploit is one specific service application. First, since driver has exactly the same privilege as the kernel, any kernel module performing auditing or monitoring tasks may be manipulated by the compromised driver, which renders the monitoring module untrustworthy. Second, the attackers may leverage the compromised driver to tamper arbitrary OS components (e.g., meta data of the application or files accessed by the application), to manipulate the service application execution for commercial revenue. Hence, it is also very difficult if not impossible to predict the corresponding damage/harm caused by the compromised driver. Last, censoring the driver execution based on legitimate behaviour profiling may help to detect the anomaly quickly, but it will generally incur significant runtime overhead with several untrusted drivers under censoring.

(2) Aware of any intrusion to the driver code, would we still be troubled by how to preserve as much continuity and accumulated state of the service application as possible in an on-the-fly or near-zero downtime style, meanwhile maintaining the correct execution of the service application?

The correctness of the service application execution is tampered once the attacker begins leveraging the exploited driver to manipulate the application. Existing recovery strategies try to roll back the application execution as close to (before) the manipulation time point as possible, to preserve more accumulated states. Due to the significant runtime overhead, however, it is generally unfeasible to take the checkpoint of the whole enterprise OS quite frequently. Thus, some amount of accumulated states shall be lost even if they derive from legitimate operations and ought to be preserved. Meanwhile,
the analysis/decision time must be short, thus the clean checkpoint can be chosen immedi-
ately to maintain the service continuity. Existing fault tolerant systems cannot help
either, since all the replicas are identical and will be compromised in the same way by
the exploit.

5.2.3 DRASP Architecture

Figure 5.1 shows the DRASP architecture with the front stage replica and the back
stage replica connecting to one front-tier proxy. Similar to DRASP, I produce the device
diversity relying on virtual platforms to emulate heterogeneous devices for the replicas
to “virtualize” the hardware diversity. As a result of pairs of heterogeneous devices,
the guest OS kernel of each replica will load different drivers as loadable kernel modules
for the corresponding devices. The guest OS and the protected service application are
replicated on the front stage VM replica and the back stage VM replica, respectively,
with the same version and installation configuration. The two replicas are synchronized
in a loose style, to ensure that the state validator and response validator can verify the
“output” (either to the remote end or to the memory/disk) of the two service applications
effectively and efficiently.

I deploy a proxy in front of the two replicas to replicate service requests to them
and verify the corresponding responses from them. After passing the validation policies,
only the responses from the front stage replica are sent out, which means that only the
front stage replica can directly interact with the outside, while the back stage replica
is behind the scene as a criterion for verification. Thus, I can safely assume that the
malicious exploits would primarily target the front stage replica if the attackers have
no knowledge of the deployed DRASP architecture. Even if the attackers are aware of the DRASP deployment, it is still quite difficult for them to launch an exploit to the driver code on the back stage replica. This is because they cannot have the details of the driver version and its corresponding vulnerabilities/bugs due to the non-interactiveness of the back stage replica. Though the attackers can launch brute force attack to explore the driver vulnerabilities/bugs on the back stage replica, but I believe that lots of such suspicious exploitation packets will definitely catch the system administrator’s attention.

5.3 DRASP Design

In this section, I present DRASP, an novel approach to swiftly detecting driver code vulnerability exploit and preserving as much correctness, accumulated state and
continuity of the service application as possible. I first introduce the model and the methodology of DRASP. Then, I discuss the synchronization issues of the front stage and back stage replicas, and regulate the validation policies for the responses and states (critical memory/files of the service application) comparison. Finally, I discuss the on-the-fly intrusion response and extendability of DRASP architecture to detect other compromise stemming from driver vulnerability exploit.

5.3.1 Model and Rationale

Modeling the Harm to Service Applications. I aim to detect any malicious attack that exploits driver code vulnerabilities, and then manipulates applications’ execution for commercial benefits. I classify the harm to the service applications by the malicious exploit as either transitional or persistent. By transitional, I mean that by manipulating the service application, the request processing routine or the state information that the processing routine relies on are tampered. As a result, the response to the request diverges, thus producing immediate intrusion revenue. For instance, by manipulating the pricing data, the attackers can purchase anything for free.

By persistent, I denote a long-term-benefit-pursuing intrusion that achieves its goal by tampering state information of the service applications. For example, by manipulating the account records, the attacker’s property could be augmented significantly. However, the service responses generated may not be tampered, since they might not rely on the maliciously manipulated (application) state information. Based on the classification of the potential harm to service applications, I audit the service-oriented commercial applications in terms of their responses and critical state information. The state of the
service applications is defined as its meta data in OS kernel, its persistent data on disk, and its whole address space.

**Validation Mechanism to Detect the Harm** During the *transitional* harm procedure, since the responses generated by the victim service application must have been tampered by attackers, I can synchronize the replicas at the granularity of per packet processing, and verify the responses of the two service applications at the front tier proxy. The rational of the response validation is that as long as the state of the service applications and the service requests are identical, the replicated applications should generate exactly the same responses.

To defeat *persistent* attack, I also verify the state of the service applications, e.g., some critical memory regions (e.g., metadata, crucial code/data segment) and persistent data of the applications (opened files by the service application process) on the two replicas in an offline comparison style. Based on the results in evaluation section, most of the service application state should be identical after a semantic mapping. Certainly, the prerequisite of the state validation is that a well-designed synchronization mechanism can ensure the state of the service applications on different replicas in the same rhythm.

### 5.3.2 Synchronization of the front stage and back stage Replicas

I do not allow the two replicas to run at large by themselves. Instead, they are well synchronized at the granularity of system call abstraction to ensure the correctness of the response and state validation. There are several abstraction layers for synchronizing replicas. The most fine-grained granularity is the instruction level synchronization.
However, this abstraction cannot work for most diverse replication architectures, including DRASP. Due to the device driver diversity I introduced, the instruction execution sequences of the two replicas cannot be exactly same. One may propose to pick up several critical instruction sequences for synchronization. However, the selection of such kind of critical instructions is still an open research issue currently. In addition, the significant runtime overhead would almost prevent the replicas from progressing, which also hinders the instruction level synchronization approach.

Another abstraction for synchronization could be the memory state transition, that is, each “write” to memory address space. Similar problems also apply here. The “write” to the kernel address space or the application address space on the two replicas can be totally different due to the device driver diversity I introduced. Though I can develop filters to bypass those different “writes” and leave other identical “writes” for synchronization, the runtime overhead introduced by each-write synchronization and filtering is intolerable for most service applications in the enterprise environment.

