CERIAS Tech Report 2011-03

Data-centric Approaches to Kernel Malware Defense

by Junghwan Rhee
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Junghwan Rhee

Entitled
Data-Centric Approaches to Kernel Malware Defense

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Dongyan Xu
Chair

Eugene H. Spafford

Xiangyu Zhang

Sonia Fahmy

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Dongyan Xu

Approved by: Sunil Prabhakar / William J. Gorman 06/21/2011
Head of the Graduate Program Date

Graduate School Form 20
(Revised 9/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

Data-Centric Approaches to Kernel Malware Defense

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Junghwan Rhee

Printed Name and Signature of Candidate

06/13/2011

Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

DATA-CENTRIC APPROACHES TO KERNEL MALWARE DEFENSE

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Junghwan Rhee

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2011

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Professor Dongyan Xu who guided and supported me throughout my study. He broadened my perspectives in research and provided invaluable suggestions to overcome the challenges that I faced. His sincere consideration of my family life and his knowledgeable advice during my job search were greatly appreciated as well.

I also greatly appreciated the assistance of Professors Eugene Spafford, Xiangyu Zhang, and Sonia Fahmy, who all served on my committee. Professor Spafford challenged me to go beyond engineering to focus on fundamentals and science in my research. Professor Zhang provided inspiration to me as I worked to solve research problems in the projects on which we worked together. Professor Fahmy helped me to clarify, organize, and improve my research, which was an essential step to preparing for my dissertation.

I was fortunate to work with brilliant colleagues, who are current members and alumni of the Lab Friends: Ryan Riley, Xuxian Jiang, Zhiqiang Lin, Paul Ruth, Ardalan Kangarlou, Sahan Gamage, and Zhongshu Gu. They all have my sincere gratitude for their help and friendship.

Last but not the least, I could not have finished my study without the enduring support of my family. I deeply appreciated the love and support of my wife, Chungah Seo, during my many years of graduate study. Our two children, William and Lauren, were delightful addition to our family during my study at Purdue, and they too have endured some of our family struggles to finish. I am very thankful to our parents: Kyuyoung Rhee, Younhee Rhee, Soontaek Seo, and Kyungsoon Yang, who have provided valuable and enduring support to our family in many ways throughout my study.

TABLE OF CONTENTS

Page

LIST OF TABLES vi

LIST OF FIGURES vii

ABSTRACT ix

1 INTRODUCTION 1
- 1.1 Problem Statement 1
- 1.2 Statement of Thesis 3
- 1.3 Contributions 4
- 1.4 Terminology 5
- 1.5 Assumptions 6
- 1.6 Organization 7

2 RELATED WORK IN MALWARE ATTACKS AND DEFENSE 8
- 2.1 Code Injection Attacks and Code Integrity-based Approaches 8
- 2.2 Non-Code Injection Attacks and Defense Approaches 10
- 2.3 Malware Defense Based on Code Behavior Signatures 11
- 2.4 Malware Defense Based on Data Signatures 12
- 2.5 Kernel Integrity Checking based on Kernel Memory Mapping 13
  - 2.5.1 Static Type-projection Mapping 15
  - 2.5.2 Dynamic Type-projection Mapping 16
- 2.6 Kernel Rootkit Profilers 17

3 DATA-CENTRIC APPROACHES TO KERNEL MALWARE DEFENSE 18
- 3.1 Code-centric Approaches versus Data-centric Approaches 18
  - 3.1.1 Code-centric Malware Defense Approaches 18
  - 3.1.2 Data-centric Malware Defense Approaches 20
- 3.2 Design 21
- 3.3 Objectives 21
- 3.4 Types of Monitoring: Internal versus External 22
  - 3.4.1 Internal Monitors 23
  - 3.4.2 External Monitors 23
- 3.5 General Data Object Properties and a Model for Kernel Memory Access Patterns 24
  - 3.5.1 General Data Object Properties 24
  - 3.5.2 A Model for Kernel Memory Access Patterns 27
  - 3.5.3 A Conceptual View of General Data Object Properties . . . 28

Page

4 KERNEL MALWARE DETECTION AND ANALYSIS WITH UN-TAMPERED AND TEMPORAL VIEWS OF KERNEL OBJECTS 30
4.1 Introduction 30
4.2 Design of LiveDM 32
4.2.1 Allocation-driven Mapping Scheme 33
4.2.2 Techniques of LiveDM 35
4.3 Implementation 38
4.4 Evaluation 40
4.4.1 Runtime Tracking of Dynamic Kernel Objects 40
4.4.2 Identifying Dynamic Kernel Objects 42
4.4.3 Code Patterns Casting Objects from Generic Types to Specific Types 43
4.4.4 Performance of Allocation-driven Mapping 44
4.5 Hidden Kernel Object Detector 45
4.5.1 Leveraging the Un-tampered View 46
4.5.2 Detecting DKOM Data Hiding Attacks 47
4.6 Temporal Kernel Malware Analysis 51
4.6.1 Systematic Visualization of Malware Influence via Dynamic Kernel Memory 52
4.6.2 Selecting Semantically Relevant Kernel Behavior Using Data Lifetime 54
4.6.3 Case (1): Privilege Escalation Using Direct Memory Manipulation 55
4.6.4 Case (2): Dynamic Kernel Object Hooking 56
4.7 Summary 58

5 CHARACTERIZING KERNEL MALWARE BEHAVIOR WITH KERNEL DATA ACCESS PATTERNS 59
5.1 Introduction 59
5.2 Design of DataGene 61
5.2.1 Data Behavior Profile Approach 63
5.2.2 Generating a Data Behavior Profile 65
5.2.3 Characterizing Malware Data Behavior 67
5.3 Implementation 75
5.4 Evaluation 76
5.4.1 Malware Signature Generation 79
5.4.2 False Positive Analysis 80
5.4.3 Detecting Rootkits using Data Behavior Signatures 83
5.4.4 Similarities among Data Behavior Signatures 84
5.4.5 Extracting Common Data Behavior Elements 87
5.4.6 Monitoring Performance 90
5.5 Summary 92

Page

6 CONCLUSIONS 93
6.1 Discussion and Limitations 94
6.2 Conclusions 98
6.3 Future Work 100
LIST OF REFERENCES 101
VITA 109

LIST OF TABLES

Table | Page

4.1 A list of core dynamic kernel objects and the source code elements used to derive their data types in static analysis. (OS: Debian Sarge). . . . 42

4.2 DKOM data hiding rootkit attacks that are automatically detected by comparing LiveDM-generated view (L) and kernel memory view (S). . 48

4.3 The list of kernel objects manipulated by `adore-ng` rootkit. (OS: Redhat 8). 53

5.1 Details of data behavior profiles for benign kernel runs. CL: # of classes, RS: # of read sites, WS: # of write sites. 77

5.2 Details of malicious and benign kernel DBPs (D) and generated signatures (S). CL: # of classes, RS: # of read sites, WS: # of write sites. . . . 78

5.3 Details of the signatures for `adore` 0.38, `SucKIT`, and `modhide` rootkits. CL: # of classes, RS: # of read sites, RD: # of number of read data behavior elements, WS: # of write sites, WD: # of write data behavior elements. 78

5.4 Benign kernel runs tested for false positives. A: $S_{adore0.38}$, S: S_{SucKIT}, M: $S_{modhide}$. CL: # of classes, RS: # of read sites, WS: # of write sites. . 81

5.5 The number of matched data behavior elements between three rootkit signatures and the kernel runs with 16 kernel rootkits (average of 5 runs). (`AD1`: `adore` 0.38, `AD2`: `adore` 0.53, `AD3`: `adore-ng` 1.56, `FL`: `fuuld`, `HL`: `hide_lkm`, `SK`: `SucKIT`, `ST`: `superkit`, `LF`: `linuxfu`, `CL`: `cleaner`, `MH`: `modhide`, `MH1`: `modhide1`) 82

5.6 The number of common data behavior elements in the combination of rootkit signatures. (`AD1`: `adore` 0.38, `AD2`: `adore` 0.53, `AD3`: `adore-ng` 1.56, `FL`: `fuuld`, `HL`: `hide_lkm`, `SK`: `SucKIT`, `ST`: `superkit`, `LF`: `linuxfu`, `CL`: `cleaner`, `MH`: `modhide`, `MH1`: `modhide1`) 85

5.7 Top common data behavior elements among the signatures of 16 rootkits. (`AD1`: `adore` 0.38, `AD2`: `adore` 0.53, `AD3`: `adore-ng` 1.56, `FL`: `fuuld`, `HL`: `hide_lkm`, `SK`: `SucKIT`, `ST`: `superkit`, `LF`: `linuxfu`, `CL`: `cleaner`, `MH`: `modhide`, `MH1`: `modhide1`) 88

5.8 Configuration of benchmarks 90

LIST OF FIGURES

Figure Page

2.1 Illustration of type-projection mapping 14

2.2 Data hiding attack via pointer manipulation 14

3.1 Code-centric malware defense approaches 19

3.2 Data-centric malware defense approaches 19

3.3 Design of data-centric malware defense architecture 21

3.4 A lifetime of a dynamic data object 25

3.5 Identification of dynamic data objects 26

3.6 A model for memory access patterns 27

3.7 Conceptual views of the general data object properties 29

4.1 Overview of LiveDM. 33

4.2 A high level view of static code analysis 37

4.3 Static code analysis. C: a call site, A: an assignment, D: a variable declaration, T: a type definition, R: a return, and F: a function declaration. 37

4.4 The usage of dynamic kernel objects during the booting stage (OS: Redhat 8). 40

4.5 The usage of dynamic kernel objects during the booting stage (OS: Debian Sarge). 41

4.6 LiveDM identifies kernel objects and generates a kernel object map at runtime.(OS: Redhat 8) 41

4.7 Performance of LiveDM for Linux 2.4 (OS: Redhat 8) 44

4.8 Performance of LiveDM for Linux 2.6 (OS: Debian Sarge) 45

4.9 Illustration of the kernel module hiding attack by `cleaner` rootkit. Note that the choice of t_1, t_2, and t_3 is for the convenience of showing data status and irrelevant to the detection. This attack is detected based on the difference between L and S. 46

Figure Page

4.10 LiveDM detects process hiding rootkit attacks and pinpoints hidden processes. 50

4.11 LiveDM detects kernel driver hiding rootkit attacks and pinpoints hidden drivers. 51

4.12 Kernel control flow (top) and the usage of dynamic memory (below) at the addresses of T_3 (Case (1)) and P_1 (Case (2)) manipulated by the `adore-ng` rootkit. Time is in billions of kernel instructions. 53

4.13 Kernel data view before and after the `adore-ng` rootkit attack. 55

4.14 Kernel control flow view before and after the `adore-ng` rootkit attack. 57

5.1 Overview of DataGene. 62

5.2 An example of kernel code in benign and malicious kernel runs. 64

5.3 Aggregating memory accesses on dynamic kernel objects regarding their classes (allocation sites) c_1 and c_2. 66

5.4 A diagram of memory access patterns (DBEs). A: a set of frequently observed DBEs in benign kernel execution, B: a set of DBEs for benign kernel runs, M: a set of DBEs for malicious runs, S: a set of DBEs specific to malware attacks, F: a set of potential false positives DBEs. 67

5.5 Controlling kernel execution instances in the signature generation stage to reduce F. The descriptions for notations are shared with Figure 5.4. . 68

5.6 Kernel execution instances in the detection stage. B': a set of DBEs for benign runs in the detection stage, B'': a set of DBEs for benign runs with false positives, M': a set of DBEs for malicious runs in the detection stage. Other notations are shared with Figure 5.4 and 5.5. 69

5.7 A procedure for signature generation and matching 70

5.8 Using a single kernel run for both of benign and malware memory access patterns 72

5.9 Similarities among the data behavior of rootkits. Types of arrows ($|I|$: # of the matched elements): thin solid ($0 < |I| < 5$), thick dashed ($5 <= |I| < 25$), and thick solid ($|I| >= 25$). 86

5.10 Performance comparison of unmodified QEMU, LiveDM, and DataGene (OS: Redhat 8) 91

ABSTRACT

Rhee, Junghwan Ph.D., Purdue University, August 2011. Data-Centric Approaches to Kernel Malware Defense. Major Professor: Dongyan Xu.

An operating system kernel is the core of system software which is responsible for the integrity and operations of a conventional computer system. Authors of malicious software (malware) have been continuously exploring various attack vectors to tamper with the kernel. Traditional malware detection approaches have focused on the code-centric aspects of malicious programs, such as the injection of unauthorized code or the control flow patterns of malware programs. However, in response to these malware detection strategies, modern malware is employing advanced techniques such as reusing existing code or obfuscating malware code to circumvent detection.

In this dissertation, we offer a new perspective to malware detection that is different from the code-centric approaches. We propose the data-centric malware defense architecture (DMDA), which models and detects malware behavior by using the properties of the kernel data objects targeted during malware attacks. This architecture employs external monitoring wherein the monitor resides outside the monitored kernel to ensure tamper-resistance. It consists of two core system components that enable inspection of the kernel data properties.

First, an external monitor has a challenging task in identifying the data object information of the monitored kernel. We designed a runtime kernel object mapping system which has two novel characteristics: (1) an un-tampered view of data objects resistant to memory manipulation and (2) a temporal view capturing the allocation context of dynamic memory. We demonstrate the effectiveness of these views by detecting a class of malware that hides dynamic data objects. Also, we present our analysis of malware attack behavior targeting dynamic kernel objects.

Second, in addition to the mapping of kernel objects, we present a new kernel malware characterization approach based on kernel memory access patterns. This approach generates signatures of malware by extracting recurring data access patterns specific to malware attacks. Moreover, each memory pattern in the signature represents abstract data behavior; therefore, it can expose common data behavior among malware variants. Our experiments demonstrate the effectiveness of these signatures in the detection of not only malware with signatures but also malware variants that share memory access patterns.

Our results utilizing these approaches in the defense against kernel rootkits demonstrate that the DMDA can be an effective solution that complements code-centric approaches in kernel malware defense.

1 INTRODUCTION

1.1 Problem Statement

An operating system (OS) kernel is the core of system software that is responsible for the integrity and operations of a conventional computer system. It has been targeted by malicious software (malware) that operates in kernel mode to implement advanced stealthy features, such as backdoors or hidden services, that can elude user-level anti-malware programs.

Malware tampers with program execution and achieves the attacker's malicious goals with a variety of techniques. Many traditional malicious programs use code injection attacks (e.g., buffer overflows and format string bugs), which inject unauthorized code into the memory and executes malware functions. Various kinds of malware, such as computer worms, viruses, exploits, and rootkits, have been using this technique to execute malicious logic [1–3]. Many intrusion detection approaches have been proposed to detect or prevent this type of malware attack [4–11].

In response to these malware defense approaches, malware writers have crafted advanced attack vectors that avoid explicit injection of malicious code to elude such detection approaches. Return-to-libc attacks [12, 13], return-oriented programming [14–16], and jump-oriented programming [17–21] use a combination of existing code pieces to compose malicious logic. Also, raw memory devices [22], third-party kernel driver code, and program bugs [23–25] provide other vectors to reuse legitimate or vulnerable code which are a part of programs for malware attacks.

Another group of defense approaches has been using the sequence of malware code to detect malware [26–30]. These approaches use malware signatures composed of malware code sequences, such as instruction sequences or system call patterns, to match malware behavior. However, in response to them, malware began to employ

techniques to vary malware code execution patterns. Several papers have presented code obfuscation [31–34] and code emulation [35] techniques, which can confuse malware detectors and avoid detection.

These arms-races observed between malware and malware detectors center around the *properties of malicious code*: injection of code and the causal sequences of malicious code patterns. Both techniques use primarily code information, ignoring the identification and properties of the accessed data objects.

In general, computer programs are structured as *code* and *data*. Therefore, malicious attacks are seen as the manipulation of the code and/or data objects of the program under attack. *Code* has been a popular target of attacks, and thus it has been intensively studied by existing malware detection approaches. In contrast, there has been little focus to date on the *data* in malware defense research.

To address the challenges of relying on only code in malware defense, we propose new approaches based on the *properties of data objects* that are targeted in malware attacks. These approaches do not require the detection of the injected code or the specific sequence of malicious code. Therefore, they are not directly subject to attacks targeting the approaches based on code properties.

These approaches, however, have unique challenges in monitoring data objects: the dynamic status of data objects and the difficulty of determining their integrity. For instance, many data objects have readable-and-writable content and the locations of dynamic objects are assigned at runtime [36]. A monitor observing data objects should have a higher level privilege than the monitored program to reliably obtain its data memory status. Monitoring kernel data objects is challenging because, in a conventional computing environment, an OS kernel directly interacts with the hardware, thereby lacking a layer below it on which to build a monitor.

1.2 Statement of Thesis

In this dissertation, we present a novel scheme that addresses these challenges and enables OS kernel malware detection approaches based on kernel data properties. The monitoring system should be designed in a way that cannot directly be altered by potentially malicious code; therefore, we use an external monitor to observe the target OS kernel. An external monitor has a challenging task in identifying the data object information of the monitored kernel, which is known as a *semantic gap* [37]. Such information should be reconstructed externally in the monitor. We propose the data-centric malware defense architecture (DMDA) which uses data object properties to detect kernel malware, which consists of two main components.

The first component is a kernel object mapping system that externally identifies the dynamic kernel objects of the monitored OS kernel at runtime, and our aim is to observe memory accesses to kernel data objects. This component is essential because it enables an external monitor to recognize the data objects that are targeted by the accesses. As well as being an infrastructure to recognize data objects, this system provides effective applications such as the detection of data hiding kernel malware attacks and the analysis of malware behavior targeting dynamic kernel objects.

In addition to the kernel data mapping system, we propose a new approach that detects malware by matching memory access patterns that specifically occur during malware attacks.

The thesis of this dissertation is as follows: *it is possible to detect a class of kernel malware that has recurring kernel memory reference patterns specific to malware attacks.* A software system that detects malware by using these patterns can be constructed using virtual machine technology without modifying the source code of the monitored kernel. Dynamic kernel analysis can produce effective malware signatures that can suppress frequent false positives in typical workloads by extracting malware memory reference patterns specific to malware attacks.

1.3 Contributions

The contributions of this dissertation are as follows:

- **General Data Properties**. In this dissertation, we use the general characteristics of data objects in memory operations (e.g., read, write, allocation, and deallocation) to characterize kernel malware behavior. We discuss how data properties are different from the code properties that are used by existing malware detection approaches. Then, we describe the details of the data properties and define a model for kernel memory access patterns with the data properties.

- **Runtime Kernel Memory Object Mapping for Malware Detection**. Our approaches externally inspect kernel data behavior for tamper-resistance of the monitor. An external monitor is required to identify kernel objects before it uses their properties because of its position. We propose a kernel mapping approach with two characteristics that can be effective in malware defense: (1) an un-tampered view of the data objects resistant to memory manipulation and (2) a temporal view that captures the allocation context of the dynamic memory.

- **Kernel Malware Detection based on Memory Access Patterns**. We propose a new approach that detects malware based on recurring kernel memory access patterns specific to malware attacks. As this approach uses the properties of data objects, it provides an alternate means to current malware defense approaches that rely only on the code properties of malware behavior. In addition, this new approach exposes the data access patterns of malware attacks in a general form. Therefore, it can be effective at detecting not only malware with signatures but also malware variants without signatures based on common data access patterns.

1.4 Terminology

This section presents the definitions of the terminology used in this dissertation.

- **Malware**. We use the definition of Malware from [38]. Malware is malicious software that is designed to disrupt or deny operation, gather information that leads to loss of privacy or exploitation, gain unauthorized access to system resources, and show other abusive behavior. It is a general term for many kinds of malicious software, such as computer worms, viruses, spyware, adware, trojan horses, and rootkits.

- **Rootkit**. We adapted the definition of rootkit from [3]. In this book, a rootkit is defined as a set of programs and code that allows a permanent or consistent, undetectable presence on a computer. This program is used to maintain access to an administrator's privilege in the system. Rootkits have various features to accomplish this goal by manipulating system resources. The features that many rootkits provide include hiding files, concealing network connections for hidden services, hiding processes, and selectively removing system logs. In particular, we focus on rootkits that operate in the kernel mode, which are called *kernel (-level) rootkits*. When we use the term "rootkit" in this dissertation, we refer to a kernel rootkit.

- **Code Injection**. Code injection is an attack mechanism to introduce unauthorized code to the program and move control flow to the new code. A wide range of malware is based on this technique. For instance, computer worms, viruses, exploits, rootkits, and recently SQL-injection belong to this category [1–3, 39].

- **Kernel Object Map**. This term represents *a map of kernel objects* that includes both static and dynamic kernel objects. The map includes detailed information about kernel data objects such as the address ranges and the types

of kernel objects. This map is used in many approaches that detect and analyze kernel malware [40–45].

- **Kernel Memory Mapping**. This term represents the process to generate a kernel object map. A typical input is a memory snapshot [40–42] or an execution of the monitored operating system [43–45]. The output is a kernel object map.

- **Virtual Machine**. We adapted our definition of a virtual machine from [46]. Virtual machine systems are efficient simulators for multiple copies of a machine on themselves. A virtual machine (VM) is the simulated machine. The simulator software is called the virtual machine monitor (VMM) or the hypervisor. The formal requirements of virtual machine systems were presented in [47]. Early systems which used virtual machines include CP-67 and the IBM 360/67. We use a virtual machine technique to implement a monitoring system for operating system kernels. This technique is also called "operating system virtualization."

1.5 Assumptions

In this dissertation, we assume that the monitored computer system has a single CPU, which has one memory address space in kernel mode; therefore, we use one kernel memory map to inspect an operating system kernel and detect kernel malware throughout this work. Multi-processor (or multi-core) systems can be supported by building kernel object maps depending on the number of kernel memory address spaces supported by the CPUs.