The approach I take is a loose synchronization for the two replicas, — synchronizing system calls made by the two service applications on the two replicas. However, non-determinism might cause the system calls made by the two replicated applications non-comparable. Figure 5.2 shows the mode transition of typical service applications. The transition from idle mode to busy mode is triggered by the incoming packets as shown in Figure 5.2(a). If one service application stays one or more cycles in the idle mode than the other application due to the late arrival of incoming packet, the system call sequences of them will diverge a little bit. This divergence can be accumulated, which causes troubles to system call synchronization of the two replicas.
Intuitively, Figure 5.2(b) demonstrates the procedure of the incoming packets going through the NIC driver, OS kernel, and finally the service application, which undergoes the state transition. The service applications on the two replicas are installed and pre-configured exactly the same, and they are running on the OS with the same kernel version. Provided that the two service applications are both in the idle mode or in the busy mode processing the same request, the system call sequences of them must be identical. Hence, I further synchronize the NIC interrupt delivery of the two replicas, which ensures that kernels on the two replicas receive the incoming packet interrupt simultaneously. Though kernels on the two replicas may perform context switch interchangeably, the service applications will transit mode once it gets scheduled by kernel. The mode transition of the two applications can be ensured as identical, thus enabling the system call synchronization.

5.3.3 Response Validation for Intrusion Detection

Comparison of responses is widely used in the fault tolerance or server validation environment such as server validation [76], COTS [78] and Splitter [34]. I integrate the comparison of service applications’ responses into the DRASP architecture to detect malicious or exploited drivers targeting service applications. The two service applications must generate exactly the same responses upon the same request before any anomaly occurs. The front-tier proxy can replicate and forward requests to both the applications on the two replicas, and verify the responses from them through comparison.

In this chapter, I consider that the service application works on HTTP protocol. However, the response validation approach is easily transformed to the applications based
Fig. 5.2 Mode Transition of Service Application

on other protocols, e.g., FTP and etc. The HTTP response generated by the service application consists of two parts: the header and the body. The header format is generally well defined. I pick up a set of fields that deserves the comparison, i.e., the version of HTTP protocol, the status code, Content-Length, Content-Type, and Last-Modified. They are almost always sent along with each response by the service applications, and differences in these headers are highly suspicious. The body of HTTP response generally includes the data generated by the service application and specific to the request, such as data records read from file.

Comparing the body of the responses from the two applications involves bitwise matching. Any mismatch from the body bitwise comparison suffices to alarm the system administrator the compromise of the application on the front stage replica. However, to
avoid eavesdropping of sensitive information, traffic between the service applications and clients is transmitted as encrypted HTTPS traffic using SSL protocol. This poses the problems of both the request replication and response validation. Since the request is encrypted between the client and the application on the front stage replica, the application on the back stage replica has no way to understand the semantics of the replicated request. As a result, the response comparison is meaningless.

To eliminate the encrypted payload problem, I can maintain three concurrent connections, one between the client and proxy, the other two between proxy and the two replicated applications. In this way, the proxy in the middle can encrypt/decrypt all the traffic going through it, thus it can see all the traffic in the clear text\(^1\). Sometimes, the traffic between the client and service applications is compressed to reduce network load. Obviously, comparing the compressed payload is also meaningless. Thus, I also rely on the proxy to decompress and recompress packets whenever needed to ensure the correctness of request replication and response validation.

Algorithm 1 shows the response validation algorithm of DRASP. I set a timer to detect that either one service application is not able to response to a request, or it sends more responses than expected. Given an appropriate timer threshold, any of above anomalies implies the compromise of the service application on the front stage replica. Within the timer threshold, the responses made by the two applications are compared against each other based on the response validation policies discussed above.

\(^1\) The proxy needs to install appropriate SSL certificate to achieve full transparency. Otherwise, the client will prompt a warning message.
Algorithm 1 Response Validation

Initialize: clear new_response, pending_response, and timer, set interrupt for new_response and timer expiration.

1. repeat: /*waiting for interrupt*/
2. if (new_response) /*begin comparison*/
3. if (pending_timer == invalid)
4. pending_timer = get_time();
5. pending_response = new_response;
6. else /*pass validation*/
7. ret = compare(new_response, pending_response);
8. if (ret = true) send(new_response);
9. else alarm(mismatch);
10. else /*response wait timer*/
11. pending_timer = invalid;
12. free(pending_response);
13. alarm(timeout);
14. continue;

5.3.4 State Validation for Intrusion Detection

Most device drivers execute with kernel privilege, hence, malicious or exploited drivers have unrestricted right to tamper metadata of the service application, inject/modify some code or data on the memory space of the service application, and distort records on the opened files of the service applications. All the above methods can be leveraged by the attackers for their commercial benefits. The response validation works well to detect the transitional harm to the service application, that is, the response is immediately distorted by the attacker through exploited driver. However, for some intelligent ones with persistent harm in mind, e.g., manipulation of some records on the opened files/memory that are currently used by the service application, response validation might not detect such compromise until the responses are generated based on the manipulated records.
Table 5.1 Metadata Validation for Service-Application-Target Exploit

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocated resources</td>
<td>struct mm_struct *mm, struct files_struct *files, etc.</td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td>struct sched_info sched_info, int static_prio, etc.</td>
</tr>
<tr>
<td>Records in dependable files</td>
<td>struct files_struct *files, struct fs_struct *fs, etc.</td>
</tr>
<tr>
<td>Identity information</td>
<td>char comm[TASK_COMM_LEN], pid_t pid, etc.</td>
</tr>
<tr>
<td>Stack statistics</td>
<td>struct thread_struct thread, void *stack, etc.</td>
</tr>
<tr>
<td>System hook</td>
<td>System call table, interrupt table, etc.</td>
</tr>
</tbody>
</table>

Thus, I also regulate state validation policies for swift detection of such kind of compromise.

Comparing the state of replicated applications on two replicas involves the whole memory space, metadata, and dependable files of the service application processes. For practical concerns, I select a subset of critical code/data sections that will seriously affect the execution correctness of the service application for comparison. First, the code segment of the application shall remain unchanged once loaded into memory. The request processing routine of the two replicated applications must be identical, thus the code segment validation can help to detect any compromise that leverages driver vulnerability exploitation to tamper service application’s code segment.

Second, the metadata of the service application needs to be validated by comparison. The metadata of the application process is maintained and updated by OS kernel, but the compromised driver with unrestricted privilege can distort it for commercial profit. Table 5.1 summarize some key metadata that could be compromised by attackers to manipulate the correct execution of the service application. Although with kernel privilege of the compromised driver, the attackers have unrestricted right to
tamper anywhere of the kernel address space, the metadata of the service application is most critical and vulnerable provided that the attacker aims at commercial profit.

Last, some sensitive or critical data of the service application must be verified through comparison, since it determines the correctness of the application’s execution and enterprise’s revenue. I allow service application developers or system administrators to configure what should be verified by comparison, because they are supposed to know much better what is more critical and valuable. By default, the global variables, static data, and function pointers of the service application are verified by bit-by-bit comparison.

Obviously, intensive state validation helps to detect the compromise swiftly, while runtime overhead arises correspondingly. In this chapter, I design both the fine-grained time slice state validation and coarse-grained time slice state validation and allow dynamic switch between them to balance accuracy and performance. The fine-grained time slice state validation is system call synchronization based offline bitwise comparison. Whenever one system call is issued by one service application process, the state of it, including code segment, metadata, and dependable files, is logged. Since the two service applications are synchronized at the granularity of system call, the states (contained in the logs) of them must be identical.

The coarse-grained time slice state validation is request synchronization based offline bitwise comparison. The state of each service application is logged whenever one request is processed, and then verified through comparison. However, attackers could manipulate the service application execution during request processing and achieve their commercial profit. Afterwards, the application resumes normal execution with legitimate
state. I believe that such kind of compromise either distorts persistent data records of the application for future profit, or manipulates the response of the application for immediate revenue. Hence, the former can still be caught by the coarse-grained time slice state validation, while the latter will be caught by the response validation.

Figure 5.3 demonstrates the state validation procedure when the logs are taken by the two service applications. The logs are parsed into three different segments as discussed above: code segment, metadata segment, and critical application data segment. I rely on reconstruction engine to establish detailed OS semantics of each segment, and apply corresponding validation policy to conduct bitwise comparison of the two logging records. Any violation of the validation policy suffices the generation of compromise alarm to the system administrator.

Note that in some cases, the heterogeneous device driver vulnerability exploit can be detected as early as when the driver is being compromised, because the driver vulnerability exploit on the back stage replica may crash the corresponding OS. Hence, whenever the back stage system crashes while the front stage system functions well, this can be a good indicator of malicious exploit to the heterogeneous device drivers. OS error/fault can also cause the back stage replica to crash, thus, system administrators need to examine system logs on the back stage replica to identify the root cause of this crash.

5.3.5 Operate-through Intrusion Response and Extendability Discussion

With the response and state validation, the compromise to the service application through vulnerable driver code exploitation can be quickly detected. During normal
execution before any exploit, the two replicated applications are well synchronized at least at the per packet processing granularity. Hence, from the front tier proxy’s point of view, the responses to incoming requests arrive generally simultaneously. Due to DRASP architecture I deployed, the service application on the front stage replica will be the victim, while the one on the back stage replica survives.

Although the victim replica malfunctions due to the exploitation, the other replica still performs well and stick to the criterion of correct execution. Thus, it is safe for the system administrator to transfer the following service workload to the survival replica, without losing any accumulated state, continuity or correctness. Simultaneously, system administrator should also prepare to restart a new criteria replica from a clean VM image, and configure the corresponding heterogeneous devices emulation and service application, to continue the DRASP protection.
In general, the state validation is more generic to most service applications in contrast to the response validation, which focuses on HTTP protocol based service applications in this dissertation. However, the response validation is easily extended to be applied on other kinds of service applications, e.g., FTP protocol based. Integrating the semantics of FTP protocol response header, DRASP performs response validation exactly the same as the HTTP protocol headers comparison.

Though in this chapter, I focus on the discussion of service application targeted exploit, with the kernel privilege of compromised drivers, attackers can perform arbitrary malicious actions to take control of the victim system. For instance, attackers can install rootkit, malware, backdoors, or bots into the victim. DRASP is easily extendable to detect such compromise stemming from driver vulnerability exploitation. Since such attacks are not application targeted, validation of response and application state may not help. Validation policy can be constructed based on system hook profiling ([81] and [36]), process list information, port mapping and etc. to swiftly detect system-wide anomaly based on replica comparison.

5.4 Implementation

I implement the DRASP architecture on Xen hypervisor, with one HVM Domain U as the front stage replica, and the other one as the back stage replica. I configure the hypervisor to emulate heterogeneous devices for the two replicas, i.e., one is with rtl8139 NIC, Gravis ultrasound GF1, Intel Open Host Controller Interface and etc, the other with e1000 NIC, sound blaster 16, Universal Host Controller Interface and etc. I run the same guest operating system and service application (e.g., Apache http server) for
the two replicas. Hence, the only differences between them are when they start booting, heterogeneous device driver modules will be loaded to their kernels to interact with the corresponding devices. I deploy tinyproxy [10] as front tier proxy to implement request replication and response validation.

5.4.1 Reconstructing OS Semantics

In order to provide the state information of the service application process, I implement reconstruction engine to dynamically reconstruct OS semantics. I start from the structure hvm.hw_cpu defined by Xen to emulate the processor for VM, and obtain all the registers there. From the register tr, I locate the kernel stack of the currently running process. At the bottom of kernel stack resides the thread_info structure, which includes a pointer to the task_struct of the corresponding process. Through the task_struct, I can obtain all the information related to this process, such as the structures describing the virtual memory, stack, scheduling status, open files, inter process communications and etc. In order to reconstruct the system hook information, I refer to the sysmap file on each Guest OS, and locate the memory address of system call table and interrupt table. Similarly, with the help of other registers and the sysmap file, I can further identify all the file objects and part of kernel data structures (almost 1000).

5.4.2 Synchronization

I rely on virtual platform underneath to synchronize the two service applications either at the granularity of system call or packet processing. By reconstructing OS semantics of each replica, I can monitor the CPL (in CS) and cr3 register to identify
the execution of the service application. Together with eax register, I determine the legitimate system calls made by the service application. Hence, the state of the service application can be logged whenever one system call issued by it is audited. I also leverage virtual platform to balance CPU allocation time for each replica by adjusting the scheduling parameters. This is to ensure that the two service applications are loosely synchronized in the progress. To implement the packet processing synchronization, I insert a small piece of synchronization code into the NIC emulation module of Xen for the two VMs. Whenever one packet from the service application is sent, the synchronization code triggers the state logging functionality to record the state information of the service application process. The system call synchronization and packet processing synchronization can be dynamically switched through the control console.