The memory system for multi-processors (or multi-core) can have either a shared memory architecture or a distributed memory architecture [48]. In a shared memory architecture, all of the CPUs share a single physical memory address space. To support this memory architecture, building one kernel memory object map is necessary, similar to a single CPU system. In distributed memory systems, CPUs have their own

local memory spaces which operate independently. The security monitor then needs to support multiple memory spaces by generating a memory map for each processor.

We assume the operating system kernel manages dynamic kernel memory with a set of kernel memory management functions. If the kernel code uses memory without explicit memory allocation and deallocation events (e.g., treat the memory as a buffer), it cannot be supported by our approach.

This dissertation presents our approach, techniques, implementation, and experiments in the context of malware defense for operating system kernels. The applicability of materials to other system layers is discussed in Section 6.1.

1.6 Organization

This dissertation is organized in six chapters. Following this introductory Chapter 1, we present related approaches in kernel malware defense and analysis in Chapter 2. The approaches based on data properties are introduced in Chapter 3. Chapter 4 presents a new mechanism to generate a runtime kernel object map, which becomes the basis of our data property-based approaches. In addition to a kernel object map, Chapter 5 presents the detection of malware using memory access patterns specific to malware. Chapter 6 concludes this dissertation and presents future work.

2 RELATED WORK IN MALWARE ATTACKS AND DEFENSE

In this section, we discuss related work on malware attacks and defense mechanisms.

2.1 Code Injection Attacks and Code Integrity-based Approaches

Code injection attacks insert unauthorized code into a program's memory space and transfer the control to the injected code. Various kinds of malware, such as computer worms [1], viruses [2, 49], shell code [50], and rootkits [3, 10, 11] use this technique to change program behavior with malicious purposes. There are various attack vectors to inject code. For instance, kernel rootkits load rootkit code into kernel memory space by using kernel drivers or raw memory devices. Then, they move the kernel control to the injected code by patching the system call table or function pointers.

This category of malware can be defeated by enforcing the integrity of the program's code and only allowing the execution of authorized and un-tampered code. There are various mechanisms to achieve this in the user space, in the kernel space, and also in the hardware level.

In the user space, several types of approaches have been developed to detect or prevent code injection attacks. Stack overflow was one of most popular attack vectors. StackGuard [6] stops stack overflow by placing canary values and detecting the manipulation of return addresses. ProPolice [7] further reorganizes stack frames to make buffer overflow difficult. In addition to stack protection, format string vulnerabilities and heap vulnerabilities are also addressed by similar approaches [51, 52].

Another category of approaches prevents attacks by enforcing the non-executable page permission, which is generally known as the NX bit. The code pages are supposed to be executable, but they are generally not meant to be modified. The data pages

can be writable, but their execution should be prevented. The pages containing the injected code are enforced to be not executable. Therefore, their execution is prevented. Memory pages should have either writable or executable permission (but not both) and this characteristic is known as the W $\bigoplus$X property. Hardware vendors have implemented it under various names (XD bit for Intel [53], Enhanced Virus Protection for AMD [54], and XN – eXecute Never – bit for ARM [55]).

For CPUs without this support, non-executable pages can be implemented as a software patch in the kernel code. Examples of software-based implementation include PAGEEXEC by PaX [56], Exec Shield by Redhat [57], and W$\wedge$X by OpenBSD [58]. Microsoft Windows supports non-executable pages when hardware support is available, and this service is called Data Execution Prevention (DEP) [59].

While these approaches are effective for user programs, kernel malware has the same privilege with kernel code. Thus, it is capable of directly manipulating kernel code and hardware configurations. For this reason, defeating kernel malware requires a monitoring mechanism which has a higher privilege than the OS kernel. For instance, many approaches are based on the hardware layer, such as a PCI device [60] or a layer comparable to hardware such as a virtual machine monitor [10, 11, 61].

Several intrusion detection approaches have been proposed to defeat kernel malware by checking kernel code integrity. Copilot [60] detects kernel rootkits by determining the manipulation of kernel text and invariant data structures. The state-based control flow integrity checking system (SBCFI) detects kernel malware by validating kernel components relevant to kernel control flow, such as the kernel text, system call table, and function pointers [61].

NICKLE [10] and SecVisor [11] proposed a stronger and more effective form of kernel code integrity checking. They determine kernel integrity violation by checking the execution of injected code at runtime. These approaches prevent kernel malware by allowing execution of only authorized and un-tampered kernel code.

2.2 Non-Code Injection Attacks and Defense Approaches

While many malware programs rely on code injection, there is another group of malware that does not require the insertion of malicious code for attacks. The malware of this class reuses an existing program's code to elude intrusion detection approaches based on code integrity. Following are several attack vectors of this group of malware.

Kernel Memory Devices. Operating systems have kernel memory devices that allow the read and write capability of raw kernel memory. For example, Linux has several devices, such as `/dev/kmem`, `/dev/mem`, and `/dev/kcore`; and Microsoft Windows has similar devices called `\Device\PhysicalMemory` and `\Device\DebugMemory`. These devices are intended for kernel debugging, efficient access to video memory, and memory forensic analysis; but if they are misused for malicious purposes, they can be a serious threat to the kernel's integrity. Some kernel rootkits use these devices to manipulate kernel memory without using kernel drivers [22]. In the Windows platform, several worms (e.g., `W32/Myfip.h` and `W32/Fanbot.A`) use raw memory device `\Device\PhysicalMemory` to tamper with kernel memory [62]

Return-oriented Programming. Return-oriented programming [14, 15] generates an attack by combining a large number of short instruction sequences (called *gadgets*) that allow arbitrary computation. This technique is also used to implement kernel level malware (e.g., return-oriented rootkits [16]). This type of malware only uses existing kernel code and does not violate the W$\bigoplus$X property. Thus code integrity-based approaches cannot detect its attacks. Several approaches have been proposed to use runtime characteristics during attacks [17–20] to detect this malware. Other approaches attempt to remove potential gadgets by removing return instructions [21] or potential instruction sequences which can be used as gadgets [63] from the program.

Jump-oriented Programming. As detection approaches for return-oriented programming appear [17–21], other instruction sequences, similar to the return gadgets, are used for constructing attacks [64–66]. These approaches use instruction sequences

that end with jump instructions to connect multiple code pieces and to create malicious logic. These approaches show that essentially any instruction sequence whose control flow can be manipulated by attackers can be used for attacks. This idea was conceptualized as "free-branches" by Kornau et. al. [67].

Vulnerable Code in OS Kernel. Most operating system kernels potentially carry programming bugs [23–25]. Some of them are found and fixed by developers. However, attackers also find bugs and use them to compromise systems. For instance, CVE-2010-3081 describes vulnerable kernel code that has existed since 2008. This vulnerability has allowed attackers to gain the administrator privilege in virtually all 64 bit Linux systems using a simple user program (called a root exploit). This bug was patched in Fall 2010. Vulnerable code such as this, being part of legitimate kernel code, is difficult to detect for code integrity-based approaches if a malware attack is triggered using kernel bugs.

Third-party Drivers. Kernel drivers are dynamically loaded at runtime. To ensure the integrity of a kernel, this driver code should be properly handled. Code integrity-based approaches [10,11] solve this problem by allowing a list of authorized drivers for execution (e.g., a white list determined by a system administrator or the drivers signed by operating system vendors [68]). These drivers are typically authorized without systematic examination of code behavior for safety. Rather, the authorization is based on trust in the developers and vendors of operating system (OS) kernels. Many hardware vendors ship proprietary drivers without disclosing the source code. In such case, the drivers may include potentially vulnerable code or hidden malicious code that can be exploited for attacks. The rootkit case of Sony [69] shows one example of code from vendors that can have undesired effects.

2.3 Malware Defense Based on Code Behavior Signatures

There has been a variety of approaches which characterize malware's behavior by using its control flow. Several approaches [26–29] build control flow graphs using

system call events, and another approach [30] uses CPU instructions to represent malware behavior. While these malware patterns are derived from the events of different system layers, they commonly represent the control flow of malware, which is a sequence of code with causal dependence. There are two kinds of challenges for these approaches.

First, advanced malware can generate variations in the control flow to avoid detection by these approaches. Several papers describe obfuscation techniques such as dead code insertion, code transformation, and instruction substitution [31–34]. Malware can obfuscate its code execution while retaining the same algorithm. In addition, researchers introduced a new obfuscation technique that hides specific trigger-based behavior by encrypting the code dependent on an input [33].

Second, malware's control flow can dynamically vary at runtime and the detection mechanism using malware's code behavior should be able to handle such variations. Balzarotti *et al.* presented a system [26] that uses system-call trace to determine analysis-aware malware. In this paper, the authors described several cases where the system-call trace can be inconsistent, such as the expiration of timeout and the delivery of signals. Their system handles this problem by using a flexible matching algorithm.

2.4 Malware Defense Based on Data Signatures

Like any other program, malware uses data structures. Some malware has its own data structures. Other malware tampers with the data structures that they target to make changes in the program's behavior. There are several approaches that detect malware based on the signatures of data structures.

Laika [70] determined data structures from a program's memory. As one application, the authors presented the detection of a botnet program by classifying data structures specific to malware. This approach is effective for user space malware because each user program has its private memory space. However, kernel memory is

shared by the kernel text and many kernel drivers. Malware's code and data are part of a huge number of legitimate kernel code and data and therein lies the challenge to applying this technique to kernel malware detection.

Dolan-Gavitt *et al.* proposed an approach that discovers data structures from memory snapshots using value constraints [71]. This approach uses value properties, such as constants, bitwise AND values, and alignments, to match data structure instances. By using a fuzzing technique, they showed the generated value properties are reliable to be used as a signature.

SigGraph is another type of data scanner which uses pointer constraints to match data objects [72]. This approach generates a signature with a pointer connection graph rooted at the data structure. To detect data structures, it scans memory snapshots in a brute-force way as it matches pointer connections.

These approaches can discover data structures from a memory image in a benign scenario. However, if those approaches are targeted for detecting malware, there could be the following challenges. First, malware can manipulate the data objects so that the data scanners fail to detect them while such objects are being properly used by malware code. For example, malicious code can set invalid values or pointers in the data structure while the injected malware code properly uses such objects. Second, it is possible that some data structures may not have enough constraints to be matched by these approaches. For instance, if a kernel data structure is simple, such as a string buffer, these approaches do not have specific constraints to match them, leading to many false positive cases.

2.5 Kernel Integrity Checking based on Kernel Memory Mapping

There have been several approaches [40, 41, 44, 45, 61] that leverage kernel memory mapping to test the integrity of OS kernels and detect kernel malware. These approaches identify kernel memory objects by recursively traversing pointers in the kernel memory starting from static objects in a similar way to garbage collection mech-

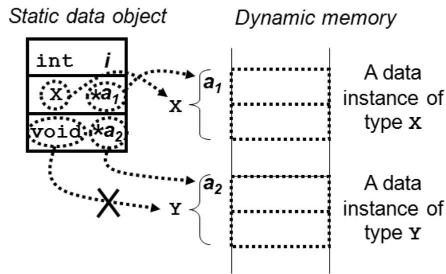


Figure 2.1.: Illustration of type-projection mapping

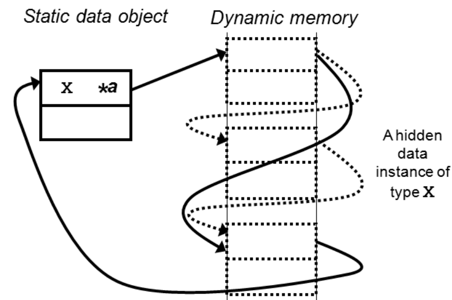


Figure 2.2.: Data hiding attack via pointer manipulation

anisms [73, 74]. A kernel object is identified by projecting the address and the type of the traversed pointer onto memory; thus, we call this mechanism *type-projection mapping*. For example, in Figure 2.1 the mapping process starts by evaluating the pointer fields of the static data object. When the second field of this object is traversed, the type `X` of the pointer is projected onto the memory located in the obtained address a_1, identifying a data instance of type `X`.

The underlying hypothesis of this mapping is that the traversed pointer type accurately reflects the type of the projected object. In practice there are several cases that this is not true. First, if an object allocated using a specific type is later cast to a generic type (e.g., `void*`), then this mapping scheme cannot properly identify this object using that pointer. For instance, in Figure 2.1 the third field of the static object cannot be used to identify the Y instance because of its generic `void*` type. Second, in modern OSes many kernel objects are linked using embedded list structures that connect the objects using list types. When these pointers are traversed, the connected objects are inaccurately identified as list objects. KOP [41] addresses these problems by generating an extended type graph using static analysis. Some other approaches [40, 42] rely on manual annotations.

When type-projection mapping is used against kernel malware, these problems may pose concerns as such inaccuracy can be deliberately introduced by kernel malware. In type-projection mapping, *the kernel memory map is based on the content*

of the kernel memory, which may have been manipulated by kernel malware. This property may affect the detection of kernel rootkits that hide kernel objects by directly manipulating pointers. For instance, Fig 2.2 shows a potential malware attack case. In this Figure, initially a singly linked circular list is composed of four data instances. If malware manipulates connecting pointers as shown in the Figure, the third instance will disappear from the linked list. The type-projection mapping does not have information to determine this attack because it constructs a map of data instances based on memory values.

To detect such attacks, a detector needs to rely on not only a kernel memory map but also additional knowledge that reveals the anomalous status of the hidden objects. For this purpose, several approaches [40–42] use data structure invariants. For example, KOP [41] detects a process hidden by the FU Rootkit [75] by using the invariant that there are two linked lists regarding process information that are supposed to match, and one of them is not manipulated by the attack. However, a data invariant is specific to semantic usage of a data structure and may not be applicable to other data structures. For type-projection mapping, it is challenging to detect data hiding attacks that manipulate a simple list structure (such as the kernel module list in Linux) without an accompanying invariant.

In general, we can categorize these approaches into two categories based on whether they make use of a static snapshot or dynamic runtime memory trace.

2.5.1 Static Type-projection Mapping

This approach uses a memory snapshot to generate a kernel memory map. SBCFI [61] constructs a map to systematically detect the violation of persistent control flow integrity. Gibraltar [40] extracts data invariants from kernel memory maps to detect kernel rootkits. A significant advantage of this approach is the low cost to generate a memory snapshot. A memory snapshot can be generated using an external monitor

such as a PCI interface [40], a memory dump utility [41], or a VMM [61], and the map is generated from the snapshot later.

The memory snapshot is generated at a specific time instance (asynchronously). Its usage is limited for analyzing kernel execution traces where dynamic kernel memory status varies over kernel execution. The same memory address, for example, could store different dynamic data objects over a period of time (through a series of deallocations and reallocations). The map cannot be used to properly determine what data was stored at that address at a specific time. We call this a *dynamic data identity problem*, and it occurs when an asynchronous kernel memory map is used for inspection of dynamic memory status in the kernel execution traces.

2.5.2 Dynamic Type-projection Mapping

This mapping approach also uses the type-projection mechanism to identify kernel objects, but its input is the trace of memory accesses recorded over runtime execution instead of a snapshot. By tracking the memory accesses of malware code, this approach can identify the list of kernel objects manipulated by the malware. PoKeR [44] and Rkprofiler [45] use this approach to profile dynamic attack behavior of kernel rootkits in Linux and Windows respectively.

As a runtime trace is used for input, this approach can overcome the asynchronous nature of static type-projection mapping. Unfortunately, current work only focuses on the data structures targeted by malware code, and may not capture other events. For example, many malware programs call kernel functions during the attack or exploit various kernel bugs, and these behaviors may appear to be part of legitimate kernel execution. In these cases, dynamic type-projection techniques need to track all memory accesses to accurately identify the kernel objects accessed by legitimate kernel execution. As this process is costly (though certainly possible), it is not straightforward for this approach to expand the coverage of the mapped data to all kernel objects.

2.6 Kernel Rootkit Profilers

Kernel rootkit profilers [44, 45] analyze a variety of aspects of rootkit behavior such as the memory access targets of malware code or user space impact. These approaches derive the types of the attack targets by transitively deriving types of kernel objects from static objects based on the rootkit behavior. However, some attacks are difficult to be understood based on such assumption because rootkits can use various other resources, such as hardware registers, to find the attack targets [76]. We demonstrated that there exist at least two real-world rootkits and two proof-of-concept rootkits which can elude PoKeR [44].

K-Tracer [77] can analyze the malicious behavior of kernel rootkits in sensitive events using dynamic slicing techniques. Its algorithm requires determination of the sensitive data so it therefore can be difficult to analyze DKOM attacks [75] whose targets may not be predetermined.

3 DATA-CENTRIC APPROACHES TO KERNEL MALWARE DEFENSE

In the previous chapter, we discussed related approaches in malware detection and analysis. Many of those approaches characterize malware behavior based on code information such as injected code and malicious control flow. We call such approaches *code-centric* approaches because of their reliance on code information. In contrast, we propose new approaches based on the properties of data objects and their access patterns. Based on their use of data information to characterize malware, we call them *data-centric* approaches. In this chapter, we first will distinguish the differences between these two approaches. Then, we will present the details of our approach of characterizing malware behavior based on data properties.

3.1 Code-centric Approaches versus Data-centric Approaches

3.1.1 Code-centric Malware Defense Approaches

Code-centric malware defense approaches use the properties of malicious code to detect malware. Such approaches are illustrated in Figure 3.1. A square block named c_x represents a code entity such as a block of CPU instructions or a larger chunk of code such as a system call.

Code injection is a commonly used technique by many malware programs. The unauthorized injected code is shown as the shaded square c_ε. This attack technique can be detected by checking the integrity of the authorized code and verifying whether only such code is being executed. This methodology is referred to as the approach based on code integrity (Section 2.1).

Malware programs often show specific code sequences during attacks. Several malware detection approaches generate malware signatures by using such sequences.

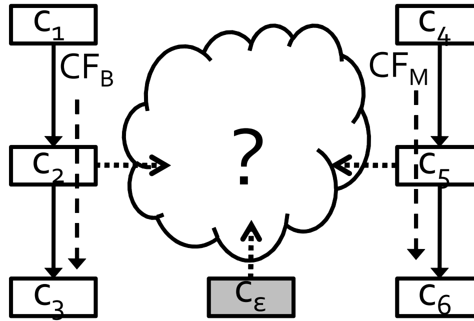


Figure 3.1.: Code-centric malware defense approaches

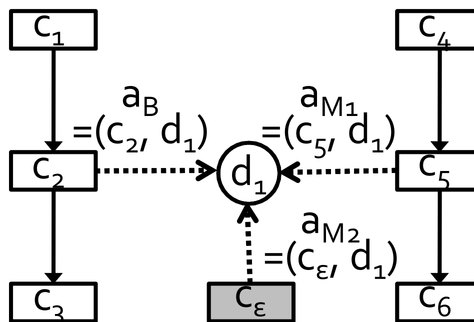


Figure 3.2.: Data-centric malware defense approaches

In Figure 3.1, a code sequence of $c_1 \rightarrow c_2 \rightarrow c_3$ represents a control flow that occurs in a normal program status. This flow is shown as a dashed arrow named as CF_B. Let us assume that a malicious program always executes the sequence of code $c_4 \rightarrow c_5 \rightarrow c_6$ (CF_M). This control flow can be used to match this malware's behavior as the malware signature (Section 2.3).

As discussed in Chapter 2, these approaches can be eluded by advanced malware techniques. Existing code can be reused to create malicious logic (e.g., use c_5 instead of c_2 to access data). Then code integrity-based approaches are not applicable to detect the attacks. Code obfuscation techniques [31–34] can change CF_M to another flow, $c_4 \rightsquigarrow c_3 \rightsquigarrow c_5 \rightarrow c_6$. Then the approaches based on malware code behavior can be eluded.

3.1.2 Data-centric Malware Defense Approaches

The effects of code execution are typically reflected to data memory, and such data accesses are shown as dotted arrows in Figures 3.1 and 3.2. A code-centric malware defense approach only uses the code information for characterizing the malware activity. Therefore, the data information is presented as a cloud in Figure 3.1.

We want to identify this missing information and use it for kernel malware detection. Specifically, we use the general characteristics of data objects regarding their usage, which are called the *general data object properties* (Section 3.5). This information characterizes the lifetime events of a data object, which include the allocation, the accesses, and the deallocation of an object. These events are expressed as the properties of program execution, such as the addresses of the code that invokes memory operations. This information is *general* from the aspect that any data types of data objects have such lifetime events.

Figure 3.2 presents our malware defense approach based on general data properties. Here the cloud area is clarified because the accessed object d_1 is identified. To make this information available, we designed a runtime kernel memory mapper (Section 4). Malware behavior can be described with more details about what data structures are accessed in addition to what code is executed.

A memory access pattern a_B in a benign control flow CF_B can be expressed as a pair of the accessing code c_2 and the accessed data d_1, (c_2, d_1). Here let us assume that this is the only access pattern found in the program source code. If another code, such as c_3, accesses d_1, this memory access can be determined as an anomaly. This data information can be applied to determine attacks exploiting the existing code. Also, in the case where malware obfuscates its control flow CF_M, if c_5 still accesses d_1 after the obfuscation, the malware attack would remain detected in this approach.

In summary, introducing data information improves the details of malware behavior descriptions. In this dissertation, we propose the data-centric malware defense architecture (DMDA), which models and detects malware behavior using the proper-

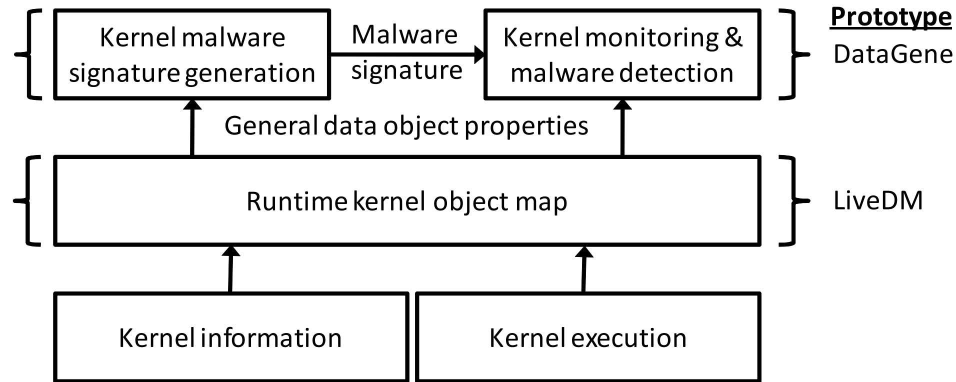


Figure 3.3.: Design of data-centric malware defense architecture

ties of kernel data objects. We first present how we can generate the data information (e.g., d_1 in Figure 3.2). Then we present our new approach to characterize malware based on kernel memory access patterns specific to malware attacks.

3.2 Design

The design of DMDA is illustrated in Figure 3.3. With the input of the operating system kernel information and kernel execution, we first generate a kernel object map (Chapter 4). This map is synchronously updated at runtime; thus, it enables us to determine the targets of the kernel memory references. Using this map, we can monitor and analyze kernel memory access patterns. By comparing benign kernel execution and malicious kernel execution compromised by kernel rootkits, we systematically extract the memory reference patterns specific to malware attacks. We match these memory access patterns as malware signatures to detect kernel rootkits (Chapter 5). In the following chapters, we will present each component in detail.

3.3 Objectives

In the design of DMDA, we seek to achieve the following objectives.

- **External Monitoring**. There is a design choice between internal and external

monitoring. Each choice has its advantages and disadvantages, which will be discussed in Section 3.4. We place primary emphasis on the tamper-resistance of the monitor; therefore, we chose a monitor to be placed outside of the monitored kernel.

- **Transparent Monitoring**. We intend to design a security monitor that does not require changes in the source code of the monitored kernel. Many widely-used modern operating systems are proprietary. Avoiding source code changes would facilitate the implementation supporting a wider scope of operating systems.

- **Un-tampered Data View**. Some kernel rootkits implement stealthy services by manipulating pointers in kernel data structures. Therefore, the memory content is subject to malware manipulation and should not be trusted. The approaches based on memory snapshots [40, 41, 61] may have tampered views because the map construction is based on memory status. We derive an un-tampered data view by using memory allocation and deallocation events, instead of memory values.

- **Temporal Data View**. Dynamic memory can represent multiple different kernel objects depending on its memory allocation context. If the kernel object map is not synchronously updated (temporal) for each allocation event, the kernel object information may be outdated. This is particularly important for our approach which uses memory reference patterns because it provides accurate targets of memory accesses. Related kernel object mapping approaches based on periodic memory snapshots [40, 41, 61] are not suitable for this purpose.

3.4 Types of Monitoring: Internal versus External

The location of a malware detector involves several design trade-offs in information collection, performance, implementation efforts, and reliability against attacks.

3.4.1 Internal Monitors

An intrusion/malware detection monitor can be embedded inside the monitored software (e.g., a user program or an OS kernel). This type of monitor is called an *internal monitor*. Internal sensors [78] and embedded detectors [6, 51] belong to internal monitors. As a part of a program, it has the convenience of being able to access and evaluate the data structures of the monitored program. The performance of these monitors depends on the frequency of the monitored activity and the overhead per activity.

Implementation efforts would differ depending on how the monitoring code is generated. If the code is manually generated, the developer may need to understand potentially vulnerable code to decide where the monitoring code should be placed. In such cases, the implementation effort would be considered high. Systematic approaches, such as compiler-based methods, could lower the amount of human efforts required.

The reliability of a monitoring activity against potential attacks is an important issue for its credibility. Internal monitors are part of the monitored code so potentially malicious code therefore has direct access to the monitoring code. Unless there is a safety mechanism to ensure the integrity of an embedded monitor, it is exposed to potential manipulation by malware.

3.4.2 External Monitors

A monitor can be placed outside the monitored software. This type of monitor is called an *external monitor*. The monitors based on an external PCI device [60], memory dump programs [40, 41], and a virtual machine monitor (VMM) [10, 37, 42, 61, 79] belong to the external type monitors group.

A significant advantage of these monitors is their reliability against potential attacks. As the monitor is located outside the monitored software, potential attack

code within the monitored program does not have direct access to the monitor. This tamper-resistance is essential for trusting the operation of the monitor.

Being outside the monitored program, this approach inherently has challenges, however, in interpreting the internal status of the monitored program. For example, when the VMM observes the memory of an OS, the memory status is viewed as raw bits and bytes. An external monitor needs to interpret this low level representation to high level information to determine an intrusion or infection. This problem is often called a *semantic gap* [37].

The overhead of these monitors occurs when the monitor obtains the information of the monitored entity. Once the information is grabbed, it can be processed in parallel, thus avoiding a slow-down of the monitored program. The implementation effort depends on what information is obtained from the monitored program and also its interpretation of the extracted data.

In this dissertation, we emphasize the tamper-resistance of the monitor; therefore, we implement our system as an external monitor. Specifically, we use a virtual machine monitor to inspect operating system kernels.

3.5 General Data Object Properties and a Model for Kernel Memory Access Patterns

3.5.1 General Data Object Properties

Data objects have several usage patterns in their lifetime. Dynamic data objects are created (allocated) by some code. The values in the data objects are read or overwritten. In the case of dynamic objects, they are destructed (deallocated) after their usage. We call the properties of data objects in such usage patterns *general data object properties*. In this section, we present the details of the properties we use to monitor OS kernels and detect kernel malware.

Figure 3.4 illustrates the lifetime of a dynamic data object, which consists of the memory operations applied to a data object and the data properties related to the operations. Code c_a calls a memory allocation function, `kmalloc`, and a memory

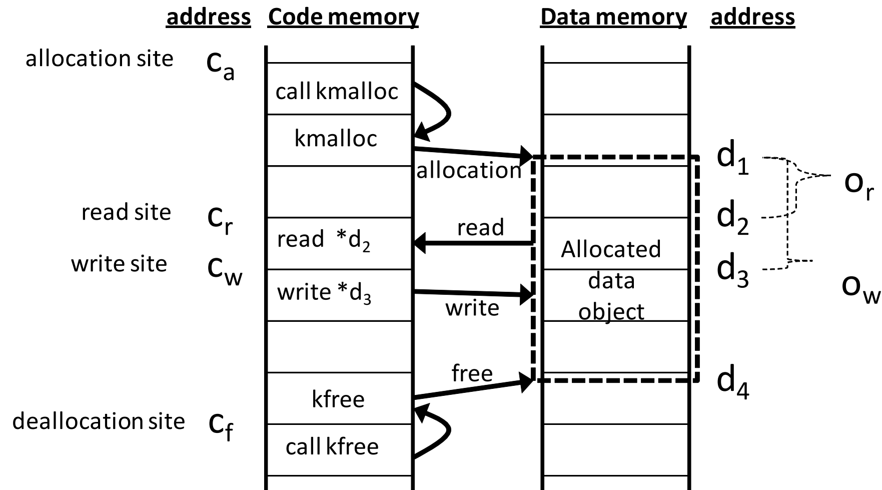


Figure 3.4.: A lifetime of a dynamic data object

block at the address range from d_1 to d_4 is allocated. This memory is used by read and write memory accesses. Code c_r reads values from the address $d_2(= d_1 + o_r)$. Code c_w writes some value to the address $d_3(= d_1 + o_w)$. At the end of the lifetime of this object, code c_f calls a deallocation function, `kfree` to free this memory.

In this example, various data object addresses (d_x), the accessed offsets (o_x), and the code addresses (c_x) represent how this program handles this object. We describe such properties related to memory operations in detail below.

Allocation and Deallocation. A dynamic object is allocated when a memory allocation function is called. The address of the code that invokes a memory allocation function is referred to as an *allocation call site*. This event is the start of the lifetime of the allocated object. A deallocation event is the end of the lifetime of a dynamic object. Similarly, the code address that calls a memory deallocation function is referred to as a *deallocation call site*. Figure 3.4 illustrates the lifetime of a dynamic data object, and these events are presented respectively as allocation and free.

Identification of Objects. When we identify data objects, we use the term, *class*, to represent the identification of static and dynamic objects in a unified way. In the case of static data objects, their data types and the address ranges are statically

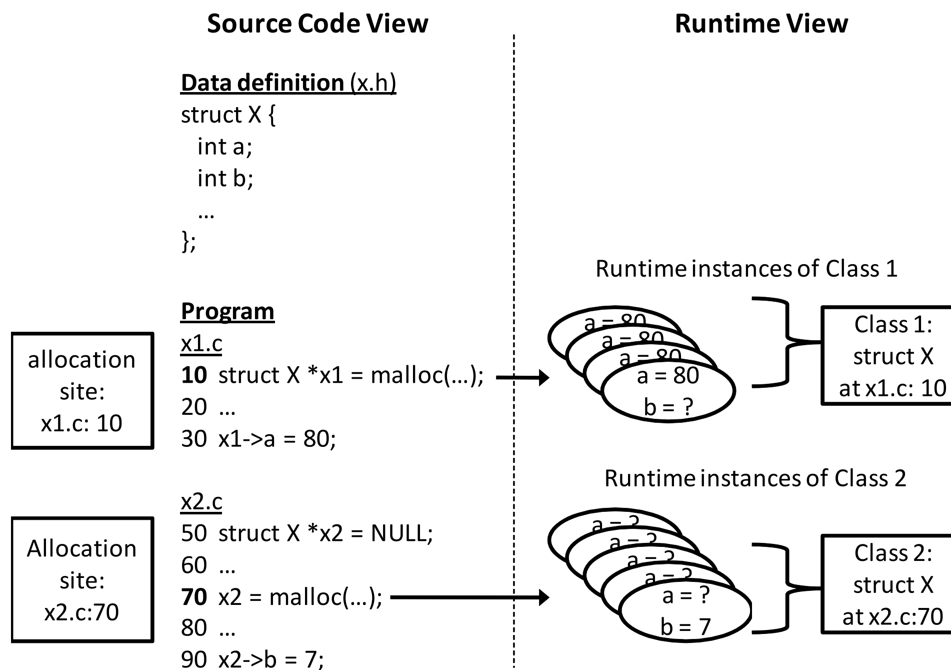


Figure 3.5.: Identification of dynamic data objects

assigned. This information is available in symbol tables (e.g., `System.map` in Linux). By assigning a specific number to each object, we can uniquely identify its type and address range. We call this unique number a class for a static object.

Identifying dynamic objects is more complicated for two reasons [36]. First, the number of instances dynamically varies at runtime. Second, most dynamic memory functions in unmodified operating systems do not maintain the type information for each object. To derive the type information of dynamic objects, we introduce a technique that infers data types using allocation call sites in Section 4. For dynamic objects, an allocation call site is used as a class.

An allocation call site can be translated to a data type, but more precisely it is a *sub-class of a data type with the origin information* where the objects are instantiated. Figure 3.5 illustrates this relationship. A data type `X` is defined in `x.h`. Two allocation sites instantiate the data objects of this data type. The objects of type `X` allocated at `x1.c:10` have the `a` fields set as 80. Another group of objects of type `X` allocated

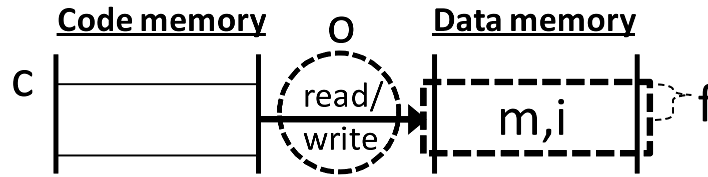


Figure 3.6.: A model for memory access patterns

at `x2.c:70` have their `b` fields set as 7. Although these objects have the same type, their usages can be different depending on their origins, namely, the allocation sites. This information enables finding the source of the data objects. It can be helpful to analyze program bugs related to the heap memory; and, in malware analysis, it also can provide understanding about the specific targets of malware attacks.

Data Fields. Non-primitive data structures typically consist of multiple data fields which are located at specific offsets in data structures. When a reference monitor inspects memory accesses, the offsets can be simple and efficient representations of data fields. Also, by using type definitions, offsets can be converted to field information.

Memory Access Code. During computation, memory values are loaded to a CPU and the computed values are stored back to the memory. Such reads and writes are the fundamental operations that a CPU accesses memory. In program execution, the set of code that accesses data objects represents how the objects are used in the program. We call the code that reads memory values a *read access site*. Similarly, the code that writes to memory is called a *write access site*.

3.5.2 A Model for Kernel Memory Access Patterns

To characterize kernel malware behavior and detect its attacks, we use kernel memory access patterns. In this section, we briefly describe a formal representation of data access patterns, which are composed of general data object properties. Chapter 5 will present more details along with a running example.

We call a memory access pattern a *data behavior element (DBE)*, which is defined as a quintuple (5-tuple) of the general object properties illustrated in Figure 3.6:

$$(c, o, m, i, f) \tag{3.1}$$

c represents the information of the accessing code, and o shows whether this access is a read ($o = 0$) or a write ($o = 1$). m and i represent the information about the accessed data object. If this object is static, m is 1 and i is the serial number that we assigned based on the information generated in the compile time. If it is a dynamic object, m is 0 and an allocation call site is used for i to represent the type of this object. This information can infer its data type using debugging information and source code analysis as described in Section 4.2.2. Finally, f shows the offset, which represents the accessed field within the data object. By using the type definition, it can be translated to a field symbol.

A DBE describes a single memory access pattern. An operating system kernel instantiates tens of thousands of kernel objects from hundreds of kernel data types. These runtime objects are read and overwritten by thousands of code sites in the kernel. We collect a set of such kernel memory access patterns to represent the runtime data access behavior of an operating system kernel. We call this set of DBEs a *data behavior profile (DBP)*, which is defined for an instance of the kernel execution that starts from its booting and ends at its shutdown.

3.5.3 A Conceptual View of General Data Object Properties

The general data object properties show how kernel data objects are used in a kernel. Therefore, the behavior of the kernel can be organized in a view *centered* by kernel data information. Figure 3.7 presents this perspective in conceptual views of the general data object properties. A rounded box represents the data objects of a specific class.

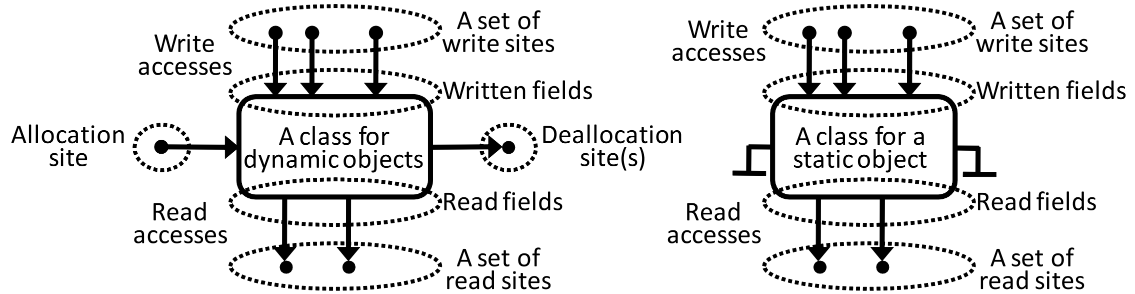


Figure 3.7.: Conceptual views of the general data object properties

The case of dynamic objects is shown in the left side of Figure 3.7. An allocation event becomes the start of its lifetime. The end of its lifetime is defined by its deallocation event(s) which is shown as a deallocation site(s). The case of static objects is shown in the right side of Figure 3.7. Note that, in this case, allocation and deallocation information is removed because static objects are determined at compile time.

At runtime, these objects are used via read and write accesses. On top of the boxes, a set of write access sites and the overwritten field offsets are shown. Similarly, under the boxes, a set of read access sites and the read fields are presented. This information is common in both dynamic and static objects.

In this chapter, we have presented the general data object properties and our model of kernel memory access patterns. This information is the foundation of our approaches, and the behavior of the benign kernel execution and kernel malware will be modeled in terms of this information. In the remainder of this dissertation, we present how to generate a runtime map of kernel data objects which enables the general data object properties in Section 4. On top of this system component, we present how to detect a class of malware which has specific memory access patterns based on the general data object properties in Section 5.

4 KERNEL MALWARE DETECTION AND ANALYSIS WITH UN-TAMPERED AND TEMPORAL VIEWS OF KERNEL OBJECTS

DMDA uses the properties of kernel data objects for malware detection. Because it employs external monitoring, the identification of data objects should be extracted from the kernel and reconstructed in the monitor. In this chapter, we will introduce a runtime kernel memory mapping mechanism and demonstrate its effectiveness.

4.1 Introduction

Dynamic kernel memory is where the majority of kernel data resides. Operating system (OS) kernels frequently allocate and deallocate numerous dynamic objects of various types. As a result of the complexity of identifying such objects at runtime, dynamic kernel memory is a source of many kernel security and reliability problems. For instance, an increasing amount of kernel malware targets dynamic kernel objects [42, 44, 75, 80]; and many kernel bugs are caused by dynamic memory errors [23–25].

Advanced kernel malware uses stealthy techniques such as directly manipulating kernel data (i.e., DKOM [75]) or overwriting function pointers (i.e., KOH [80]) located in dynamic kernel memory. This allows attacks such as process hiding and kernel-level control flow hijacking. These anomalous kernel behaviors are difficult to analyze because they involve manipulating kernel objects that are dynamically allocated and deallocated at runtime; unlike persistent kernel code or static kernel data that are easier to locate, monitor, and protect.

To detect these attacks, some existing approaches use kernel memory mapping based on the contents of runtime memory snapshots [40, 41, 61] or memory access traces [44, 45]. These approaches commonly identify a kernel object by projecting the type and address of a pointer onto the memory. However, such a technique may

not always be accurate – for example, when an object is type cast to a generic type or when an embedded list structure is used as part of larger data types. In benign kernel execution, such inaccuracy can be corrected [41]; but it becomes a problem in malware analysis as the memory contents may have been manipulated by kernel malware. For example, a DKOM attack to hide a process may modify the `next_task` and `prev_task` pointers in the process list. This causes the process to disappear from the OS view as well as from the kernel memory map. To detect this attack, some existing approaches rely on data invariants such as that the list used for process scheduling should match the process list. However, not every data structure has an invariant. Additionally, the kernel memory map generated from a snapshot [40,41,61] reflects kernel memory status at a specific time instance. Therefore, the map is of limited usage in analyzing kernel execution. Some mapping approaches are based on logging malware memory accesses [44,45] and thus provide temporal information. However they only cover objects accessed by the malware code and cannot properly handle certain attack patterns because of assumptions in its mapping algorithm [76].

In this chapter, we present a new kernel memory mapping scheme called *allocation-driven mapping* that complements the existing approaches. Our scheme identifies dynamic kernel objects by capturing their allocations and does not rely on the runtime content of kernel memory to construct the kernel object map. As such, the map is resistant to attacks that manipulate the kernel memory. On top of our scheme, we build a hidden kernel object detector that uses the un-tampered view of kernel memory to detect DKOM data hiding attacks without requiring kernel object-specific invariants. In addition, our scheme keeps track of each kernel object's life time. This temporal property is useful in the analysis of kernel/kernel malware execution. We also build a temporal malware behavior monitor that systematically analyzes the impact of kernel malware attacks via dynamic kernel memory using a kernel execution trace. We address a challenge in the use of kernel memory mapping for temporal analysis of kernel execution: A dynamic memory address may correspond to different kernel objects at different times because of the runtime allocation and deallocation

events. This problem can be handled by allocation-driven mapping. The lifetime of a dynamic kernel object naturally narrows the scope of a kernel malware analysis.

The contributions of this chapter are summarized as follows:

- We present a new kernel memory mapping scheme called allocation-driven mapping that has the following properties desirable for kernel malware analysis: un-tampered identification of kernel objects and temporal status of kernel objects.
- We implement allocation-driven mapping at the virtual machine monitor (VMM) level. The identification and tracking of kernel objects take place in the VMM without modification to the guest OS.
- We develop a hidden kernel object detector that can detect DKOM data hiding attacks without requiring data invariants. The detector works by comparing the status of the un-tampered kernel map with that of kernel memory.
- We develop a malware behavior monitor that uses a temporal view of kernel objects in the analysis of kernel execution traces. The lifetimes of dynamic kernel objects in the view guide the analysis to the events triggered by the objects manipulated by the malware.

We have implemented a prototype of allocation-driven mapping called LiveDM (Live Dynamic kernel memory Map). It supports three off-the-shelf Linux distributions. LiveDM is designed for use in non-production scenarios such as honeypot monitoring, kernel malware profiling, and kernel debugging.

4.2 Design of LiveDM

In this section, we first introduce the allocation-driven mapping scheme, based on which our LiveDM system is implemented. We then present key enabling techniques to implement LiveDM.

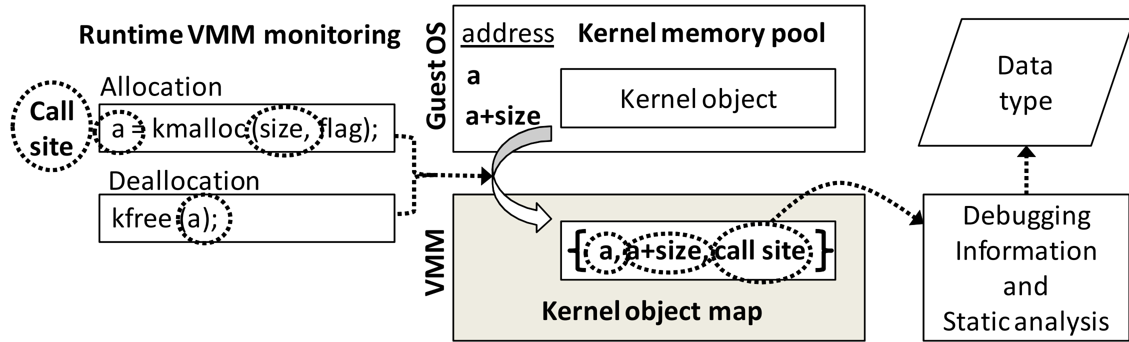


Figure 4.1.: Overview of LiveDM.