5.4.3 Validation of Responses and States

The response comparison is implemented in the front tier tinyproxy. The responses from the two replicas are first matched based on the destination IP, sequence number, and etc. in the packet header. The matched responses are compared based on the response validation policies, without caring the white spaces, comments, and etc. The logs taken by the two replicas are first reconstructed with OS semantics, and then rearranged by semantic blocks. I perform bitwise comparison of each semantic block for validation. For system call synchronization log validation, any mismatch deserves system administrator to carefully examine the semantics of the mismatch, thus taking corresponding actions, e.g., shutting down the front-stage replica and switching to the back-stage replica. In contrast, for packet processing synchronization log validation, only the mismatch in
data records of dependable files causes such alarm. This is because the variance of the memory state, or metadata could exist when synchronizing applications at the loose packet processing granularity.

5.5 Summary

In this chapter, I present a novel approach, DRASP, to prevent exploited drivers from compromising service applications for commercial profit, and preserve as much accumulated state, continuity and correctness as possible. DRASP relies on virtual platform underneath to emulate heterogeneous devices for guest operating system, and correspondingly produce driver-diverse replicas. Thus, one driver vulnerability exploit can at most succeed on one replica, with the other survival. Service applications identically replicated on the two replicas are synchronized either at the system call or packer processing granularity. By validating the responses and states of the two applications on the two replicas, DRASP approach can swiftly detect the intrusion to the service application through the driver vulnerability exploit.

Meanwhile, since the other replica is immune from such compromise, the service workload can be efficiently transferred to the application running on that replica to preserve continuity and accumulated state. DRASP architecture is totally compatible with most existing fault tolerant system environment, and pretty easy to be deployed for enterprise servers. In addition, DRASP does not require any modification to existing drivers or operating systems, thus rendering the direct reuse of most device drivers on the commodity operating systems. The evaluation demonstrates that DRASP can
achieve on-the-fly driver-orientated intrusion response and ensure the correctness of the applications’ execution.
In this chapter, I analyze the effectiveness of PEDA, Heter-device, and DRASP at (1) comprehensively analyzing intrusion harm of production workload systems and (2) swiftly detecting driver code exploit to manipulate service applications. I also evaluate the performance cost of using PEDA, Heter-device, and DRASP with common service applications.

6.1 PEDA

I evaluate PEDA system in terms of logging efficiency and intrusion analysis comprehensiveness. The Xen hypervisor and the backend system are running separately on two Lenovo Thinkstation Tower D10 machines, each with Dual Intel(R) PRO/1000 NICs. I run CentOS 5.2 (kernel version 2.6.18) as coordination platform on backend system and as Xen Domain 0 on Xen hypervisor. Both Xen-HVM Domain U and QEMU Guest OS are installed with Fedora 6 (kernel version 2.6.20). Both QEMU and Xen-HVM are preconfigured to have the same amount of memory (1 GB), the same NAT based networking, and the same kind of devices emulation for Guest OS or Domain U. I also deploy a router, connecting these two machines with Gigabit Ethernet and responsible for packets direction to them. Outside the router, I have two Dell OptiPlex
745 machines used as client request simulation and attacker platforms, with Gigabit Ethernet connection to the router.

### 6.1.1 Logging Efficiency

I install *Apache 1.3.9* on Xen-HVM Domain U as a *http* server. I evaluate the runtime overhead introduced by non-deterministic events plus system call logging, and measure the server performance degradation during the pre-checkpointing phase and service downtime during the stop-and-copy phase.

**Runtime Overhead** I compare the performances of *apache* running on the native Xen-HVM Domain U and on the Xen-HVM Domain U with the instrumentation. On the Dell machines, I simulate clients sending continuous requests over concurrent connections to fetch an 8 KB file. Figure 6.1 shows that the native Xen-HVM Domain U achieves the throughput of almost 800 Mbps, which is considered as baseline performance in the evaluation. It did not reach up to 2000 Mbps (Two Gigabit NICs on the machine) probably due to the impact of network I/O virtualization introduced by Xen.

![Baseline Throughput](image)
The baseline performance can be improved by optimization proposed in [59] to achieve the near-native throughput. Figure 6.2 shows the apache throughput with the packets redirected to the logging backend system (from the edge router). Compared with Figure 6.1, I can see that the logging achieves about 95% baseline performance, which the 5% runtime overhead is mainly caused by the system call logging.

Downtime and Performance Degradation Caused by Checkpointing In order to simulate the Amazon-style server during 2 am-5 am, I reduce user requests to apache server by 90%. Figure 6.3 shows that the server throughput decreases correspondingly. At the time (about 5 seconds from the very beginning) when I issued the command `xm checkpoint`, the server throughput drops by 24%, which lasts for almost 4 seconds. This performance degradation can be explained by the fact that I introduced a pre-checkpointing phase, during which the whole virtual disk and physical memory are recorded. Following is the service downtime (no throughput) during the “stop-and-copy” phase, which only lasts for less than 0.4 second. To take a checkpoint of a running system, the service downtime cannot be eliminated, because it is generally impossible.
to take a consistent whole system checkpoint considering the fact that a running system may do “write” operations to either memory or disk. I reduce this kind of service downtime by introducing a pre-checkpointing phase, which takes the large amount of copying workload from the “pausing” phase. Note that in Figure 6.3, the short pulse immediately after the downtime is likely to be caused by the accumulated requests from the performance degradation period.

6.1.2 Intrusion Analysis Comprehensiveness

To show the comprehensiveness and precision of the intrusion analysis, I conduct two case studies of real-life intrusion. For the page limit, I only show the detailed results of the first case in terms of fine-grained intrusion root identification and dynamic taint tracking. For the second case, I focus on showing the advance of PEDAS system over previous system-call-level intrusion analysis.
6.1.2.1 Case Study 1

The attack scenario of Case 1 is as follows. The attacker first logs into the server system by `ssh` using an unprivileged user account. Then he downloads a Linux NULL pointer dereference exploit and launches the attack [7] to gain root privilege. Afterwards, he mails back, examines and modifies the `syslog.conf` file to let the system logs be sent to his email account. Finally, he deletes all files under the `/var/log/` directory to hide his intrusion “footprint”.