4.2.1 Allocation-driven Mapping Scheme

Allocation-driven mapping is a kernel memory mapping scheme that generates a kernel object map by *capturing the kernel object allocation and deallocation events* of the monitored OS kernel. LiveDM uses a VMM to track the execution of the running kernel. Figure 4.1 illustrates how LiveDM works. Whenever a kernel object is allocated or deallocated, LiveDM will intercede and capture its address range and the information to derive the data type of the object subject to the event (details in Section 4.2.2) to update the kernel object map.

First, this approach does not rely on any content of the kernel memory which can potentially be manipulated by kernel malware. Therefore, the kernel object map provides *an un-tampered view* of kernel memory wherein the identification of kernel data is not affected by the manipulation of memory contents by kernel malware. This tamper-resistant property is especially effective to detect sophisticated kernel attacks that directly manipulate kernel memory to hide kernel objects. For instance, in the type-projection mapping if the pointer fields of the static objects are nullified, dynamic objects cannot be identified because those objects cannot be reached by recursively scanning pointers in the memory. In practice, there can be multiple pointer references to a dynamic object. However, malware can completely isolate an object to be hidden by tampering with all pointers pointing to the object. The address of the hidden

object can be safely stored in a non-pointer storage (e.g., `int` or `char`) to avoid being discovered by the type-projection mapping algorithm while it can be used to recover the object when necessary. Many malicious programs carefully control their activities to avoid detection and prolong their stealthy operations, and it is a viable option to suspend a data object in this way temporarily and activate it again when needed [81].

In the allocation-driven mapping approach, however, this attack will not be effective. As shown in Figure 4.1, each dynamic object is recognized upon its allocation. Therefore the identification of dynamic objects is reliably obtained and protected against the manipulation of memory contents. The key observation is that allocation-driven mapping captures the *liveness status* of the allocated dynamic kernel objects. For malware writers, this property makes it significantly more difficult to manipulate this view. In Section 4.5, we show how this mapping can be used to automatically detect DKOM data hiding attacks without using any data invariant specific to a kernel data structure.

Second, LiveDM reflects a *temporal* status of dynamic kernel objects because it captures their allocation and deallocation events. This property enables the use of the kernel object map in temporal malware analysis where temporal information, such as kernel control flow and dynamically changing data status, can be inspected to understand complicated kernel malware behavior.

In Section 2.5.1, we pointed out that a *dynamic data identity problem* can occur when a snapshot-based kernel memory map is used for dynamic analysis. Allocation-driven mapping provides a solution to this problem by accurately tracking all allocation and deallocation events. This means that even if an object is deallocated and its memory reused for a different object, LiveDM will be able to properly track it.

Third, allocation-driven mapping does not suffer from the casting problem that occurs when an object is cast to a generic pointer because it does not evaluate pointers to construct the kernel object map. For instance, a general pointer such as a `void` pointer does not hinder the identification of the data instance that is pointed to by the pointer because this object is determined by capturing its allocation. However,

we note that another kind of casting can pose a problem: If an object is allocated using a generic type and it is cast to a specific type later, allocation-driven mapping will detect the earlier generic type. However, our study in Section 4.4 shows that this behavior is unusual in Linux kernels.

There are a number of challenges in implementing the LiveDM system based on allocation-driven mapping. For example, kernel memory allocation functions do not provide a simple way to determine the type of the object being allocated.[1] One solution is to use static analysis to rewrite the kernel code to deliver the allocation types to the VMM, but this would require the construction of a new type-enabled kernel, which is not readily applicable to off-the-shelf systems. Instead, we use a technique that derives data types by using runtime context (i.e., call stack information). Specifically, this technique systematically captures code positions for memory allocation calls by using virtual machine techniques (Section 4.2.2) and translates them into data types so that OS kernels can be transparently supported without any change in the source code.

4.2.2 Techniques of LiveDM

We employ a number of techniques to implement allocation-driven mapping. At the conceptual level, LiveDM works as follows. First, a set of kernel functions (such as `kmalloc`) are designated as kernel memory allocation functions. If one of these functions is called, we say that an allocation event has occurred. Next, whenever this event occurs at runtime, the VMM intercedes and captures the allocated memory address range and the code location calling the memory allocation function. This code location is referred to as an *allocation call site* and we use it as a unique identifier for the allocated object's type at runtime. Finally, the source code around each allocation call site is analyzed offline to determine the type of the kernel object being allocated.

---

[1]Kernel level memory allocation functions are similar to user level ones. The function `kmalloc`, for example, does not take a type but a size to allocate memory.

Runtime Kernel Object Map Generation

At runtime, LiveDM captures all allocation and deallocation events by interceding whenever one of the allocation/deallocation functions is called. There are three things that need to be determined at runtime: (1) the call site, (2) the address of the object allocated or deallocated, and (3) the size of the allocated object.

To determine the call site, LiveDM uses the return address of the call to the allocation function. In the instruction stream, the return address is the address of the instruction after the call instruction. The captured call site is stored in the kernel object map so that the type can be determined during offline source code analysis.

The address and size of objects being allocated or deallocated can be derived from the arguments and return value. For an allocation function, the size is typically given as a function argument and the memory address as the return value. For a deallocation function, the address is typically given as a function argument. These values can be determined by the VMM by leveraging *function call conventions*.[2] Function arguments are delivered through the stack or registers, and LiveDM captures them by inspecting these locations at the entry of memory allocation/deallocation calls. To capture the return value, we need to determine where the return value is stored and when it is stored there. Integers up to 32-bits as well as 32-bit pointers are delivered via the `EAX` register and all values that we would like to capture are either of those types. The return value is available in this register when the allocation function returns to the caller. To capture the return values at the correct time the VMM uses a virtual stack. When a memory allocation function is called, the return address is extracted and pushed on to this stack. When the address of the code to be executed matches the return address on the stack, the VMM intercedes and captures the return value from the `EAX` register.

[2] A function call convention is a scheme to pass function arguments and a return value. We use the conventions for the x86 architecture and the `gcc` compiler [82].

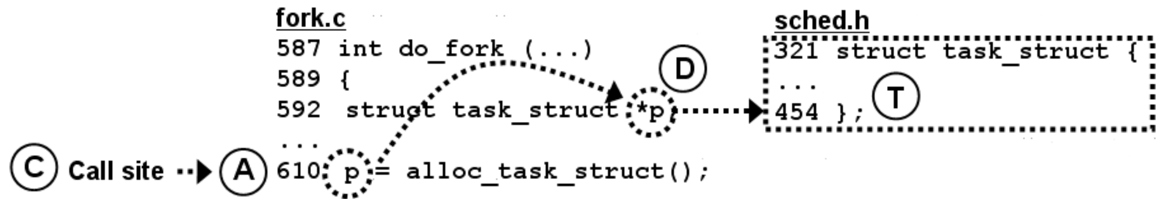


Figure 4.2.: A high level view of static code analysis

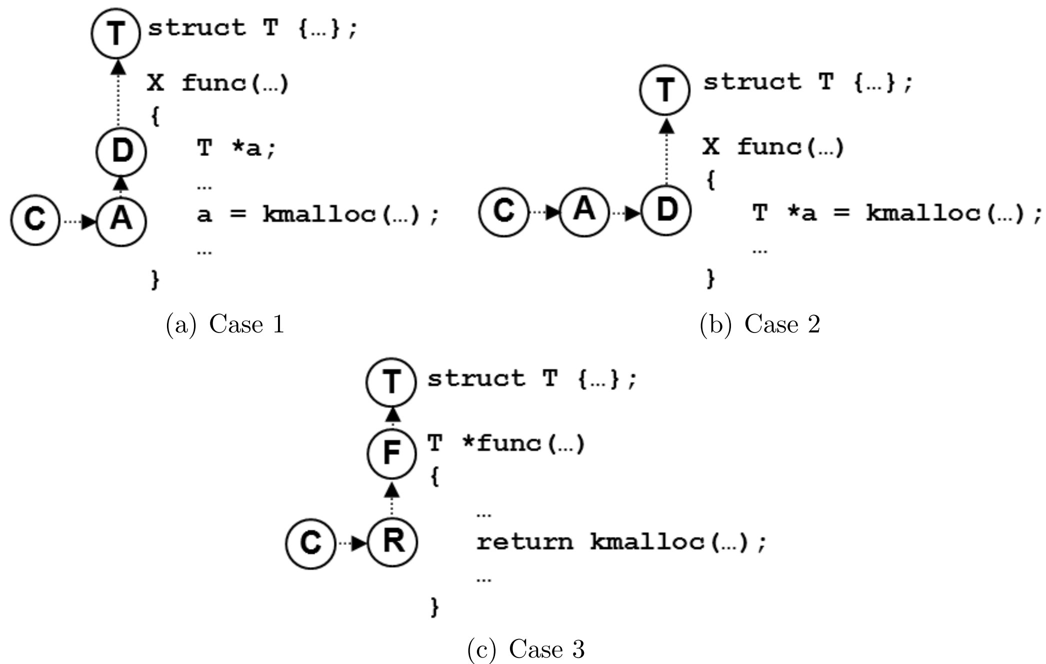


Figure 4.3.: Static code analysis. C: a call site, A: an assignment, D: a variable declaration, T: a type definition, R: a return, and F: a function declaration.

Dynamic Data Type Inference

The object type information related to kernel memory allocation events is determined using static analysis of the kernel source code offline. Figure 4.2 illustrates a high level view of our method. First, the allocation call site (C) of a dynamic object is mapped to the source code `fork.c:610` using debugging information found in the kernel binary. This code assigns the address of the allocated memory to a pointer variable at the left-hand side (LHS) of the assignment statement (A). As this variable's type can represent the type of the allocated memory, it is derived by traversing

the declaration of this pointer (D) and the definition of its type (T). Specifically, during the compilation of kernel source code, a parser sets the dependencies among the internal representations (IRs) of such code elements. Therefore, the type can be found by following the dependencies of the generated IRs.

For type resolution, we enumerate several patterns in the allocation code as shown in Figure 4.3(a), 4.3(b), and 4.3(c). Case 1 is the typical pattern (C→A→D→T) as just explained. In Case 2, the definition (D) and allocation (A) occur in the same line. The handling of this case is similar to that of Case 1. Case 3, however, is unlike the first two cases. The pattern in Case 3 does not use a variable to handle the allocated memory address, rather it directly returns the value generated from the allocation call. When a call site (C) is converted to a return statement (R), we determine the type of the allocated memory using the type of the returning function (F). In Figure 4.3(c), this pattern is presented as C→R→F→T.

Prior to static code analysis, we generate the set of information about these code elements to be traversed (i.e., C, A, D, R, F, and T) by compiling the kernel source code with the compiler that we instrumented (Section 4.3).

4.3 Implementation

Allocation-driven mapping is general enough to work with an OS that follows the standard function call conventions (e.g., Linux, Windows, etc.). Our prototype, LiveDM, supports three off-the-shelf Linux OSes of different kernel versions: Fedora Core 6 (Linux 2.6.18), Debian Sarge (Linux 2.6.8), and Redhat 8 (Linux 2.4.18).

LiveDM can be implemented on any software virtualization system, such as VMware (Workstation and Player) [83], VirtualBox [84], and Parallels [85]. We choose the QEMU [86] with KQEMU optimizer for implementation convenience.

In the kernel source code, many wrappers are used for kernel memory management, some of which are defined as macros or inline functions and others as regular functions. Macros and inline functions are resolved as the core memory function calls at compile

time by a preprocessor; thus, their call sites are captured in the same way as core functions. However, in the case of regular wrapper functions, the call sites will belong to the wrapper code.

To solve this problem, we take two approaches. If a wrapper is used only a few times, we consider that the type from the wrapper can indirectly imply the type used in the wrapper's caller because of its limited use. If a wrapper is widely used in many places (e.g., `kmem_cache_alloc` – a slab allocator), we treat it as a memory allocation function. Commodity OSes, which have mature code quality, have a well defined set of memory wrapper functions that the kernel and driver code commonly use. In our experience, capturing such wrappers, in addition to the core memory functions, can cover the majority of the memory allocation and deallocation operations.

We categorize the captured functions into four classes: (1) page allocation/free functions, (2) `kmalloc/kfree` functions, (3) `kmem_cache_alloc/free` functions (slab allocators), and (4) `vmalloc/vfree` functions (contiguous memory allocators). These sets include the well defined wrapper functions as well as the core memory functions. In our prototype, we capture about 20 functions in each guest kernel. The memory functions of an OS kernel can be determined from its design specification (e.g., the Linux Kernel API) or kernel source code.

Automatic translation of a call site to a data type requires a kernel binary that is compiled with a debugging flag (e.g., `-g` to `gcc`) and whose symbols are not stripped. Modern OSes, such as Ubuntu, Fedora, and Windows, generate kernel binaries of this form. Upon distribution, typically the stripped kernel binaries are shipped; however, unstripped binaries (or symbol information in Windows) are optionally provided for kernel debugging purposes. The experimented kernels of Debian Sarge and Redhat 8 are not compiled with this debugging flag. Therefore, we compiled the distributed source code and generated the debug-enabled kernels. These kernels share the same source code with the distributed kernels, but the offset of the compiled binary code can be slightly different because of the additional debugging information.

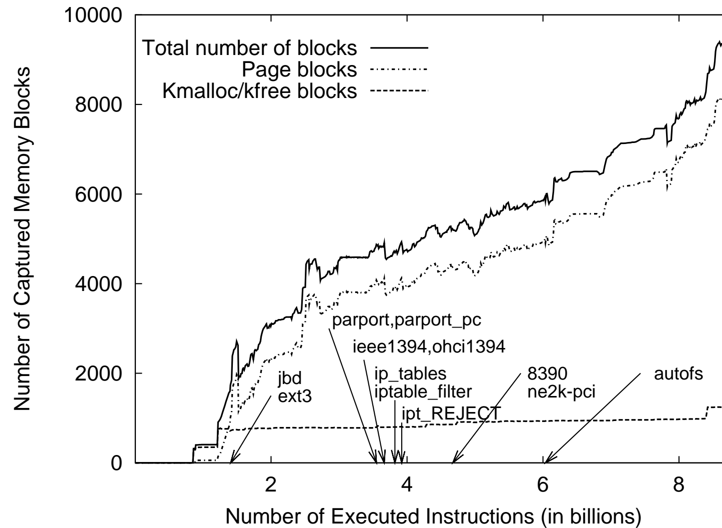


Figure 4.4.: The usage of dynamic kernel objects during the booting stage (OS: Redhat 8).

For static analysis we use a `gcc` [82] compiler (version 3.2.3) that we instrumented to generate internal representations for the source code of the experimented kernels. We place hooks in the parser to extract the abstract syntax trees for the code elements necessary in the static code analysis.

4.4 Evaluation

In this section, we evaluate the basic functionality of LiveDM with respect to the identification of kernel objects, casting code patterns, and the performance of allocation-driven mapping. The guest systems are configured with 256MB RAM and the host machine has a 3.2Ghz Pentium D CPU and 2GB of RAM.

4.4.1 Runtime Tracking of Dynamic Kernel Objects

LiveDM synchronously identifies dynamic kernel objects on their allocations and deallocations. Therefore unlike other kernel memory mapping approaches that sample memory status, LiveDM can continuously track changes in kernel memory status.

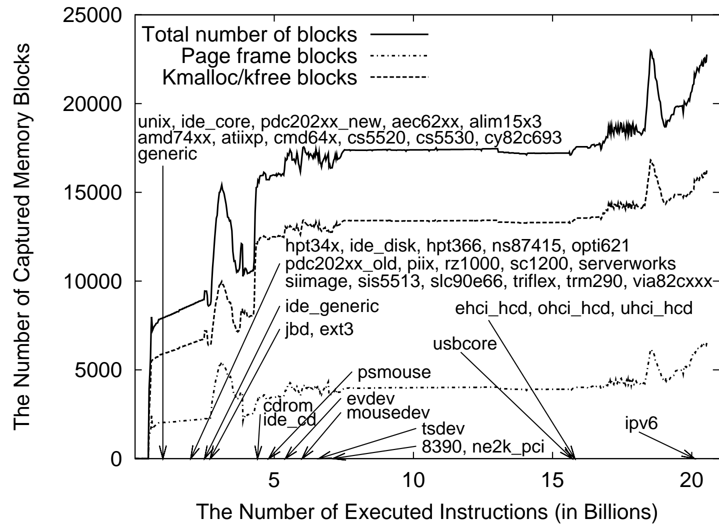


Figure 4.5.: The usage of dynamic kernel objects during the booting stage (OS: Debian Sarge).

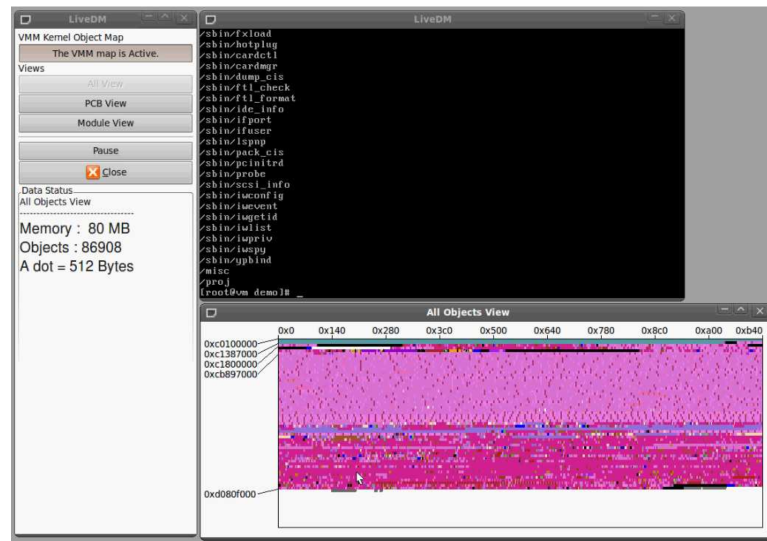


Figure 4.6.: LiveDM identifies kernel objects and generates a kernel object map at runtime.(OS: Redhat 8)

Figure 4.4 and 4.5 present the statistics of runtime dynamic kernel objects during the booting stage in two Linux operating systems. Figure 4.6 illustrates the GUI interface of our prototype implementation. The black screen at the top shows the

Table 4.1: A list of core dynamic kernel objects and the source code elements used to derive their data types in static analysis. (OS: Debian Sarge).

	Call Site	Declaration	Data Type	Case	#Objects
Task/Sig	kernel/fork.c:248	kernel/fork.c:243	task_struct	1	66
	kernel/fork.c:801	kernel/fork.c:795	sighand_struct	1	63
	fs/exec.c:601	fs/exec.c:587	sighand_struct	1	1
	kernel/fork.c:819	kernel/fork.c:813	signal_struct	1	66
Memory	arch/i386/mm/pgtable.c:229	arch/i386/mm/pgtable.c:229	pgd_t	2	54
	kernel/fork.c:433	kernel/fork.c:431	mm_struct	1	47
	kernel/fork.c:559	kernel/fork.c:526	mm_struct	1	7
	kernel/fork.c:314	kernel/fork.c:271	vm_area_struct	1	149
	mm/mmap.c:923	mm/mmap.c:748	vm_area_struct	1	1004
	mm/mmap.c:1526	mm/mmap.c:1521	vm_area_struct	1	5
	mm/mmap.c:1722	mm/mmap.c:1657	vm_area_struct	1	48
	fs/exec.c:402	fs/exec.c:342	vm_area_struct	1	47
File system	kernel/fork.c:677	kernel/fork.c:654	files_struct	1	54
	kernel/fork.c:597	kernel/fork.c:597	fs_struct	2	53
	fs/file_table.c:76	fs/file_table.c:69	file	1	531
	fs/buffer.c:3062	fs/buffer.c:3062	buffer_head	2	828
	fs/block_dev.c:232	fs/block_dev.c:232	bdev_inode	2	5
	fs/dcache.c:692	fs/dcache.c:689	dentry	1	4203
	fs/inode.c:112	fs/inode.c:107	inode	1	1209
	fs/namespace.c:55	fs/namespace.c:55	vfsmount	2	16
	fs/proc/inode.c:93	fs/proc/inode.c:90	proc_inode	1	237
	drivers/block/ll_rw_blk.c:1405	drivers/block/ll_rw_blk.c:1405	request_queue_t	2	18
	drivers/block/ll_rw_blk.c:2950	drivers/block/ll_rw_blk.c:2945	io_context	1	10
Network	net/socket.c:279	net/socket.c:278	socket_alloc	1	12
	net/core/sock.c:617	net/core/sock.c:613	sock	1	3
	net/core/dst.c:125	net/core.dst.c:119	dst_entry	1	5
	net/core/neighbour.c:265	net/core/neighbour.c:254	neighbour	1	1
	net/ipv4/tcp_ipv4.c:134	net/ipv4/tcp_ipv4.c:133	tcp_bind_bucket	2	4
	net/ipv4/fib_hash.c:586	net/ipv4/fib_hash.c:461	fib_node	1	9

guest operating system. The kernel object map is illustrated below this screen. The statistics of active kernel objects are shown in the left pane.