**Fine-grained Intrusion Root Identification** I assume that the IDS detects the maliciously modified file `syslog.conf` and the missing files under the directory `/var/log/`. These intrusion symptoms are notified to the PEDA system. Afterwards, I start the intrusion root identification from the detected intrusion symptoms, and trace the automatically-generated dependency graph backward. I tailor the intrusion flows at the system object level from the dependency graph, and locate the system-object-level intrusion root `wget`. Figure 6.4 shows the fine-grained intrusion root identification procedure. I audit system calls issued by `wget` to identify the buffers containing the intrusion packet. Finally, I can obtain the disk sectors used to store the intrusion packet, which is taken as fine-grained taint seed for infection diagnosis.

**Infection Diagnosis** In order to show sufficient information regarding the intrusion behaviour, and the details of how the intrusion happens on the server system and what has been infected by the intrusion propagation, I start a whole system dynamic taint tracking from the disk sectors containing the intrusion packet during replay. For the space limitation, Figure 6.5 presents only partial outcome of the infection diagnosis.
The rectangle denotes memory address space or disk sectors on the server system. The ellipse attached to each rectangle includes the OS semantics from the reconstruction engine, including the processes which the address space belongs to, the files which the disk sectors are allocated to, or the dynamic libraries which the memory address space is loaded to. It is sufficient to demonstrate that the cross-layer infection diagnosis features with specific infected memory space, disk sectors, and kernel address space, which are far beyond the system-call-level intrusion tracking. Moreover, the infected memory address information provides the system admin a feasible way to “sweep out” any intrusion harm on the victim system. In addition, I can catch the “blind spot” of system-call-level intrusion analysis. For instance, I can capture how the attacker obtains the root privilege through the intrusion packet.
6.1.2.2 Case Study 2

Case Study 2 is designed to demonstrate the advance of PEDA system over other system-call-level intrusion analysis. The attacker logs into the system by ssh using an unprivileged user account. Then he launches the sendmail local escalation exploit to gain root access. The attacker uses the root shell to download and install the adore rootkit, which replaces several kernel hooks in the system call table with its own implementation. Afterwards, he uses the same root shell to download and install the ARK rootkit, which replaces system binaries (e.g., syslogd, login, ssbd, ls, ps, netstat, and etc.) with backdoored versions. I rely on IDS to detect the modification of system binaries by integrity check. By virtue of the backward system call dependency tracking, all of
PEDA, SHELF [84] and Backtracking [52] can identify ssh as the system-object-level intrusion root. However, neither SHELF nor Backtracking can locate the malicious kernel hook modification by adore rootkit. Instead, they are only capable of diagnosing the intrusion infection of ARK rootkit due to their system call flow auditing. Rather, the fine-grained intrusion root identification of PEDA system can audit the system calls issued by ssh, and identify the disk sectors containing the downloaded rootkits adore and ARK as taint seed. By applying dynamic taint tracking and semantics reconstruction, PEDA is able to capture not only the damage identified by SHELF and Backtracking, but also the intrusion harm of the kernel hooks modification implanted by adore rootkit, such as the replaced sys_write and etc.

6.2 Heter-device

In this section, I present experimental results on Heter-device framework in three aspects. First, I present the comparison results on OS kernel APIs called by different functions of heterogeneous drivers. Second, I show the effectiveness of Heter-device in capturing compromised drivers by two case studies. Last, I evaluate the performance overhead incurred by Heter-device approach. The host OS is Ubuntu 10.10 with kernel version 2.6.35, and both of the two guest operating systems are installed with Fedora 5 (kernel version 2.6.15). I choose qemu-0.12.5 as the virtual machine monitor emulating two virtual platforms: one with Realtek RTL8139 NIC and Gravis ultrasound GF1, the other with Intel 82540EM NIC and sound blaster 16.
6.2.1 Profiling Heterogeneous Drivers

First, I load Heterogeneous NIC drivers e1000 and 8139too on the two replicas running Intel 82540EM NIC and Realtek RTL8139 NIC respectively. The trusted kernel module monitors alloc_netdev function to trace the newly allocated net_device structure for the network card. Then the function pointers in net_device, such as open, stop, hard_start_xmit, etc., are audited during the initialization of NIC drivers to obtain the addresses of these driver functions. The functions implemented by both heterogeneous drivers are correlated as the synchronization entries of the two replicas. I start to audit the control flow transition between OS kernel and NIC driver since the booting of the two replicas. Then I trigger a set of user commands (e.g., ssh, sftp, ping, and etc.) and applications (e.g., Firefox, Filezilla, and etc.), which involve network card operations, to invoke the interaction between OS kernel and NIC driver. Simultaneously, the kernel API calls issued by each synchronized function of heterogeneous drivers are profiled.

Table 6.1 shows the profiling results of heterogeneous drivers e1000 and 8139too. Although implemented by different teams, the same functions of heterogeneous drivers typically invoke a similar set of kernel APIs. In particular, I find that 8139too calls kernel_thread kernel API in open function. Thus, I monitor the forked kernel thread and observe the following kernel APIs invoked during the thread’s lifetime: daemonize, allow_signal, interruptible_sleep_on_timeout, refrigerator, flush_signal, rtnl_lock_interruptible,
rtnl_unlock, and complete_and_exit. Table 6.1 also indicates that previous works will generate lots of false positive when referring to exported functions in System.map as trusted entries from drivers to OS kernel\(^1\).

### 6.2.2 Case Study 1: Kernel Integrity Manipulation

I refer to the implementation of adore-ng kernel rootkit, and integrate its malicious code into the function snd_gf1_stop_voice of gus (for Gravis ultrasound GF1) driver. When users try to turn off the audio, the injected code gets executed to replace the functions of readdir, lookup, and get_info with its own implementation to hide files, processes and ports. The newly generated driver gus is recompiled and loaded into OS kernel. In contrast, driver sb16 (for sound blaster 16) remains unchanged.

During the assessment of drivers gus and sb16, I simulate user’s command to turn off the audio, which is replicated to both replicas. The modification of those static kernel data (function pointers) by driver gus is observed and alarmed, while driver sb16 does not exhibit such behaviour. Then I clear this alarm, let the two replicas run forward, and issue process and file listing commands. I observe that the control flow transition from OS kernel to driver gus code section through unrecognized entry. Afterwards, driver gus calls kernel APIs, i.e., readdir, lookup, and get_info, from its execution context. In contrast, driver sb16 on the other replica is not involved in the process and file listing procedures.

\(^1\)Similar profiling has been performed on heterogeneous sound card drivers gus (for Gravis ultrasound GF1) and sb16 (for sound blaster 16). The profiling results are excluded due to page restriction.
Table 6.1 Kernel APIs called by different functions in e1000 and 8139too. For each synchronization function, the upper box contains the invoked kernel APIs defined in System.map file of guest OS, while the lower box includes the indirected invoked kernel APIs that are called by drivers through the following procedure. Drivers call some other extern kernel functions (not defined in System.map file) by including some .h files, and these functions in turn invoke the indirected kernel APIs identified by us.

<table>
<thead>
<tr>
<th>Synch. Entry</th>
<th>Kernel APIs by e1000</th>
<th>Kernel APIs by 8139too</th>
</tr>
</thead>
<tbody>
<tr>
<td>*open</td>
<td>request_irq, mod_timer, kmalloc, pci_clear_mwi, vmalloc_node</td>
<td>netif_carrier_on, netif_carrier_off, request_irq, spin_unlock_irqrestore, spin_lock_irqsave, kernel_thread</td>
</tr>
<tr>
<td></td>
<td>dma_alloc_coherent, _alloc_skb, _alloc_pages</td>
<td>dma_alloc_coherent</td>
</tr>
<tr>
<td>*stop</td>
<td>free_irq, netif_carrier_off, nmapel, vfree, kfree</td>
<td>free_irq, wait_for_completion, kill_proc, spin_lock_irqsave, spin_unlock_irqrestore</td>
</tr>
<tr>
<td></td>
<td>netpoll_trap, dma_free_coherent, lock_timer_base, list_del, _kfree_skb, local_irq_save, local_irq_restore</td>
<td>netpoll_trap, dma_free_coherent</td>
</tr>
<tr>
<td>*interrupt_handler</td>
<td>spin_lock, spin_unlock, eth_type_trans, netif_rx</td>
<td>spin_lock, spin_unlock</td>
</tr>
<tr>
<td></td>
<td>__alloc_skb, netpoll_trap, kfree_skb, local_irq_save, local_irq_restore</td>
<td>netpoll_trap, local_irq_save, local_irq_restore</td>
</tr>
<tr>
<td>*tx_timeout</td>
<td>schedule_work</td>
<td>spin_lock, spin_lock_irqsave, spin_unlock_irqrestore</td>
</tr>
<tr>
<td></td>
<td>spin_lock, spin_lock_irqsave, __wake_up</td>
<td>spin_unlock_irqrestore</td>
</tr>
<tr>
<td>*do_ioctl</td>
<td>request_irq, spin_unlock_irqrestore, free_irq, spin_lock_irqsave, netif_carrier_off, mod_timer</td>
<td>spin_lock_irq, spin_unlock_irq</td>
</tr>
<tr>
<td></td>
<td>lock_timer_base, list_del, _kfree_skb, local_irq_save, local_irq_restore, netpoll_trap</td>
<td>capable</td>
</tr>
<tr>
<td>*hard_start_xmit</td>
<td>spin_trylock, spin_unlock_irqrestore, pskb_pull_tail, pskb_expand_head</td>
<td>spin_lock_irq, spin_unlock_irq</td>
</tr>
<tr>
<td></td>
<td>local_irq_save, netpoll_trap, local_irq_restore</td>
<td>_kfree_skb, netpoll_trap</td>
</tr>
<tr>
<td>*poll</td>
<td>spin_lock, spin_unlock, disable_irq, netif_carrier_ok</td>
<td>spin_lock, spin_unlock, netif_receive_skb</td>
</tr>
<tr>
<td></td>
<td>local_irq_save, _kfree_skb, local_irq_restore</td>
<td>local_irq_disable, _alloc_skb, local_irq_enable, list_del, local_irq_save, local_irq_restore</td>
</tr>
<tr>
<td>*set_multicast_list</td>
<td></td>
<td>spin_lock_irqsave, spin_unlock_irqrestore</td>
</tr>
<tr>
<td>*get_stats</td>
<td></td>
<td>spin_lock_irqsave, spin_unlock_irqrestore</td>
</tr>
</tbody>
</table>
6.2.3 Case Study 2: Resource Abuse and Confidentiality Tampering

With the kernel privilege of compromised driver, attackers can launch resource
starvation attack to reduce the productivity of the victim systems, or tamper confidential-
tiability by intercepting bypassing data. I simulate resource abuse by inserting malicious
code into the source code of RTL8139 NIC driver. In particular, after `spin_lock` is called
in function `rtl8139_interrupt`, repeated call of `alloc_skb` is issued until kernel memory is
overwhelmed. Then the driver is recompiled and loaded into OS kernel as `8139too` mod-
ule. I repeat a subset of the user commands and applications in Section 6.2.1. During
the assessment, after the synchronization of `interrupt_handler` function entry, `e1000_intr
quickly returns. However, `rtl8139_interrupt` continues running with lots of `alloc_skb` calls
recorded. The verification alarms such anomaly immediately with a pre-defined differ-
ence threshold (200 in the experiment) reached.

Furthermore, I also simulate confidentiality tampering attack by injecting malici-
ous code into the packet transmission function `e1000_xmit_frame` of e1000 NIC driver.
The newly compiled `e1000` module will intercept all the outgoing packets and redirect
them to a remote machine. During the assessment, I replicate Apache `http` servers on
both the two replicas, and simulate continuous client requests to them on another ma-
cine. The verification on the front-tier proxy matches the output packets from the
two replicas. An alarm is signaled when the amount of unmatched packets from the
replica with `e1000` module reaches the pre-defined threshold (20 in the experiment) in
two minutes.
Table 6.2 Runtime Performance of Different Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Key Kernel Data</th>
<th>Whole Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMbench</td>
<td>1.2021</td>
<td>1.2444</td>
</tr>
<tr>
<td>Apache Benchmark</td>
<td>1.4420</td>
<td>2.8273</td>
</tr>
<tr>
<td>Interbench</td>
<td>1.3356</td>
<td>1.4160</td>
</tr>
<tr>
<td>Kernel Decompression</td>
<td>1.1663</td>
<td>1.2262</td>
</tr>
</tbody>
</table>

6.2.4 Performance Evaluation

The runtime overhead of Heter-device highly depends on the amount of key kernel data that needs to be verified. Table 6.2 shows the performance (the ratio of Heter-device execution and QEMU native execution) of Heter-device architecture based on several benchmarks. By key kernel data protection, I only verify static key kernel data, including system call table, IDT, root symbols in System.map files. In contrast, by whole kernel protection, the entire kernel address space is verified by Heter-device during driver assessment. During each round of evaluation, both the heterogeneous NIC and sound card drivers are verified for fine-grained control flow transition.