4.4.2 Identifying Dynamic Kernel Objects

To demonstrate the ability of LiveDM to inspect the runtime status of an OS kernel, we present a list of important kernel data structures captured during the execution of Debian Sarge OS in Table 4.1. These data structures manage the key OS status such as process information, memory mapping of each process, and the status of file systems and network which are often targeted by kernel malware and kernel bugs [23–25, 42, 44, 60, 61, 87]. Kernel objects are recognized using allocation call sites shown in column **Call Site** during runtime. Using static analysis, this information

is translated into the data types shown in column **Data Type** by traversing the allocation code and the declaration of a pointer variable or a function shown in column **Declaration**. Column **Case** shows the kind of the allocation code pattern described in Section 4.2.2. The number of the identified objects for each type in the inspected runtime status is presented in column **#Objects**. At that time instance, LiveDM identified total of 29488 dynamic kernel objects with their data types derived from 231 allocation code positions.

To evaluate the accuracy of the identified kernel objects, we build a reference kernel where we modify kernel memory functions to generate a log of dynamic kernel objects and run this kernel in LiveDM. We observe that the dynamic objects from the log accurately match the live dynamic kernel objects captured by LiveDM. To check the type derivation accuracy, we manually translate the captured call sites to data types by traversing kernel source code as done by related approaches [41, 70]. The derived types at the allocation code match the results from our automatic static code analysis.

4.4.3 Code Patterns Casting Objects from Generic Types to Specific Types

In Section 4.2.1, we discussed that allocation-driven mapping has no problem handling the situation where a specific type is cast to a generic type, but casting from generic types to specific types can be a problem. To estimate how often this type of casting occurs, we manually checked all allocation code positions where the types of kernel objects are derived for the inspected status. We checked for the code pattern that memory is allocated using a generic pointer and then the address is cast to the pointer of a more specific type. Note that this pattern does not include the use of generic pointers for generic purposes. For example, the use of void or integer pointers for bit fields or buffers is a valid use of generic pointers. Another valid use is kernel memory functions that internally handle pre-typed memory using generic pointers to redefine it to various types. We found 25 objects from 10 allocation code

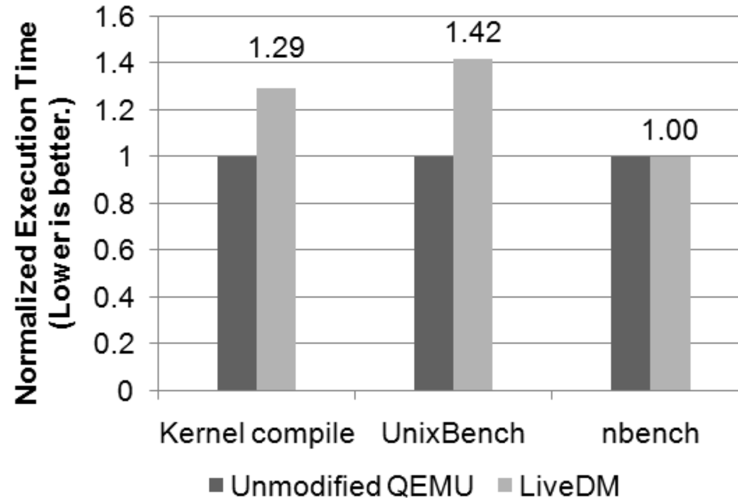


Figure 4.7.: Performance of LiveDM for Linux 2.4 (OS: Redhat 8)

positions (e.g., `tty_register_driver` and `vc_allocate`) exhibiting this behavior at runtime. Such objects are not part of the core data structures shown in Table 4.1, and they account for only 0.085% of all objects. Hence we consider them as non-significant corner cases. As the code positions where this casting occurs are available to LiveDM, we believe that the identification of this behavior and the derivation of a specific type can be automated by performing static analysis on the code after the allocation code.

4.4.4 Performance of Allocation-driven Mapping

As LiveDM is mainly targeted for non-production environments such as honeypots and kernel debugging systems, performance is not a primary concern. Still, we would like to provide a general idea of the cost of allocation-driven mapping. To measure the overhead to generate a kernel object map at runtime, we ran three benchmarks: compiling the kernel source code, UnixBench (Byte Magazine Unix Benchmark 5.1.2), and nbench (BYTEmark* Native Mode Benchmark version 2). The normalized runtime overhead of our implementation is presented in Figure 4.7 and Figure 4.8. Compared to unmodified QEMU, our prototype incurs (in the worst case) 41.77% overhead for

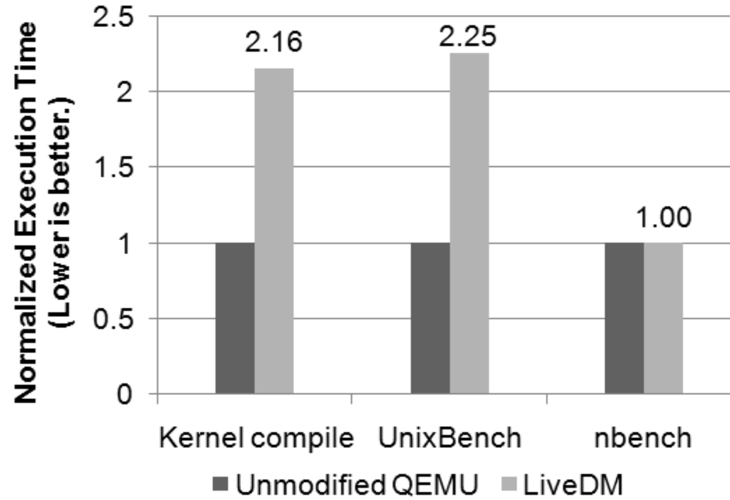
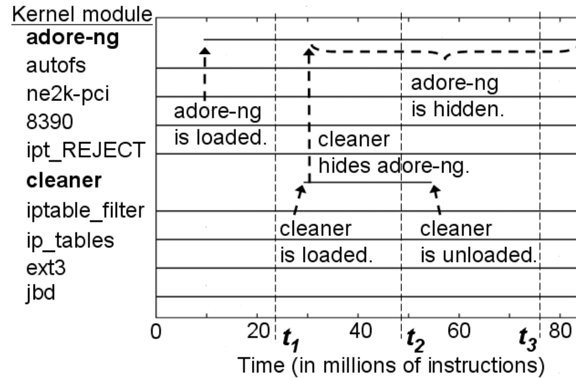


Figure 4.8.: Performance of LiveDM for Linux 2.6 (OS: Debian Sarge)

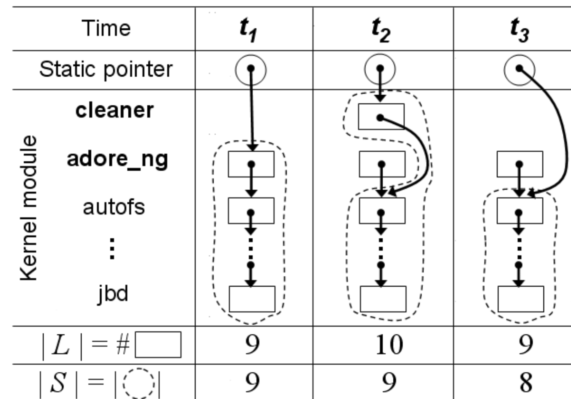
Redhat 8 (Linux 2.4) and 125.47% overhead for Debian Sarge (Linux 2.6). For CPU intensive workload such as nbench, the overhead is near zero because the VMM rarely intervenes. However, applications that use kernel services requiring dynamic kernel memory have higher overhead. As a specific example, compiling the Linux kernel exhibited an overhead of 29% for Redhat 8 and 115.69% for Debian Sarge. It is important to note that these numbers measure overhead when compared to an unmodified VMM. Software based virtualization will add additional overhead as well. For the purpose of inspecting fine-grained kernel behavior in non-production environments, we consider this overhead acceptable. The effects of overhead can even be minimized in a production environment by using decoupled analysis [88].

4.5 Hidden Kernel Object Detector

One problem with static type-projection approaches is that they are not able to detect dynamic kernel object manipulation without some sort of data invariant. In this section we present a hidden kernel object detector built on top of LiveDM that does not suffer from this limitation.



(a) Temporal live status of kernel modules based on allocation-driven mapping.



(b) Live set (L) and scanned set (S) for kernel modules at t_1, t_2, and t_3.

Figure 4.9.: Illustration of the kernel module hiding attack by `cleaner` rootkit. Note that the choice of t_1, t_2, and t_3 is for the convenience of showing data status and irrelevant to the detection. This attack is detected based on the difference between L and S.

4.5.1 Leveraging the Un-tampered View

Some advanced DKOM-based kernel rootkits hide kernel objects by simply removing all references to them from the kernel's dynamic memory. We model the behavior of this type of DKOM data hiding attack as a data anomaly in a list. If a dynamic kernel object does not appear in a kernel object list, then it is orphaned and hence an anomaly. As described in Section 4.2.1, allocation-driven mapping provides an un-tampered view of the kernel objects not affected by manipulation of the actual

kernel memory content. Therefore, if a kernel object appears in the LiveDM-generated kernel object map but cannot be found by traversing the kernel memory, then that object has been hidden. More formally, for a set of dynamic kernel objects of a given data type, a live set L is the set of objects found in the kernel object map. A scanned set S is the set of kernel objects found by traversing the kernel memory as in the related approaches [40, 41, 61]. If L and S do not match, then a data anomaly will be reported.

This process is illustrated in the example of the `cleaner` rootkit that hides the `adore-ng` rootkit module (Figure 4.9). Figure 4.9(a) presents the timeline of this attack using the lifetime of kernel modules. Figure 4.9(b) illustrates the detailed status of kernel modules and corresponding L and S at three key moments. Kernel modules are organized as a linked list starting from a static pointer variable. When the `cleaner` module is loaded after the `adore-ng` module, it modifies the linked list to bypass the `adore-ng` module entry (shown at t_2). Therefore, when the `cleaner` module is unloaded, the `adore-ng` module disappears from the module list (t_3). At this point in time the scanned set S based on static type-projection mapping has lost the hidden module, but the live set L keeps the view of all kernel modules alive. Therefore, the monitor can detect a hidden kernel module because of the condition, $|L| \neq |S|$.

4.5.2 Detecting DKOM Data Hiding Attacks

There are two dynamic kernel data lists which are favored by rootkits as attack targets: the kernel module list and the process control block (PCB) list.[3] However other linked list-based data structures can be similarly supported as well. The basic procedure is to generate the live set L and periodically generate and compare with the scanned set S. We tested 8 real-world rootkits and 2 of our own rootkits (`linuxfu` and `fuuld`) previously used in [44, 76, 89], and these rootkits commonly hide kernel

[3] A process control block (PCB) is a kernel data structure containing administrative information for a particular process. Its data type in Linux is `task_struct`.

Table 4.2: DKOM data hiding rootkit attacks that are automatically detected by comparing LiveDM-generated view (L) and kernel memory view (S).

Rootkit Name	$\lvert L\rvert$ - $\lvert S\rvert$	**Manipulated Data**		**Operating System**	**Attack Vector**
		Type	**Field**		
hide_lkm	# of hidden modules	module	next	Redhat 8	/dev/kmem
fuuld	# of hidden PCBs	task_struct	next_task, prev_task	Redhat 8	/dev/kmem
cleaner	# of hidden modules	module	next	Redhat 8	LKM
modhide	# of hidden modules	module	next	Redhat 8	LKM
hp 1.0.0	# of hidden PCBs	task_struct	next_task, prev_task	Redhat 8	LKM
linuxfu	# of hidden PCBs	task_struct	next_task, prev_task	Redhat 8	LKM
modhide1	1 (rootkit self-hiding)	module	next	Redhat 8	LKM
kis 0.9 (server)	1 (rootkit self-hiding)	module	next	Redhat 8	LKM
adore-ng-2.6	1 (rootkit self-hiding)	module	list.next, list.prev	Debian Sarge	LKM
ENYELKM 1.1	1 (rootkit self-hiding)	module	list.next, list.prev	Debian Sarge	LKM

objects by directly manipulating the pointers of such objects. LiveDM successfully detected all these attacks based on the data anomaly from kernel memory maps and the results are shown in Table 4.2.

In the experiments, we focus on a specific attack mechanism – data hiding via DKOM – rather than the attack vectors – how to overwrite kernel memory – or other attack features of rootkits for the following reason. There are various attack vectors including the ones that existing approaches cannot handle and they can be easily utilized. Specifically, we acknowledge that the rootkits based on a loadable kernel module (LKM) can be detected by code integrity approaches [10, 11] with the white listing scheme of kernel modules. However, there exist alternate attack vectors such as `/dev/mem`, `/dev/kmem` devices, return-oriented techniques [14, 16], kernel bugs, and unproven code in third-party kernel drivers that can elude existing kernel rootkit detection and prevention approaches. We present the DKOM data hiding cases of LKM-based rootkits as part of our results because these rootkits can be easily converted to make use of these alternate attack vectors.

We also include results for two other rootkits that make use of these advanced attack techniques. `hide_lkm` and `fuuld` in Table 4.2 respectively hide kernel modules and processes without any kernel code integrity violation (via `/dev/kmem`) purely based on DKOM, and current rootkit defense approaches cannot properly detect these attacks. However, our monitor effectively detects all DKOM data hiding attacks regardless of attack vectors by leveraging the LiveDM-generated kernel object map. Allocation-driven mapping can uncover the hidden object even in more adversarial scenarios. For example, if a simple linked list having no data invariant is directly manipulated without violating kernel code integrity, LiveDM will still be able to detect such an attack and uncover the specific hidden object.

In the experiments that detect rootkit attacks, we generate and compare L and S sets every 10 seconds. When a data anomaly occurs, the check is repeated in 1 second. (The repeated check ensures that a kernel data structure was not simply in

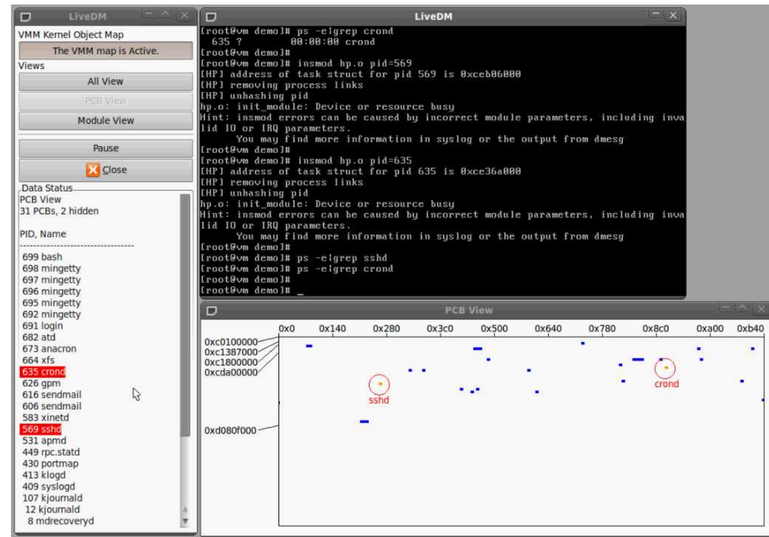


Figure 4.10.: LiveDM detects process hiding rootkit attacks and pinpoints hidden processes.

an inconsistent state during the first scan.) If the anomaly persists, then we consider it as a true positive.

To ensure that the detected cases are not caused by memory leaks, we generated the hash values for the memory corresponding to the hidden objects after the violation of the lifetime invariant is detected. The hash values for the hidden modules and PCBs are being changed which confirm that the detected cases are not memory leaks.

With these monitoring policies, we successfully detected all tested DKOM hiding attacks without any false positives or false negatives.

We note that while this section focuses on data hiding attacks based on DKOM, data hiding attacks without manipulating data (such as rootkit code that filters system call results) may also be detected using the LiveDM system. Instead of comparing the un-tampered LiveDM-generated view with the scanned view of kernel memory, one could simply compare the un-tampered view with the user-level view of the system.

Figure 4.10 and 4.11 demonstrate how our prototype implementation detects data hiding rootkit attacks. Figure 4.10 demonstrates the attack of `hp` rootkit. This rootkit hides two processes, `crond` and `sshd`. LiveDM systematically detects such hidden

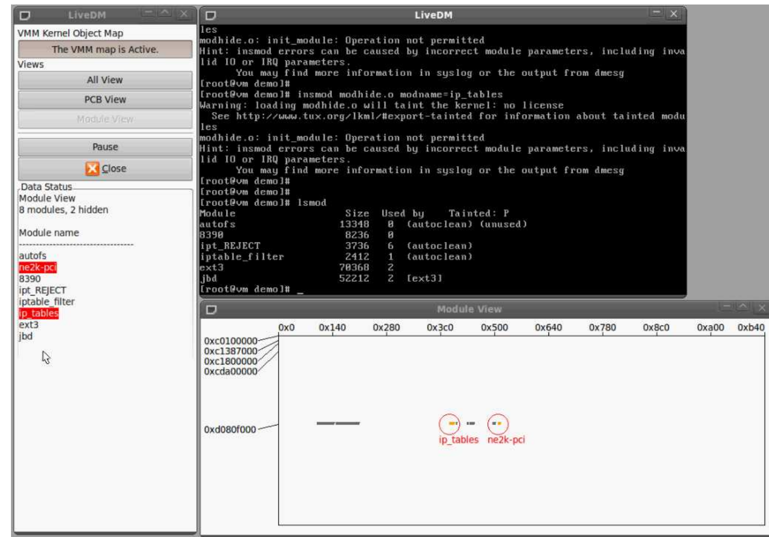


Figure 4.11.: LiveDM detects kernel driver hiding rootkit attacks and pinpoints hidden drivers.

processes and pinpoints them in the list of processes at the left pane and in the map of processes. Similarly Figure 4.11 shows the detection of `modhide` rootkit attacks. Two kernel drivers, `ip_tables` and `ne2k-pci`, are hidden via pointer manipulation. Those hidden drivers are successfully detected by LiveDM and shown in the driver list and in the map of kernel drivers.

4.6 Temporal Kernel Malware Analysis

Kernel rootkit analysis approaches based on dynamic type-projection are able to perform temporal analysis of a running rootkit. One problem with these approaches, however, is that they are only able to track malware actions that occur from injected rootkit code. If a rootkit modifies memory indirectly through other means such as legitimate kernel functions or kernel bugs, these approaches are unable to follow the attack. Allocation-driven mapping does not share this weakness. To further illustrate the strength of allocation-driven mapping, we built a temporal malware behavior

monitor (called a temporal monitor or a monitor below for brevity) that uses a kernel object map in temporal analysis of a kernel execution trace.

In this section, we highlight two features that allocation-driven mapping newly provides. First, allocation-driven mapping enables the *use of a kernel object map covering all kernel objects in temporal analysis*; therefore for any given dynamic kernel object we can inspect how it is being used in the dynamic kernel execution trace regardless of the accessing code (either legitimate or malicious), which is difficult for both static and dynamic type-projection approaches. Second, the data lifetime in allocation-driven mapping lets the monitor *avoid the dynamic data identity problem* (Section 2.5.1) which can be faced by an asynchronous memory map.

4.6.1 Systematic Visualization of Malware Influence via Dynamic Kernel Memory

Our monitor systematically inspects and visualizes the influence of kernel malware attacks targeting dynamic kernel memory. To analyze this dynamic attack behavior, we generate a full system trace including the kernel object map status, the executed code, and the memory accesses during the experiments of kernel rootkits. When a kernel rootkit attack is launched, if it violates kernel code integrity, the rootkit code is identified by using our previous work, NICKLE [10]. Then the temporal monitor systematically identifies all targets of rootkit memory writes by searching the kernel object map. If the attack does not violate code integrity, the proposed technique in the previous section or any other approach can be used to detect the dynamic object under attack. The identified objects then become the *causes* of malware behavior and their *effects* are systematically visualized by searching the original and the modified kernel control flow triggered by such objects. For each object targeted by the rootkit, there are typically multiple behaviors using its value. Among those, this monitor samples a pair of behaviors caused by the same code, the latest one before the attack and the earliest one after the attack, and presents them for a comparison.

Table 4.3: The list of kernel objects manipulated by `adore-ng` rootkit. (OS: Redhat 8).

Runtime Identification		Offline Data Type Interpretation	
Call Site	**Offset**	**Type / Object (Static, Module object)**	**Field**
fork.c:610	0x4,12c,130	task_struct (Case (1))	flags,uid,euid
fork.c:610	0x134,138,13c	task_struct (Case (1))	suid,fsuid,gid
fork.c:610	0x140,144,148	task_struct (Case (1))	egid,sgid,fsgid
fork.c:610	0x1d0	task_struct (Case (1))	cap_effective
fork.c:610	0x1d4	task_struct (Case (1))	cap_inheritable
fork.c:610	0x1d8	task_struct (Case (1))	cap_permitted
generic.c:436	0x20	proc_dir_entry (Case (2))	get_info
(Static object)		proc_root_inode_operations	lookup
(Static object)		proc_root_operations	readdir
(Static object)		unix_dgram_ops	recvmsg
(Module object)		ext3_dir_operations	readdir
(Module object)		ext3_file_operations	write

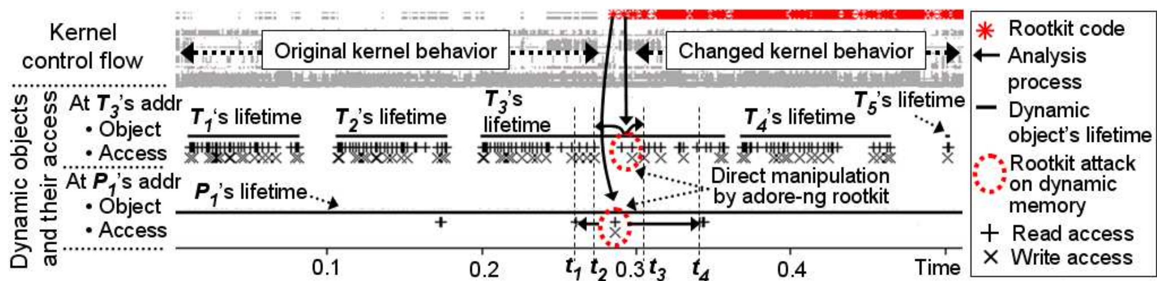


Figure 4.12.: Kernel control flow (top) and the usage of dynamic memory (below) at the addresses of T_3 (Case (1)) and P_1 (Case (2)) manipulated by the `adore-ng` rootkit. Time is in billions of kernel instructions.

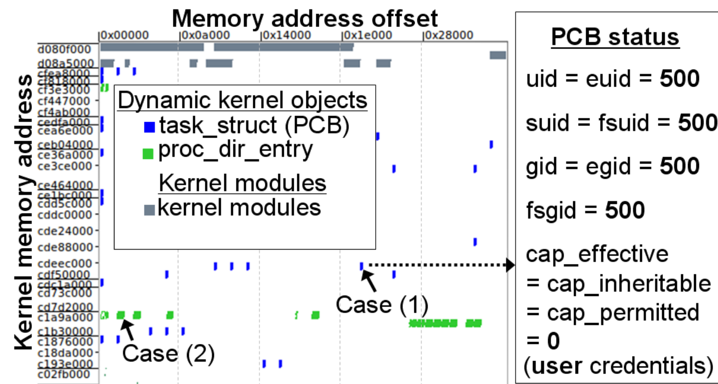
As a running example in this section, we will present the analysis of the attacks by the `adore-ng` rootkit. This rootkit is chosen because of its advanced malware behavior triggered by dynamic objects; and other rootkits can be analyzed in a similar way. Table 4.3 lists the kernel objects that the `adore-ng` rootkit tampers with. In particular, we focus on two specific attack cases using dynamic objects: (1) The first case is the manipulation of a PCB (T_3) for privilege escalation and (2) the second case is the manipulation of a function pointer in a dynamic `proc_dir_entry` object (P_1) to hijack kernel control flow. Figure 4.12 presents a detailed view of kernel control flow and the usage of the targeted dynamic kernel memory in the attacks. The X axis shows the execution time, and kernel control flow is shown at top part of this figure.

The space below shows the temporal usage of dynamic memory at the addresses of T_3 and P_1 before and after rootkit attacks. Thick horizontal lines represent the lifetime of kernel objects which are temporally allocated at such addresses. $+$ and $\times$ symbols below such lines show the read and write accesses on corresponding objects. The aforementioned analysis process is illustrated as solid arrows. From the times when T_3 and P_1 are manipulated (shown as dotted circles), the monitor scans the execution trace backward and forward to find the code execution that consumes the values read from such objects (i.e., $+$ symbols).

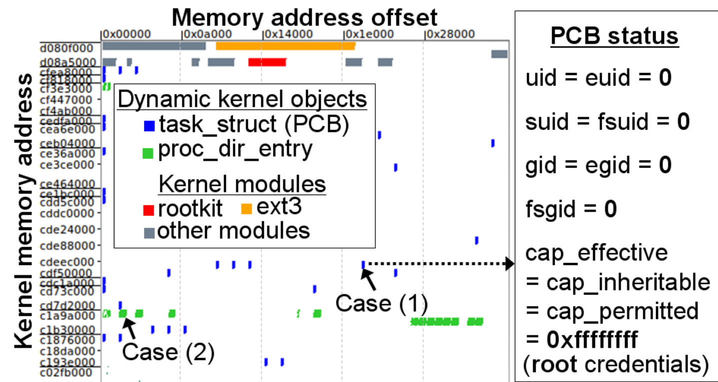
4.6.2 Selecting Semantically Relevant Kernel Behavior Using Data Lifetime

Our monitor inspects dynamic memory states in the temporal execution trace and as such we face the *dynamic data identity problem* described in Section 4.2.1. The core of the problem is that one memory address may correspond with multiple objects over a period of time. This problem can be solved if the lifetime of the inspected object is available because the monitor can filter out irrelevant kernel behaviors triggered by other kernel objects that share the same memory address. For example, in Figure 4.12, we observe the memory for T_3 is used for four other PCBs (i.e., T_1, T_2, T_4, and T_5) as well in the history of kernel execution. Simply relying on the memory address to analyze the trace can lead to finding kernel behavior for *all five PCBs*. However, the monitor limits the inspected time range to the lifetime of T_3 and select only semantically relevant behaviors to T_3. Consequently it can provide a reliable inspection of runtime behavior only relevant to attacks.

Other kernel memory mapping approaches commonly cannot handle this problem properly. In static type-projection, when two kernel objects from different snapshots are given we cannot determine whether they represent the same data instance or not, even though their status is identical because such objects may or may not be different data instances depending on whether memory allocation/deallocation events occur between the generation of such snapshots. Dynamic type-projection mapping



(a) The original data view at t_2.



(b) The manipulated data view at t_3.

Figure 4.13.: Kernel data view before and after the `adore-ng` rootkit attack.

is only based on malware instructions, and thus does not have information about allocation and deallocation events which occur during legitimate kernel execution.

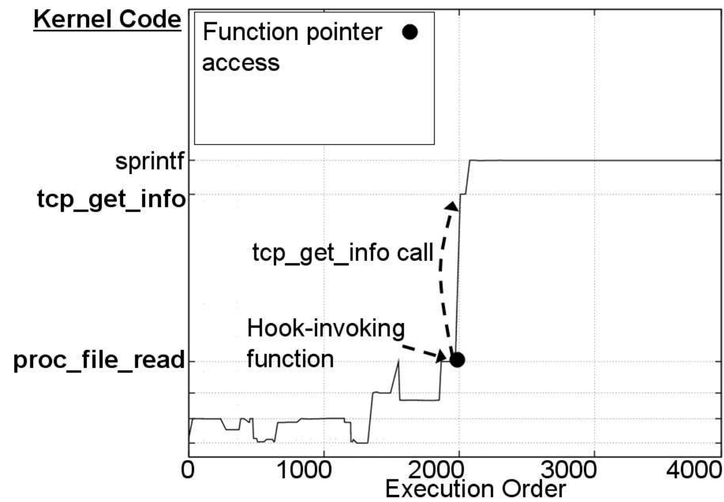
4.6.3 Case (1): Privilege Escalation Using Direct Memory Manipulation

To demonstrate the effectiveness of our temporal monitor we will discuss two specific attacks employed by `adore-ng`. The first is a privilege escalation attack that works by modifying the user and group ID fields of the PCB. The PCB is represented by T_3 in Figure 4.12. To present the changed kernel behavior from the manipulation of T_3, the temporal monitor finds the latest use of T_3 before the attack (at t_2) and the earliest use of it after the attack (at t_3). The data views at such times are presented

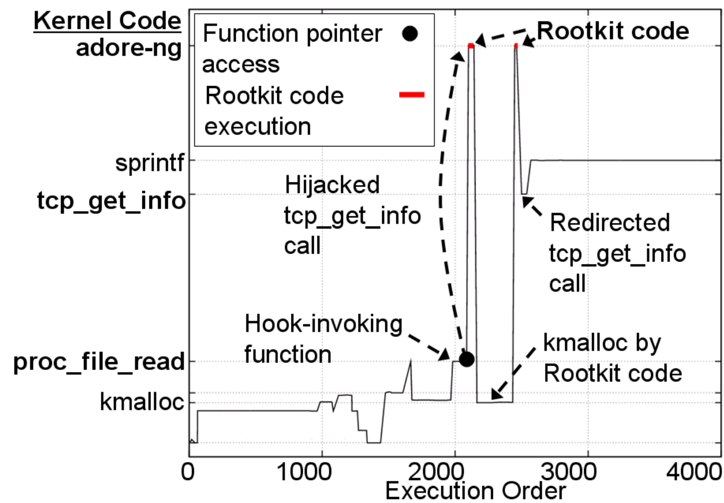
in Figure 4.13(a) and 4.13(b) as 2-dimensional memory maps where a kernel memory address is represented as the combination of the address in the Y axis and the offset in the X axis. These views present kernel objects relevant to this attack before and after the attack. The manipulated PCB is marked with "Case (1)" in the views and the values of its fields are shown in the box on the right side of each view (PCB status). These values reveal a stealthy rootkit behavior that changes the identity of a user process by directly patching its PCB (DKOM). Before the attack (Figure 4.13(a)), the PCB has the credentials of an ordinary user whose user ID is 500. However, after the attack, Figure 4.13(b) shows the credentials of the root user. This direct transition of its status between two accounts is abnormal in conventional operating system environments. `su` or `sudo` allow privileged operations by forking a process to retain the original identity. Hence we determine that this is a case of privilege escalation that illegally enables root privileges for an ordinary user.

4.6.4 Case (2): Dynamic Kernel Object Hooking

The next `adore-ng` attack hijacks kernel code execution by modifying a function pointer and this attack is referred to as Kernel Object Hooking (KOH) [80]. This behavior is observed when the influence of a manipulated function pointer in P_1 (see Figure 4.12) is inspected. To select only the behaviors caused by this object, the monitor guides the analysis to the lifetime of P_1. The temporal monitor detects several behaviors caused by reading this object and two samples are chosen among those to illustrate the change of kernel behavior by comparison: the latest original behavior before the attack (at t_1) and the earliest changed behavior after the attack (at t_4). The monitor generates two kernel control flow graphs at these samples, each for a period of 4000 instructions. Figure 4.14(a) and 4.14(b) present how this manipulated function pointer affects runtime kernel behavior. The Y axis presents kernel code; thus, the fluctuating graphs show various code executed at the corresponding time of X axis. A hook-invoking function (`proc_file_read`) reads the function pointer and



(a) The original control flow at t_1.



(b) The hijacked control flow at t_4.

Figure 4.14.: Kernel control flow view before and after the `adore-ng` rootkit attack.

calls the hook code pointed to by it. Before the rootkit attack, the control flow jumps to a legitimate kernel function `tcp_get_info` which calls `sprintf` after that as shown in Figure 4.14(a). However, after the hook is hijacked, the control flow is redirected to the rootkit code which calls `kmalloc` to allocate its own memory, then comes back to the original function (Figure 4.14(b)).

4.7 Summary

In this chapter, we presented allocation-driven mapping, a kernel memory mapping scheme, and its implementation – LiveDM. By capturing the kernel objects' allocation and deallocation events, our scheme provides an un-tampered view of kernel objects that will not be affected by kernel malware's manipulation of kernel memory content. The LiveDM-generated kernel object map accurately reflects the status of dynamic kernel memory and tracks the lifetimes of dynamic kernel objects. This temporal property is highly desirable in temporal kernel execution analysis where both kernel control flow and dynamic memory status can be analyzed in an integrated fashion. We have demonstrated the effectiveness of the LiveDM system by developing a hidden kernel object detector and a temporal malware behavior monitor and applying them to a corpus of kernel rootkits.

5 CHARACTERIZING KERNEL MALWARE BEHAVIOR WITH KERNEL DATA ACCESS PATTERNS

In this chapter, we demonstrate the second component of DMDA, characterization of kernel malware behavior with kernel data access patterns. Based on the kernel object mapping system presented in the previous chapter, the memory access patterns specific to malware attacks are determined and matched to detect the kernel malware. Moreover we analyze common malware behavior in terms of memory access patterns to determine the applicability of our approach to malware variants.

5.1 Introduction

Characterizing malware behavior is a non-trivial research problem and there have been many approaches to address its challenges. A large body of work uses malware's control flow patterns, such as instruction sequences or system-call sequences, to detect or analyze malware [26–30]. In response to such approaches, malware often employs various obfuscation techniques to confuse malware analyzers [31–34]. Meanwhile, these approaches face challenges arising from execution dynamics, such as dynamic code paths and the impact of other system components (e.g., network latency and signals), which can cause variations in the characterized malware patterns. The situation is more complicated in the kernel space because operating system (OS) kernels have a highly dynamic workload, including interrupts, the coordination of user processes, and the management of low level resources (e.g., page tables).

For detection and prevention of kernel malware, there is another collection of work called the code integrity-based approach [10, 11]. This approach allows only authorized code for execution and considers any code outside the white list as malicious. Therefore, this approach is effective for kernel rootkits that introduce new code to

kernel space. However, other advanced rootkits perform the attacks by exploiting only legitimate kernel code (e.g., the usage of memory devices [22], kernel bugs, and return-oriented programming [90]); and such attacks are not properly handled by this approach. In addition, this approach authorizes kernel driver code based on policies trusting OS developers or venders without systematic examination of the code. For example, existing code integrity-based approaches [10, 11] allow the kernel text and a list of benign kernel modules included in the OS distributions. These policies do not provide safety from hidden malicious code inside the authorized code. Thus the capability of examining kernel drivers for potentially malicious behavior regardless of such policies is desirable.

In this chapter, we introduce an alternative approach that characterizes kernel malware behavior by using its data access patterns. We assume that when kernel malware tampers with core kernel data, there exist kernel data access patterns specific to the attacks. As such, we could take a subset of data access patterns that consistently appears in multiple kernel execution instances only when the malware is active and generate a malware signature using the subset.[1] These patterns under constraints neither include malware's temporal control flow information, nor the code-specific information about the malware. Therefore, this approach is less susceptible to obfuscation and more effective for matching malware variants.

To evaluate the effectiveness of our approach, we generated the signatures of three classic rootkits and matched them with benign kernel runs and malicious kernel runs where the rootkits are active. This experiment detects the presence of 16 kernel rootkits that have a variety of attack goals and mechanisms without triggering any false positives in typical benign workload. We further analyzed the data behavior of these rootkits and found that a majority of them exhibit shared behaviors. We argue that such common behavior can be used to effectively detect malware variants (e.g., polymorphic rootkits, different versions, and similar rootkits).

The contributions of this chapter are as follows:

[1] We use the terms "a kernel execution instance" and "a kernel run," to represent an instance of the OS kernel execution, which starts from its booting and ends at its shutdown.

- We present a complementary approach that characterizes kernel malware behavior by using its data access patterns specific to the attacks. This approach can be applied to detect kernel rootkits that do not violate kernel code integrity.
- This approach can automatically construct malware signatures by using a binary-only malware program. Malware behavior is extracted by capturing a subset of kernel behavior that consistently appears across kernel execution instances only when the malware is active.
- This signature uses data behavior with generalized code information and does not involve control flow of malware code execution. Hence it can detect the variants of kernel malware by exposing similar data behavior across kernel malware.

We have implemented a prototype called DataGene based on our approach. DataGene is mainly designed for non-production systems such as a honeypot and a malware analysis system. For instance, when a new proprietary driver is deployed, DataGene can inspect it for potential hidden malicious behavior similar to the behavior observed in existing kernel malware. If a newly distributed kernel malware sample shares any data behavior with existing kernel malware, DataGene can detect it and extract its data behavior. It can be further used to detect this malware and its variants. In addition, DataGene can detect challenging kernel rootkits that do not violate kernel code integrity. Therefore, this data-centric approach can complement the code integrity-based approach in the defense against kernel malware.

5.2 Design of DataGene

In this section, we present the design of DataGene that characterizes the behavior of kernel malware and determines its presence based on data access patterns. As DataGene uses information regarding memory accesses, our design employs virtual

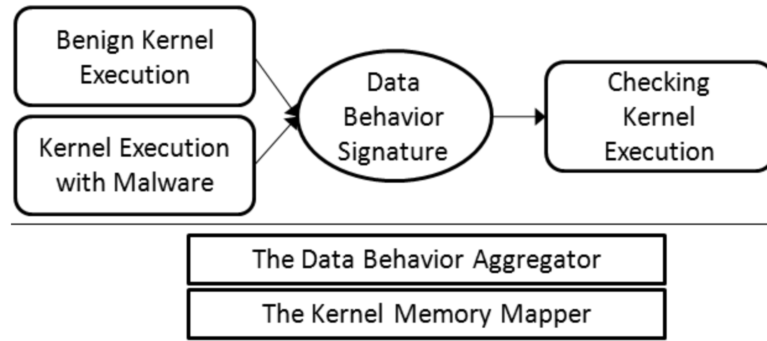


Figure 5.1.: Overview of DataGene.

machine techniques to capture the accesses. The overview of DataGene is presented in Figure 5.1, and the components of this system are as follows.

As a basic unit to represent the kernel's data behavior, DataGene generates a summary of the access patterns for all kernel objects accessed in a kernel execution instance. To identify dynamic kernel memory objects, this process takes advantage of a kernel memory mapping process (shown as The Kernel Memory Mapper in Figure 5.1). For each access on kernel memory in the guest OS, the virtual machine monitor (VMM) intercedes and records the information of the kernel memory access, such as the accessing code, the accessed memory type, and the accessed offset (The Data Behavior Aggregator).

To determine the malware behavior, the memory access patterns for two kinds of kernel execution instances are generated: benign kernel runs and malicious kernel runs where kernel malware is active. By taking the difference between the two sets of memory access patterns, we extract the data behavior specific to the kernel malware and generate its signature (Data Behavior Signature). To detect kernel malware, the generated signature is compared to the memory access patterns of a tested kernel execution instance (Checking Kernel Execution).

5.2.1 Data Behavior Profile Approach

In this section, we present basic terminologies that represent the memory access patterns of kernel execution.

Definition 5.2.1 (Data Behavior Element) *A data behavior element (DBE) represents a pattern of a memory access. It is defined as a quintuple, (c, o, m, i, f): the address of the code that accesses memory (c), the kind (read or write) of memory access (o), the kind (static or dynamic) of the accessed memory (m), the class of the accessed memory (i), and the accessed offset(s) (f) inside the memory of the class i.*

c is the address of the kernel code that reads or writes kernel memory. o represents the kind of memory access which is 0 for a memory read and 1 for a memory write.

The kind of accessed memory, m, is 0 for a dynamic object and 1 for a static object. The class i is defined differently, depending on m. Static objects are known at compile time; therefore, we are able to assign unique numbers as their identifiers. A class of a static object can represent either a static data object or a kernel function in the kernel text. In the case of dynamic kernel objects, there are multiple memory instances for the same data type at runtime. Dynamic kernel objects allocated by the same code correspond to the data instances of the specific data type used in the allocation code. Thus, we aggregate the access patterns of dynamic kernel objects that share the allocation code. The address of this code (called an allocation code site) is used as a unique class for such objects.

f is an offset, or a range of offsets, accessed by the code at c. We allow a range of offsets because if this object is an array, the accessed offsets can vary for the same accessing code. Handling them as separate data behavior elements can cause a high number of elements with slightly different offsets for the same accessing code. To avoid this problem, we use a threshold to convert a list of elements whose offsets are different (but with the same accessing code) to an element with an offset range.

Definition 5.2.2 (Kernel Execution Instance) *A kernel execution instance or a kernel run is an instance of the OS kernel execution.*

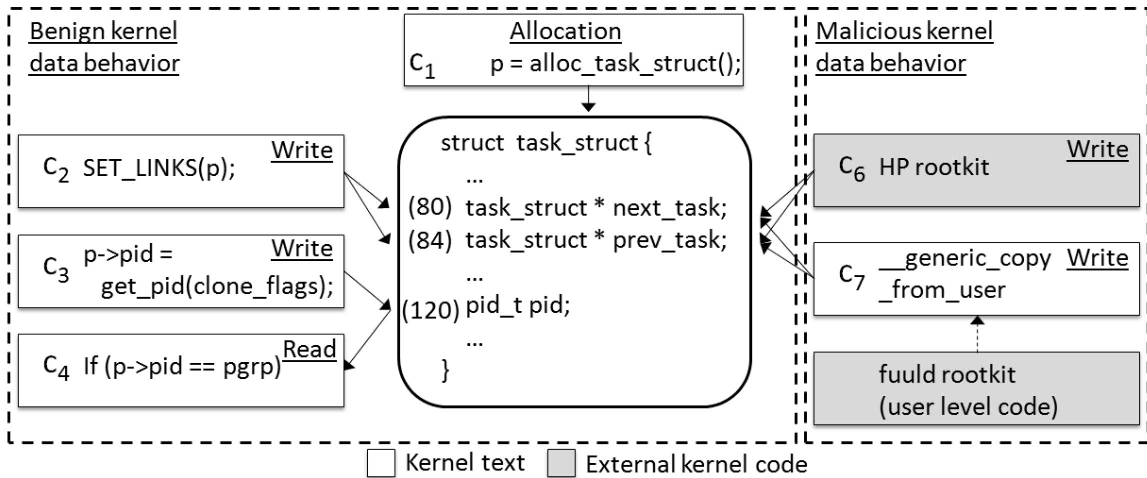


Figure 5.2.: An example of kernel code in benign and malicious kernel runs.

Definition 5.2.3 (Data Behavior Profile) *For a kernel execution instance r, a data behavior profile (DBP) is defined as a set of memory access patterns (DBEs) observed and it is denoted as D_r.*

A data behavior profile represents a set of data behavior elements observed in a kernel execution instance. It is a summary of all observed kernel-mode memory access patterns in the kernel run.