I use LMbench to evaluate the pipe bandwidth, and also evaluate the time consumed to decompress Linux kernel 3.0 as shown in Table 6.2. Since both of them involve little interaction with either NIC or sound card, the pipe bandwidth and CPU capacity are mostly retained. I run Apache Benchmark to evaluate the network performance, and Interbench to evaluate the audio performance. Table 6.2 demonstrates that network throughput drops more significantly than audio performance. I think the main reason is that NIC drivers interact more frequently with OS kernel during packet transmission than sound card drivers do during audio playing.
6.3 DRASP

I report experiments on DRASP architecture by showing the effectiveness of DRASP in the driver-vulnerability-orientated service-application-target intrusion detection, and the efficiency of DRASP. I run Xen hypervisor 3.3.0 as virtual platform, and configure it to emulate rtl8139 NIC plus Gravis ultrasound card for one DomU, and e1000 NIC plus sound blaster card for the other. DRASP requires device emulation support from virtual platform, which should provide different device emulation options for the same functionality, e.g., e1000 NIC and rtl8139 NIC. Currently, such options are limited, and for some types of devices, there is even no option provided, i.e., video card driver and etc. Although this restricts the evaluation of DRASP only to the NIC and sound card drivers, I think it is sufficient to demonstrate the proof-of-concept of DRASP architecture. As the renaissance of virtualization, I believe that such device emulation options will bloom in the near future. Both of the two guest operating systems on DomU are installed with Fedora 5 (kernel version 2.6.15). I run Apache http server on the two guest OSes as service application. I deploy the modified tinyproxy as the front tier proxy for request replication and response validation.

6.3.1 Capacity in Driver Vulnerability Orientated Intrusion Detection

Response Validation Based Detection. Attackers can set the existing driver function pointer to arbitrary code either implemented by the attackers or of the kernel’s. As a result, the attackers can execute any code with kernel privilege. I simulate such an attack by inserting a buffer overflow bug into packet processing function of the rtl8139
NIC driver 8139too, while the e1000 NIC driver e1000 does not have such a bug at the corresponding function. The attacker exploits the bug in the driver 8139too, and then redirects the hard_start_xmit function pointer to attacker’s injected code. Thus, whenever any packet is transmitted on the victim (the front stage) replica, the control flow will be redirected to the driver’s injected code. The injected code simulates the manipulation of some payload data of each packet for commercial profit. Since the driver e1000 is not inserted with such a bug, the exploit shall not succeed and the function pointer hard_start_xmit cannot be tampered on the back stage replica. Thus, the response from back stage replica is the correct one without manipulation. As a result, the response validator on the front tier proxy detects such mismatch among the responses made by the two service applications, and confirms a compromise of the front stage replica.

**State Validation Based Detection.** With the kernel privilege of the exploited driver, attackers can directly tamper with metadata or kernel data structures related to the service application process for commercial benefits. I simulate such an attack by inserting the same buffer overflow bug as above into the rtl8139 NIC driver 8139too, while the e1000 NIC driver e1000 does not have such a bug at the corresponding function. The attacker exploits the bug in the driver 8139too on the front stage replica, and manipulates some contents of the dependable files of Apache http server (.html file). Since the driver e1000 is not inserted with such a bug, the exploit shall not succeed and the corresponding dependable file of Apache http process will not be tampered on the back stage replica. As a result, the state validator detects such mismatch by comparing
the dependable files logged by the two service applications, and confirms a compromise of the front stage replica.

6.3.2 Efficiency

The runtime overhead of DRASP depends on the granularity of state validation that is applied. In order to evaluate the runtime overhead introduced by state and response validation, I compare the performances of *Apache http server 1.3.9* running on the native Xen-HVM Domain U and on the Xen-HVM Domain U with the instrumentation (including request replication, response and state validation functionality). On another machine, I simulate clients sending continuous requests over concurrent connections to fetch an 8 KB file. Figure 6.6 shows that the native Xen-HVM Domain U achieves the throughput of almost 810 Mbps, which is considered as baseline performance in the evaluation. Figure 6.7 and 6.8 demonstrate the throughput of *Apache http server* with

---

2For such dependable file manipulation, both the fine-grained and coarse-grained time slice state validation can detect the compromise.
the instrumentation. Figure 6.7 is with coarse-grained time slice state validation and achieves around 790 Mbps throughput, while Figure 6.8 is with fine-grained time slice state validation and achieves around 740 Mbps throughput. Compared with Figure 6.6, the overhead introduced are 2.47% and 8.64% respectively. I believe that such overhead is reasonable and affordable for most service applications.

6.3.3 Continuity, Accumulated State and Correctness Preservation

I run Apache http server 1.3.9 on the instrumentation platform, and ab [1] on another machine to evaluate the continuity, accumulated state and correctness preservation functionality of DRASP. I configure ab to simulate 10 concurrent connections sending requests to Apache http server for 600 seconds. During this period, tinyproxy continues to switch between the front stage replica and the back stage replica with the frequency of once per second. I repeat the experiment for 10 times, and do not find any missing/false response for all the requests. Hence, I can conclude that once the front stage replica
Fig. 6.8 Fine-grained Throughput

is compromised, the service will continue without any interruption by transferring the workload to the back stage replica.
Chapter 7

Conclusion

In this chapter, I will discuss the limitation and future work of PEDA, Heter-device, and DRASP systems, followed by conclusion.

7.1 Discussion and Future Work

First, the automatic intrusion backtracking is not 100% accurate, especially at the granularity of memory cell or disk sector. PEDA system relies on intrusion backtracking to locate the fine-grained intrusion root, which in turn is provided as taint seed to infection analyzer. To reduce the false positive on PEDA’s intrusion harm analysis result, the intrusion backtracking of PEDA involves some human interference to accurately locate the fine-grained intrusion root. Second, to replay the execution of a busy server with significantly high workload, the amount of non-deterministic events to be recorded might be huge. In this case, it may not be feasible for PEDA to store a history of events that is much longer than the expected intrusion detection delay. Thus, if the intrusion is detected much later than its occurrence, the first run compromised execution cannot be completely replayed due to the removal of long time ago non-deterministic event logs.