Figure 5.2 presents kernel code showing the examples of data behavior elements. The rounded box in the middle of Figure 5.2 shows a dynamic kernel object allocated by the code at the address c_1. This figure shows how this object is accessed by several code sites in kernel execution. Two fields, `next_task` (offset 80) and `prev_task` (offset 84), are written by the code at c_2. The code at c_3 reads the `pid` field (offset 120) and another code at c_4 reads this field. Therefore, the data behavior elements for this code example are as follows.

$$(c_2, 1, 0, c_1, 80)\,, (c_2, 1, 0, c_1, 84)\,, (c_3, 0, 0, c_1, 120)\; (c_4, 0, 0, c_1, 120)$$

These elements are the access patterns in a benign kernel run. If kernel malware is active in this kernel, the access patterns can be extended because of the malware

behavior. For instance, if kernel rootkits `hp` and `fuuld` are active as shown in the right-hand section of Figure 5.2, there would be additional accesses to the `next_task` and the `prev_task` fields by the code at c_6 and c_7. Consequently, the data behavior profile is extended with the additional elements as follows.

$$(c_6, 1, 0, c_1, 80), (c_6, 1, 0, c_1, 84), (c_7, 1, 0, c_1, 80), (c_7, 1, 0, c_1, 84)$$

Here c_6 represents the code of the `hp` rootkit, which is in the form of a kernel driver. The code integrity-based rootkit defense approach [10, 11] can determine this access as malicious based on the fact that this driver code is not in the authorized code list. In contrast, the code at c_7 is part of legitimate kernel code, which is indirectly exploited to overwrite this data structure. This rootkit case does not violate kernel code integrity; therefore, an approach based on code integrity cannot detect this attack behavior.

In both cases, malware behavior appears only when the malware runs. Our approach seeks to capture such behavior specific to the attack to determine the presence of malware.

5.2.2 Generating a Data Behavior Profile

In this section, we present the process for generating a data behavior profile, which summarizes the access patterns for all kernel objects accessed in a kernel run. Based on this information, we generate the signature of malware and inspect a kernel run for malicious data access patterns. A data behavior profile is generated based on two underlying functions. First, kernel objects should be identified with their unique classes. Second, the access patterns on numerous (e.g., tens of thousands in modern OSes) dynamic data instances should be summarized regarding their classes. We present two system components to provide these functions.

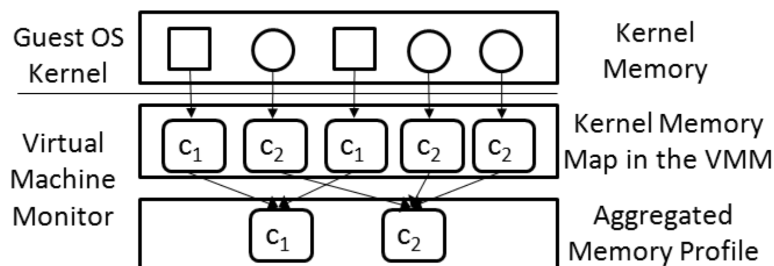


Figure 5.3.: Aggregating memory accesses on dynamic kernel objects regarding their classes (allocation sites) c_1 and c_2.

The Kernel Memory Mapper

DataGene uses the patterns of memory accesses on kernel objects and requires a kernel memory mapping mechanism to identify the targets of kernel memory accesses. LiveDM presented in Chapter 4 provides runtime kernel memory mapping that enables the identification of a memory access' target. LiveDM identifies kernel objects by transparently capturing the allocation and deallocation events of kernel memory. The generated map maintains the allocation code for each dynamic object as its runtime identifier. In offline static analysis, this identifier can be automatically translated into a data type by traversing kernel source code. We implemented the *kernel memory mapper* by employing LiveDM's approach.

The Data Behavior Aggregator

In a kernel execution instance, there exist a varying number of dynamic kernel data instances. To compare the access patterns of dynamic kernel objects in different kernel runs, it is necessary to aggregate the memory accesses on such objects regarding their classes. The allocation code represents the instantiation of a data type at a specific code position. By using a memory allocation code site as the classifier of dynamic kernel objects, we can aggregate the access patterns of dynamic instances of the same type and of a similar usage.

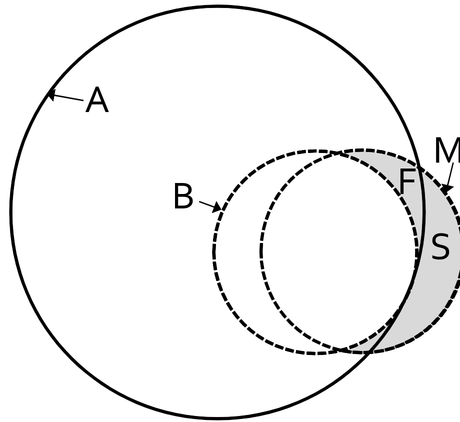


Figure 5.4.: A diagram of memory access patterns (DBEs). A: a set of frequently observed DBEs in benign kernel execution, B: a set of DBEs for benign kernel runs, M: a set of DBEs for malicious runs, S: a set of DBEs specific to malware attacks, F: a set of potential false positives DBEs.

Figure 5.3 illustrates this aggregation process. When a dynamic kernel object is allocated in a guest OS kernel, the kernel memory mapper stores its address range and the allocation code site as the class information in the kernel memory map. We have a memory mapping layer to aggregate the memory accesses on dynamic kernel objects regarding their data classes. Whenever kernel code reads or writes any dynamic kernel object, the VMM intercedes and identifies the targeted object by using its class information from the kernel memory map. If this memory access pattern is new, it is recorded in the aggregated memory profile.

5.2.3 Characterizing Malware Data Behavior

In this section we demonstrate how we characterize the behavior of kernel malware based on data behavior profiles.

DataGene characterizes malware behavior by using recurring memory access patterns specifically observed in malware attacks. Among those patterns, only the patterns that are rare in benign kernel execution should be chosen and it is a challenging task to differentiate such memory access patterns. This problem is illustrated in

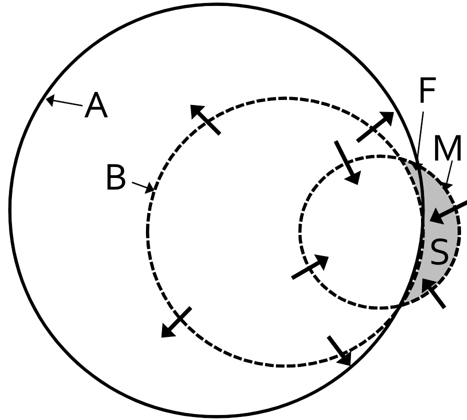


Figure 5.5.: Controlling kernel execution instances in the signature generation stage to reduce F. The descriptions for notations are shared with Figure 5.4.

Figure 5.4. The set A represents the DBEs that frequently occur in benign kernel execution. We seek to derive the set of DBEs specific to malware attacks, S. This set can be easily derived if A is available. However, obtaining the set A is challenging because it requires a full understanding of possible computation of the kernel. The halting problem, which is undecidable, shows the complexity of this problem. Instead, we derive S by using benign kernel runs (B) and malicious kernel runs (M). Specifically, a malware signature is generated by subtracting B from M. The resultant set includes both specific malware behavior (S) and potential false positive cases (F).

Hsin Pan and Eugene H. Spafford [91,92] proposed a new debugging approach that determines statements involved in program failures and reduces the search domain containing faults. This approach uses dynamic slicing and a set of heuristics to determine the minimal set of information to be examined for identifying program bugs. Our approach is similar to theirs in using multiple execution instances and extracting the common program execution patterns. By adapting the techniques and the findings of their approach, we expect future improvement of our approach.

As this methodology is based on dynamic kernel execution, it is difficult to eliminate the potential false positive set F. However, if we reduce F to the set of memory access patterns rarely triggered, we would be able to avoid frequent false positives in

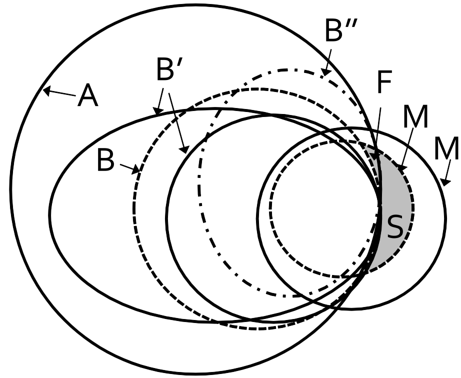


Figure 5.6.: Kernel execution instances in the detection stage. B': a set of DBEs for benign runs in the detection stage, B'': a set of DBEs for benign runs with false positives, M': a set of DBEs for malicious runs in the detection stage. Other notations are shared with Figure 5.4 and 5.5.

a typical workload. In the signature generation stage, we would be allowed to control the workload of kernel execution instances. Therefore, we can use several techniques to reduce the F set. Figure 5.5 shows the configuration we wish to achieve. By enlarging B and limiting M, we can control a majority of the memory access patterns in F to be covered by B pruning out frequent benign access patterns from the potential signature.

Figure 5.6 illustrates the relationships among the sets in the detection stage. M and B are the kernel runs used to generate the malware signature. M' is a tested malware run, and S will match it because it captures the recurring attack patterns of the malware. B' shows the tested benign kernel runs. We note that B' can be larger than B or it can partially overlap with the B set. However, as far as those sets do not overlap with F, they will not trigger false positives. This is the goal that we would like to achieve in our work. Note that if there is a kernel run, B'' that has the patterns of F, it will cause false positives.

Figure 5.7 illustrates a procedure for signature generation and matching. In the signature generation stage, we use the sets of benign kernel execution and malicious kernel execution as we control the workload to reduce F. After the signature is generated, we test false positives using another set of benign kernel execution. If

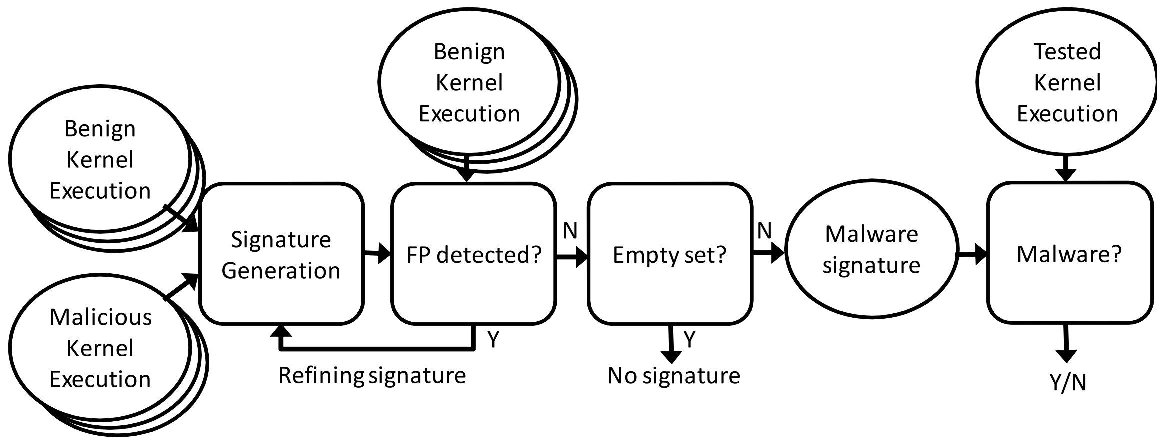


Figure 5.7.: A procedure for signature generation and matching

a false positive is detected, the triggering memory access pattern is not specific to malware patterns and therefore should be eliminated from the signature. This process can be done systematically by merging the tested benign runs to the set of benign runs for signature generation and regenerating the signature. If all memory access patterns are pruned out and the signature turns out to be empty, the malware in question does not have memory access patterns specific to the malware attack. This is out of the malware class that we wish to detect and is not covered by the DataGene system. Once the signature is generated, it is used to detect malware in the tested kernel execution.

Challenges and Our Solutions

DataGene characterizes malware behavior by using dynamic kernel execution. We list several challenges caused by our use of dynamic analysis. We also present our solutions for these challenges:

- **Variations in the Runtime Kernel Behavior.** Generally, the difficulty in obtaining a complete set of kernel execution paths is a well-known challenge for an approach based on dynamic execution. If we focus on the data behavior in benign execution, it is in fact a problem because the runtime kernel behavior

is highly dynamic across different runs. However, we focus on the data behavior specific to malware that consistently appears only when the malware is active. For instance, in Figure 5.6 even though benign runs (B') are highly dynamic, if they are not overlapped with F they would not trigger false positives.

- **Irregular Access Patterns on Kernel Stacks.** Kernel stacks are kernel objects that have irregular access patterns. Whenever a kernel function is called or returns, the stack is accessed for various purposes such as return values, function arguments, and local variables. As the kernel control flow is highly dynamic, the set of code sites that access the stack and the accessed offsets within the stack vary significantly. Also, the contents of kernel stacks are irregular at different runs. As such, a simple way to handle this problem is to exclude stacks from our analysis. The kernel memory mapper provides the identifier for kernel stacks and we solve this problem by removing the information for such dynamic objects from the analysis.

- **Varying Offsets in Arrays.** Some data structures (e.g., arrays and buffers) have a range of space, a part of which can be used at runtime. For example, the accessed offsets of a buffer can be different depending on the data contained in it. This problem is handled by using multiple instances of kernel execution. If the accessed offset of memory is different in each execution, it is not used for a malware signature because it may not be used in another run. Only the data behavior that occurs in a consistent pattern when malware is active becomes the candidate for the signature.

Characterizing Malicious Data Behavior

To reliably characterize the data behavior of kernel malware in dynamic execution, we use multiple kernel runs in the signature generation stage. $D_{M,j}$ is a data behavior profile for a malicious kernel run j with malware M. $D_{B,k}$ represents a data behavior profile for a benign kernel execution k. We apply the set operations on n malicious

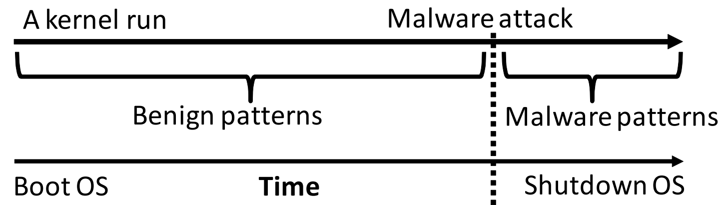


Figure 5.8.: Using a single kernel run for both of benign and malware memory access patterns

kernel runs and m benign runs as follows. The generated signature is called a *data behavior signature* for the malware M and shown as S_M.

$$S_M = \bigcap_{j \in [1,n]} D_{M,j} - \bigcup_{k \in [1,m]} D_{B,k} \tag{5.1}$$

This formula represents that S_M is the set of data behavior that *consistently* appears in n malware runs. However, this is also a set of behaviors *specific* to the attacks that *rarely* appears in m benign runs. The underlying observation from this formula is that kernel malware will consistently perform malicious operations during attacks so we extract malware behavior by taking the intersection of malicious runs. Such behavior should not occur in benign runs. Therefore, we subtract the union of benign runs from the derived malware behavior.

When we generate kernel execution runs with kernel malware, we use the cumulative memory access patterns before the attack as the benign kernel run and consider only the new patterns after the attack as the malware kernel run (shown in Figure 5.8). This technique prunes out significant benign access patterns from the malicious kernel run.

False positives may occur if a part of a signature is observed in a new tested benign run. The cause of this problem is not unknown kernel behavior, but rather a part of a signature not being properly pruned out in the signature generation. By exercising a variety of workloads in multiple kernel execution instances, we expect that such

potential behavior for this error can be significantly reduced from such constraints so that it does not cause frequent false positives in a typical workload.

Generalizing Malware Code Identity

DataGene aims at matching the variants of the rootkits whose signatures are available. For example, DataGene can be used to inspect suspicious data activity in the execution of new signed drivers (which may include hidden malicious code), the execution of an unknown driver (which may be malware or its variant), or kernel execution (where legitimate kernel code can be exploited indirectly for attacks).

To cover variants of malicious code, DataGene does not use specific identification of kernel drivers. When we generate signatures, we generalize the information specific to kernel drivers, thus allowing signatures to be tested against any driver from new signed drivers to new driver-based rootkits. Specifically, when the signature for a driver-based rootkit is generated, all code sites in this malicious driver are substituted by a single anonymous code site, ε. Some rootkits allocate memory and place their code on it, and any code site in such memory is also generalized as ε. In this process, we also generalize all benign kernel modules in the same way and subtract their memory access patterns from the candidates for the signature to collect only the behavior specific to the malware.

We preserve the code sites in the kernel text. The malware exploiting legitimate kernel code (e.g., the rootkits using memory devices or return-oriented rootkits) is handled by specific access patterns of legitimate code that are not observed in benign runs. In addition, when we match a malware signature with the data behavior profile of a kernel run, we generalize the driver code in the tested run similarly for comparison.

Matching a Malware Signature with a Kernel Run

The likelihood that a malware program M is present in a tested run r is determined by deriving a set of data behavior elements in S_M which belong to the data behavior

Algorithm 1 Derive the intersection of S_M and D_r.

```
1: function CheckSignature(S_M, D_r)
2:     I ← ∅
3:     for each e in S_M do
4:         for each e′ in D_r do
5:             if CompareElements(e, e′)= 1 then
6:                 I ← I ∪ {e}
7:             end if
8:         end for
9:     end for
10:    return I
11: end function
12: function CompareElements(e, e′)
13:    if e.c ≠ e′.c ∨ e.o ≠ e′.o ∨ e.m ≠ e′.m ∨ e.i ≠ e′i then
14:        return 0
15:    end if
16:    if e.f is an offset then
17:        if e′.f is an offset then
18:            if e.f = e′.f then
19:                return 1
20:            end if
21:        else                                  ▷ e′.f is a range of offsets.
22:            if e.f ∈ e′.f then
23:                return 1
24:            end if
25:        end if
26:    else                                      ▷ e.f is a range of offsets.
27:        if e′.f is a range of offsets then
28:            if e.f ⊂ e′.f then
29:                return 1
30:            end if
31:        end if
32:    end if
33:    return 0
34: end function
```

profile, D_r. This set I corresponds to the intersection of S_M and D_r[2] (i.e., $I = \{i | i \in S_M \wedge i \in D_r\}$); however, this set may not be symmetric for S_M and D_r because we allow two representations (i.e., an offset and a range of offsets) for the f field of a data behavior element. Algorithm 1 presents how this set I is generated.

Specifically, a data behavior signature S_M and a data behavior profile D_r consist of data behavior elements for all of the static and dynamic data structures. The CheckSignature function in Algorithm 1 compares each element of S_M and D_r, and returns the set of common elements, I. Two for-loops at lines 3 and 4 generate

[2]The data behavior signature (S_M) is a data behavior profile (i.e., a set of data behavior elements) because it is derived by the intersection and union of data behavior profiles.

a pair of elements each from S_M and D_r, and those elements are compared by calling the CompareElements function at line 5.

To consider the two compared elements e and e' as identical, their c, o, m, and i fields first should be equal. Next, their offset fields ($e.f$ and $e'.f$) are compared. Because the offset field can be either of an offset or a range of offsets, there are several cases shown in lines 16-33. If $e.f$ is an offset, it can match either an offset or a range of offsets. If both $e.f$ and $e'.f$ are an offset, their values should be identical. If $e.f$ is an offset and $e'.f$ is a range, they can match if $e.f$ belongs to $e'.f$'s range. If $e.f$ is a range of offsets, it can only match a range of offsets that includes $e.f$.

5.3 Implementation

DataGene generates the patterns of kernel memory accesses transparently without making changes in the source code of the OS. To implement this feature, we employ virtualization techniques. We used the QEMU [86] virtualizer with the KQEMU optimizer for our implementation. The host machine has 3.2Ghz Pentium D CPU and 2GB RAM. The guest machine is configured with 256MB RAM and the Redhat 8 operating system. This experimental platform is chosen for the convenience of implementation. However, our mechanism is generic and applicable to other operating systems and virtual machine platforms.

We implement the kernel memory mapper and the data aggregator in the VMM. The kernel memory mapper tracks kernel memory allocation and deallocation calls and captures dynamic kernel objects at runtime similar to [43]. When there is a request to the VMM, a data behavior profile can be dumped into a file anytime during the execution of the guest OS. For the purpose of generating a signature, dumping the profile once the OS is completely shutdown is preferred to capture most data behavior. However, to detect kernel malware, the data behavior profile can be periodically generated and compared with the signature while the OS is running.

In the benign runs, we performed various workload from daily commands to non-trivial application benchmarks. The tested workload includes kernel compilation, Apache webserver, UnixBench (Byte Magazine Unix Benchmark suite), nbench (BYTEmark Native Mode Benchmark), mysql database, thttp webserver, `find`, `gzip`, `ssh`, `scp`, `lsmod`, `ps`, `top`, and `ls` utilities. Some workloads were executed for several hours to allow any background administrative operation to be performed. We also used the workload of benign module loading and simple operations of the `/dev/kmem` device (e.g., open and close without overwriting kernel memory).

Among the memory accesses for kernel modules, we exclude the accesses to a kernel module by the same module which correspond to the accesses to a module's local variables. This information is not used to generalize the internal module activity. However, the accesses across modules are used after generalizing the accessing code information. In addition, the kernel data structure `module` having the administrative information regarding a kernel module is mapped to the head of each module's memory. We treat this part of memory as a separate data structure from the remaining module code or data.

5.4 Evaluation

In this section we evaluate the effectiveness of our data behavior signatures. First, we extract the signatures of three classic rootkits and match them with benign and malicious kernel runs. Second, we compare the signatures of all of the tested kernel rootkits to determine common data behavior across different rootkits and how such common behavior can be effective in detecting the variants of rootkits. Third, we list specific data elements that are shared by rootkit signatures, which provide an in-depth understanding of the attack operations that are common across kernel rootkits.

Table 5.1: Details of data behavior profiles for benign kernel runs. CL: # of classes, RS: # of read sites, WS: # of write sites.