Second, currently QEMU supports limited number of emulated devices. Most existing device emulation modules in virtual machines such as virtualbox, Xen and KVM are all based on QEMU. Hence, assessing other drivers, e.g., keyboard, mouse and etc.,
is impossible right now on Heter-device architecture. Aware of the design, attackers can
craft malicious drivers only for those devices that have not been emulated by QEMU.
I suggest that system engineers consider the devices that can be emulated by QEMU
right now, thus facilitating the driver assessment of Heter-device architecture. As the
renaissance of virtualization, I hope that such device emulation options will bloom in the
near future, making Heter-device more general and practical. Similar issue also applies
to DRASP.

Third, there exist some counter-attacks to Heter-device and DRASP architecture.
There is always the possibility that the malicious code is not detected because the en-
environment or the workload did not trigger it. Furthermore, transparently translating
instructions [33] and faithfully emulating hardware devices are challenging tasks. For
instance, QEMU can be detected by various methods as discussed in [44]. Aware of the
design, attackers can craft malicious code that first examines whether it runs on emulated
platforms. If so, the malicious code will “keep silent” to avoid being detected or profiled.
Otherwise, it will compromise the victim system. Hence, the compromised drivers with
“split-personality” can generally evade the auditing and verification of Heter-device and
DRASP.

Fourth, existing Heter-device approach involves manual intervention during the
driver assessment. For instance, key kernel data to be recorded and verified should be
provided by system engineers in advance, though I also offer a candidate list. Moreover,
the verification procedure (i.e., control flow transition verification) requires system en-
gineers to investigate the variance to reduce false positive. Furthermore, such manual
inspection can also help to determine which driver is compromised, since the two repli-
cas serve as reference model for each other rather than always treating one as golden
standard. My future work is set to comprehensively profile the driver’s behaviour, thus
improving the automation by providing more general key kernel data verification list and
enforcing more detailed verification policies.

Fifth, if the attacker knows the DRASP architecture in detail, i.e., the drivers
of both the front stage replica and the back stage replica, he can probably compromise
DRASP by sending two continuous attacks, the first one targeting at the driver on
the back stage replica and the following one at the front stage replica. The purpose
of the two attacks are the same, compromising the same kernel data structure. As a
result, the state validator module cannot detect the intrusion due to the non-existence
of verification violation evidence. However, since all the responses are provided by the
front stage replica, it is almost impossible for an outside attacker to know exactly the
driver vulnerabilities on the “isolated” back stage replica. The attacker can launch
brute force attack to explore the driver bugs on the back stage replica, but lots of those
suspicious exploit packets will definitely draw the system admin’s attention.

Sixth, although Heter-device architecture is totally compatible with most ex-
isting fault tolerant systems, deploying Heter-device approach as a real-time compro-
mised driver detection system requires performance boost. Due to performance overhead
(largely incurred by QEMU), Heter-device is currently deployed as honeypot to assess
drivers before they are put into use in server systems. Hence, it should be my future work
to facilitate hardware support, such as Intel VT or EPT techniques into the Heter-device
architecture to feasibly detect compromised drivers in responsive server environment.
Last, existing implementation of PEDA, Heter-device, and DRASP is based on Linux, so I plan to migrate them to Windows systems. Since the design of Heter-device, DRASP and PEDA is totally out of the operating system scope, the implementation is on the virtual machine monitor, without any modification to the operating system. Hence, I believe it is quite feasible and realistic to migrate the PEDA, Heter-device, and DRASP implementations to Windows operating systems.

7.2 Conclusion

In this dissertation, I propose a set of automatic security analysis mechanisms to help commodity systems be resilient to vulnerability exploitation. These mechanisms finally help server systems automatically preserve service continuity and correct execution upon vulnerability exploitation.

First, in order to enable swift and safe recovery of the protected server systems after attacks, PEDA is deployed to achieve post mortem fine-grained intrusion analysis. PEDA helps security technicians swiftly identify the fine-grained intrusion root “breakin” to the server and precisely pinpoint the infection propagation throughout the server. PEDA effectively decouples the analysis work off the online server execution by novelty integrating the backward system call dependency tracking and forward fine-grained taint analysis. The proposed heterogeneous VM migration significantly reduces the runtime overhead of online server execution. That is, the decoupled intrusion analysis runs on binary translation based virtual machine, while the server production
workload runs on much faster hardware-assisted virtual machine. Evaluation demonstrates PEDA’s advance over existing intrusion analysis systems in terms of efficiency and comprehensiveness.

Second, I introduce a novel diversity based honeypot, Heter-device, to assess the trustworthiness of drivers from multiple aspects, including kernel integrity manipulation, resource starvation and confidentiality tampering. Heter-device relies on virtual platforms to emulate heterogeneous devices for guest operating systems, and correspondingly produce driver-diverse replicas. The diverse replicas are deployed as honeypot to audit and verify the heterogeneous drivers’ execution by placing synchronization and monitoring “sensors”. I also propose automatic address-alias correlation, a subset of kernel data for default integrity protection, and a set of policies to defeat resource abuse and confidentiality tampering. The case studies show that Heter-device can capture various kernel integrity manipulation, resource abuse, and confidentiality tampering launched from compromised drivers, thus delivering the trustworthy drivers after assessment.

Last, DRASP reuses Heter-device architecture to produce driver-diverse replicas. Thus, one driver vulnerability exploit can at most succeed on one replica, with the other survival. Service applications identically replicated on the two replicas are synchronized either at the system call or packer processing granularity. By validating the responses and states of the two applications on the two replicas, DRASP approach can swiftly detect the intrusion to the service application through the driver vulnerability exploit. Meanwhile, since the other replica is immune from such compromise, the service workload can be efficiently transferred to the application running on that replica to preserve continuity and accumulated state. Evaluation demonstrates that DRASP can achieve
on-the-fly driver-orientated intrusion response and ensure the correctness of the applica-
tions’ execution. I believe that the comprehensive intrusion analysis functionality of
PEDA, the driver trustworthiness analysis of Heter-device, and the operate-through in-
trusion response of DRASP would have a profound impact on the anomaly prevention,
detection, analysis, and recovery of both academy and industry.
References

http://httpd.apache.org/docs/2.0/programs/ab.html.


Vita

Shengzhi Zhang

Education


Ph.D in Computer Science & Engineering

Area of Specialization: System Security

*(Tongji University)* (Shanghai) 2002–2006

B.S. in Electrical Engineering & Automation

Area of Specialization: Power Systems and Communication Systems

Research Experience

**Doctoral Research** The Pennsylvania State University 2007–2012

Dissertation Advisor: Prof. Peng Liu

**Graduate Research** Inha University, South Korea 2006–2007

Research Advisor: Prof. Sang-Jo Yoo