Benchmark	Properties of a DBP					
	Dynamic Objects			Static Objects		
	CL	RS	WS	CL	RS	WS
boot&shutdown	200	9372	3732	15800	27287	3070
kernel compile	200	9260	3740	15800	30357	5895
apache	204	10205	4087	15800	27496	3121
find	200	9008	3614	15800	27087	2977
scp+gzip	201	10364	4205	15800	32471	6486
unixbench	201	9122	3679	15800	27222	3032
nbench	200	9028	3621	15800	27155	3009
mysql	201	9265	3736	15800	27142	3006
thttpd	206	10551	4212	15800	27442	3110
utils	201	10671	4186	15800	27840	3815
long 1	223	23934	5176	15800	31353	6837
long 2	207	11365	4503	15800	29749	4632
long 3	206	10976	4342	15800	29609	4605
long 4	204	10857	4301	15800	29556	4617
long 5	204	10978	4332	15800	29687	4617
Union	221	13918	5608	15800	39283	11449

Table 5.2: Details of malicious and benign kernel DBPs (D) and generated signatures (S). CL: # of classes, RS: # of read sites, WS: # of write sites.

	DBP (D) / Signature (S)	Dynamic Objects			Static Objects		
		CL	RS	WS	CL	RS	WS
adore 0.38	$D_{adore0.38,1}$	201	9148	3663	15800	27189	3005
	$D_{adore0.38,2}$	201	9114	3654	15800	27141	2998
	$D_{adore0.38,3}$	201	9143	3668	15800	27133	2989
	$D_{adore0.38,4}$	201	9149	3663	15800	27166	2996
	$D_{adore0.38,5}$	201	9127	3660	15800	27135	2992
	$\bigcap D_{adore0.38}$	193	8716	3296	15800	21333	2518
	$\bigcup D_{benign}$	221	13918	5608	15800	39283	11449
	$S_{adore0.38}$	2	1	2	1	1	1
SucKIT	$D_{SucKIT,1}$	201	9086	3645	15800	31786	3012
	$D_{SucKIT,2}$	201	9091	3653	15800	31757	2993
	$D_{SucKIT,3}$	201	9092	3655	15800	31781	3003
	$D_{SucKIT,4}$	201	9099	3665	15800	31754	2995
	$D_{SucKIT,5}$	201	9101	3651	15800	31761	2987
	$\bigcap D_{SucKIT}$	193	8720	3303	15800	22564	2515
	$\bigcup D_{benign}$	221	13918	5608	15800	39283	11449
	S_{SucKIT}	5	13	8	1192	1212	6
modhide	$D_{modhide,1}$	200	8987	3620	15800	27100	2983
	$D_{modhide,2}$	200	8999	3613	15800	27145	2997
	$D_{modhide,3}$	200	8985	3605	15800	27101	2985
	$D_{modhide,4}$	200	9013	3616	15800	27096	2988
	$D_{modhide,5}$	200	8985	3613	15800	27092	2984
	$\bigcap D_{modhide}$	192	8608	3276	15800	21306	2517
	$\bigcup D_{benign}$	221	13918	5608	15800	39283	11449
	$S_{modhide}$	1	0	1	0	0	0

Table 5.3: Details of the signatures for `adore` 0.38, `SucKIT`, and `modhide` rootkits. CL: # of classes, RS: # of read sites, RD: # of number of read data behavior elements, WS: # of write sites, WD: # of write data behavior elements.

Rootkit Name	Dynamic Objects					Static Objects					Total
	CL	RS	RD	WS	WD	CL	RS	RD	WS	WD	DBE
`adore`	2	1	5	2	14	1	1	8	1	7	35
`SucKIT`	5	13	29	8	12	1192	1212	11963	6	6	12010
`modhide`	1	0	0	1	1	0	0	0	0	0	1

5.4.1 Malware Signature Generation

When a data behavior signature is generated, the information specific to the malicious code is generalized in large. Therefore, we hypothesize that data behavior signatures may be effective not only to detect the malware whose signature is available, but also to determine the presence of related malware. To validate this hypothesis, we generated the signatures of three representative rootkits, and tested benign kernel runs and malicious kernel runs with 16 rootkits.

To generate malware signatures, we chose three rootkits: `adore 0.38`, `SucKIT`, and `modhide`. The `adore` rootkit has been studied in several rootkit defense approaches [10, 44, 60, 61]. This rootkit has several versions with differences in features and we chose an old version, 0.38, for the signature to evaluate its effectiveness toward newer rootkit versions (0.53 and 1.56). `SucKIT` is known for its attack vector, the `/dev/kmem` device, that avoids the conventional driver-based mechanism [22]. Several other rootkits followed this trend, using this device while having different goals. `modhide` is a rootkit packaged with the `adore` rootkits to hide them from the list of kernel modules.

To generate each malware signature, we used five malicious kernel runs with rootkits and 15 benign runs. Table 5.1 presents the details of data behavior elements of benign kernel execution instances. The first column shows the name of benchmark. The benchmarks named as *long x* run a mix of listed benchmarks for several hours. The next three columns show information about the dynamic objects, such as the number of classes for dynamic kernel objects, the number of code sites that read the dynamic kernel objects, and the number of code sites overwriting the dynamic kernel objects. The next three columns have similar information for the static kernel objects. As static objects (kernel functions and static data structures) are known at compile time, the number of classes for the static objects has the same value in different runs. These numbers represent a variety of data access behavior of the operating system kernel.

Table 5.2 shows the record of malicious kernel execution instances where kernel rootkits are running. In each rootkit category (e.g., `adore 0.38`) the information about malicious DBPs are listed first in the top five rows. Then the information regarding the intersection of such DBPs is shown. Next, the union of benign DBPs is presented. In the following row, details of the derived rootkit signature are shown.

Table 5.3 presents the details of three rootkit signatures. Three data behavior signatures of the `adore`, `SucKIT`, and `modhide` rootkits have 35, 12010, and 1 data behavior elements (DBEs), respectively. `SucKIT` has a significantly higher number of elements because it scans kernel memory to collect information about the attack targets (e.g., the system-call table), and this behavior is observed as reading numerous static objects with a variety of offsets. The `modhide` rootkit simply manipulates the kernel module list; thus, it has a few elements.

5.4.2 False Positive Analysis

To evaluate the false positives of the generated signatures, we compare the signatures with new benign kernel execution instances. Table 5.4 shows the result of this experiment. This table has the same set of benchmarks and the same format as Table 5.1. In these kernel runs, we generated an additional variety in the workload (e.g., an additional run) so that such kernel runs contain more code paths and data operations beyond the kernel runs used for generating signatures. This additional runtime variation results in more code sites for memory accesses (i.e., higher numbers in # of read code sites and # of write code sites).

In this experiment, no false positive cases were found, which confirms that our signature generation procedure captures a reasonably close set of the data behavior specific to the kernel rootkits and that the tested runs did not contain any data behavior that appears in the signatures.

Table 5.4: Benign kernel runs tested for false positives. A: $S_{adore0.38}$, S: S_{SucKIT}, M: $S_{modhide}$. CL: # of classes, RS: # of read sites, WS: # of write sites.

Benchmark	Properties of a DBP						Signature Match		
	Dynamic Objects			Static Objects					
	CL	RS	WS	CL	RS	WS	A	S	M
boot&shutdown	200	9391	3743	15800	27310	3079	0	0	0
kernel compile	201	9766	3903	15800	31813	7646	0	0	0
apache	204	10249	4108	15800	27507	3126	0	0	0
find	200	9448	3756	15800	27318	3076	0	0	0
scp+gzip	201	10774	4233	15800	36909	8132	0	0	0
unixbench	201	9520	3799	15800	27353	3087	0	0	0
nbench	200	9460	3758	15800	27333	3084	0	0	0
mysql	201	9731	3890	15800	27378	3095	0	0	0
thttpd	206	10942	4356	15800	27720	3237	0	0	0
utils	202	10723	4225	15800	27866	3203	0	0	0
long 1	223	12610	4980	15800	30151	4707	0	0	0
long 2	223	12636	4911	15800	30172	4714	0	0	0
long 3	223	12635	4925	15800	30156	4710	0	0	0
long 4	223	13087	5285	15800	31053	6968	0	0	0
long 5	223	13118	5281	15800	33776	8025	0	0	0

Table 5.5: The number of matched data behavior elements between three rootkit signatures and the kernel runs with 16 kernel rootkits (average of 5 runs). (`AD1`: `adore` 0.38, `AD2`: `adore` 0.53, `AD3`: `adore-ng` 1.56, `FL`: `fuuld`, `HL`: `hide_lkm`, `SK`: `SucKIT`, `ST`: `superkit`, `LF`: `linuxfu`, `CL`: `cleaner`, `MH`: `modhide`, `MH1`: `modhide1`)

Signature (S_M)		# of matched DBEs between S_M and the kernel runs with the rootkits shown below ($\|I\|$).															
M	$\|S_M\|$	`AD1`	`AD2`	`AD3`	`FL`	`HL`	`SK`	`ST`	`hp`	`kbdv3`	`knark`	`LF`	`Rial`	`CL`	`kis`	`MH`	`MH1`
`AD1`	35	*35*	30	14	0	0	2	2	2	5	20	3	4	0	2	0	0
`SK`	12010	2	1	1	16	16	*12010*	11983	0	0	1	0	0	0	16	0	0
`MH`	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	*1*	1
Detected		√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
# of effective S_M		2	2	2	1	1	2	2	1	1	2	1	1	1	2	1	1

5.4.3 Detecting Rootkits using Data Behavior Signatures

Malicious kernel runs were next tested by using three signatures to determine any running malware based on the similarity of the data access patterns between the compared signature and the kernel run. We tested a total of 80 kernel runs of 16 rootkits having a variety of targets and attack vectors. For instance, seven rootkits (`fuuld`, `hide_lkm`, `hp`, `linuxfu`, `cleaner`, `modhide`, and `modhide1`) directly manipulate kernel objects (DKOM [75]). Four rootkits (`fuuld`, `hide_lkm`, `SucKIT`, and `superkit`) manipulate kernel memory by using the `/dev/kmem` memory device, among which two rootkits (`fuuld` and `hide_lkm`) directly manipulate only kernel data and do not violate kernel code integrity. Therefore, they are not detected by code integrity-based defense systems [10, 11].

Table 5.5 presents the number of matched data behavior elements between signatures and kernel runs with rootkits (I). Two left-hand columns show the information about signatures: the name (M) of the rootkit used for the signature and the size of the signature ($|S_M|$). The remaining 16 columns present the number of data behavior elements common in the compared signature (based on the rootkit in the row heading) and the kernel run (where the rootkit in the column heading is active). The presented numbers are the averages of five kernel runs. However, the numbers are consistent in the runs with the same rootkit.

If the rootkit used for the signature and the rootkit in the tested run are identical, the entire signature was matched giving $|I| = |S_M|$. For example, the signatures of `adore 0.38`, `SucKIT`, and `modhide` rootkits fully match the kernel runs with those rootkits (shown in *italics*). We consider that a tested run includes a potential malware running if one or more signatures have a matched element with the kernel run. In our experiments, all kernel runs with rootkits shared elements with one or more signatures (shown in the row at the bottom of the table), leading to the detection of 16 kernel rootkits.

5.4.4 Similarities among Data Behavior Signatures

In the previous section we demonstrated that a variety of rootkits can be detected by using the signatures of a few classic rootkits because they have common data access patterns. In this section we quantitatively measure the similarities in data behavior across rootkits by generating and comparing the signatures of the tested rootkits.

We first generated the signatures of 16 kernel rootkits by applying the set operations (Section 5.2.3) on five malicious kernel runs with rootkits and 15 benign kernel runs. Then we calculated the similarities among signatures by applying Algorithm 1 on the combinations of 16 rootkit signatures. Table 5.6 lists the number of common data behavior elements in such combinations. For a pair of rootkits M_1 in the row heading and M_2 in the column heading, the cross section of the corresponding row and column shows the number of data behavior elements common in two signatures of M_1 and M_2. This number may not be symmetric for M_1 and M_2 because a data behavior element can have two representations for its f field (an offset or a range of offsets). If M_1 and M_2 are the same rootkit, the number of elements is shown in *italics*.

For the rootkit M_2 in the column heading, if positive numbers are listed in the column, the signatures of the rootkits (in the row headings) can be used to determine M_2. The number of such signatures (except S_{M_2} itself) is presented at the second bottom row (# of effective S_M). The maximum size of such signatures is shown in the bottom row (Max |effective S_M|). In our experiments, a rootkit shares its data behavior with 2∼10 of other rootkits (more than six rootkits in average). The rootkits show similar data behavior not only among close variants (e.g., different versions of `adore` rootkits) but also across the rootkits having different attack mechanisms (e.g., `SucKIT` shows similarities with driver-based rootkits such as `knark` or `kis`).

The similarities of data behavior across rootkits are visualized in Figure 5.9. A node represents a rootkit signature and an arrow shows the similarity between two signatures using three different arrow types. An arrow from a node M_1 to a node M_2

Table 5.6: The number of common data behavior elements in the combination of rootkit signatures. (`AD1`: `adore` 0.38, `AD2`: `adore` 0.53, `AD3`: `adore-ng` 1.56, `FL`: `fuuld`, `HL`: `hide_lkm`, `SK`: `SucKIT`, `ST`: `superkit`, `LF`: `linuxfu`, `CL`: `cleaner`, `MH`: `modhide`, `MH1`: `modhide1`)

M	$\lvert S_M \rvert$	AD1	AD2	AD3	FL	HL	SK	ST	hp	kbdv3	knark	LF	Rial	CL	kis	MH	MH1
AD1	35	*35*	30	14	0	0	2	2	2	5	20	3	4	0	2	0	0
AD2	46	30	*46*	24	0	0	1	1	2	5	19	2	4	0	2	0	0
AD3	97	14	24	*97*	0	0	1	1	2	4	9	6	0	2	2	0	0
FL	19	0	0	0	*19*	13	16	16	0	0	0	0	0	0	0	0	0
HL	3406	0	0	0	13	*3406*	13	13	0	0	0	0	0	0	0	0	0
SK	12010	2	1	1	16	13	*12010*	11983	0	0	1	0	0	0	16	0	0
ST	11998	2	1	1	16	13	11983	*11998*	0	0	1	0	0	0	1	0	0
hp	17	2	2	2	0	0	0	0	*17*	0	1	5	0	0	1	0	0
kbdv3	16	5	5	4	0	0	0	0	0	*16*	4	0	0	0	0	0	0
knark	67	20	19	9	0	0	1	1	1	4	*67*	1	4	0	2	0	0
LF	24	3	2	6	0	0	0	0	5	0	1	*24*	0	0	1	0	0
Rial	46	4	4	0	0	0	0	0	0	0	4	0	*46*	0	0	0	2
CL	3	0	0	2	0	0	0	0	0	0	0	0	0	*3*	0	1	1
kis	31203	2	2	2	0	0	16	1	1	0	2	1	0	0	*31203*	0	2
MH	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	*1*	1
MH1	6	0	0	0	0	0	0	0	0	0	0	0	2	1	2	1	*6*
# of effective S_M		10	10	10	3	3	8	8	6	4	10	6	4	3	9	2	4
Max $\lvert$effective $S_M\rvert$		30	30	24	16	13	11983	11983	5	5	20	6	4	2	16	1	2

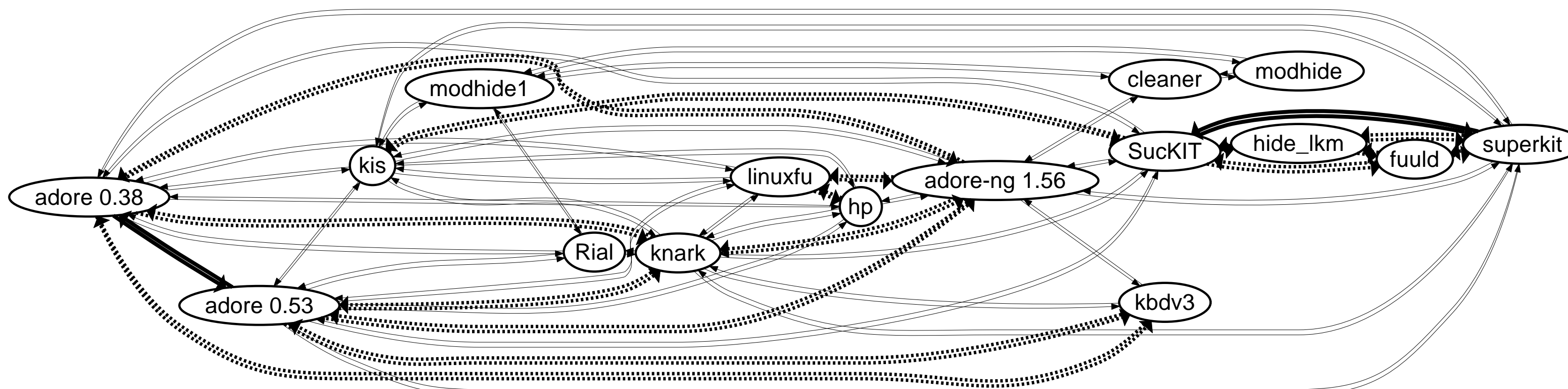


Figure 5.9.: Similarities among the data behavior of rootkits. Types of arrows ($|I|$: # of the matched elements): thin solid ($0 < |I| < 5$), thick dashed ($5 <= |I| < 25$), and thick solid ($|I| >= 25$).

means that the signature M_1 can be used to determine the rootkit of the signature M_2. This figure illustrates that several groups of rootkits have strong similarities. The family of `adore` rootkits (i.e., `adore 0.38`, `adore 0.53`, and `adore-ng 1.56`) are strongly related in general. The `adore-ng 1.56` is connected to other versions with less strong connections, thick dashed arrows, because in newer `adore` versions (bigger than 1.0 whose name is changed to `adore-ng`), the internal attack vector is substantially changed to use dynamic objects instead of static objects. A group of rootkits using the `/dev/kmem` memory device (i.e., `SucKIT`, `hide_lkm`, `fuuld`, and `superkit`) have a strong relationship to one another. The `SucKIT` and the `superkit` are especially connected by using thick solid arrows because they share a majority of data behavior. Some rootkits have relationships with different kinds of rootkits. For example, the `kis` rootkit is connected to driver-based rootkits such as the `adore` rootkits and the `knark` rootkit; but, it is also closely related to `/dev/kmem` based rootkits such as the `SucKIT`.

As seen in Figure 5.9, the data behavior is not only common in the family of rootkits or similar kinds, but also is available across different kinds of rootkits. The signatures of these related rootkits can be interchangeably used to detect one another.

5.4.5 Extracting Common Data Behavior Elements

In this section we demonstrate the details of common rootkit attacks which are systematically extracted based on the similarities in rootkits' data behaviors. The data behavior elements (DBEs) from the signatures of all experimented rootkits are ranked with the order of the appearance in rootkits' signatures (N). The top DBEs are presented in Table 5.7 after being classified into several categories.

The first three columns present the information regarding rootkits which share data behavior elements. The number N and the names of rootkits whose signatures share a DBE are listed. A short description of the element is provided in the next column by considering the information of the DBE.

Table 5.7: Top common data behavior elements among the signatures of 16 rootkits. (AD1: adore 0.38, AD2: adore 0.53, AD3: adore-ng 1.56, FL: fuuld, HL: hide_lkm, SK: SucKIT, ST: superkit, LF: linuxfu, CL: cleaner, MH: modhide, MH1: modhide1)

Rootkits			Accessing code		Accessed data		
N	Rootkits with common behavior	Rootkit behavior	Code (c)	o	m	Data class (i)	Field,Offset (f)
7	AD1, AD2, AD3, hp, knark, LF, kis	Reading a process's ID	ε	R	D	task_struct	pid
6	AD1, AD2, AD3, SK, ST, knark	Reading a process's flag	ε	R	D	task_struct	flags
5	AD1, AD2, AD3, kbdv3, knark	Privilege escalation	ε	W	D	task_struct	uid, euid, gid, egid
5	AD1, AD2, AD3, hp, LF	Listing processes	ε	R	D	task_struct	next_task
4	AD1, SK, ST, kis	Setting an address space	ε	W	D	task_struct	addr_limit
4	AD1, AD2, AD3, knark	Privilege escalation	ε	W	D	task_struct	suid, fsuid, fsgid
3	AD1, AD2, AD3	Privilege escalation	ε	W	D	task_struct	cap_effective
3	AD1, AD2, AD3	Privilege escalation	ε	W	D	task_struct	cap_inheritable
3	AD1, AD2, AD3	Privilege escalation	ε	W	D	task_struct	cap_permitted
3	AD1, AD2, kbdv3	Reading a user's ID	ε	R	D	task_struct	uid
3	AD1, AD3, LF	Reading a process' name	ε	R	D	task_struct	comm
2	hp, LF	Hiding a process	ε	W	D	task_struct	next_task, prev_task
4	FL, HL, SK, ST	Manipulation via /dev/kmem	read_kmem, write_kmem	R,W	D	file	f_pos
4	FL, HL, SK, ST	Manipulation via /dev/kmem	memory_lseek	W	D	file	f_pos
3	FL, SK, ST	Manipulation via /dev/kmem	do_write_mem	R,W	D	file	f_pos
3	CL, MH, MH1	Hiding a kernel module	ε	W	D	module	next
2	kis, MH1	Hiding a kernel module	ε	W	S	module_list	0
4	AD1, AD2, knark, Rial	Hijacking a system call	ε	W	S	sys_call_table	# 141
3	AD1, AD2, knark	Hijacking a system call	ε	W	S	sys_call_table	# 2,37,120,220
3	AD1, AD2, Rial	Hijacking a system call	ε	W	S	sys_call_table	# 6
2	Rial, MH1	Hijacking a system call	ε	W	S	sys_call_table	# 5
2	knark, Rial	Hijacking a system call	ε	W	S	sys_call_table	# 3
2	SK, ST	Hijacking a system call	ε	W	S	sys_call_table	# 59
2	SK, ST	Hijacking a system call	__generic_copy_from_user	W	S	sys_call_table	# 59
2	AD1, AD2	Hijacking a system call	ε	W	S	sys_call_table	# 39
2	AD2, AD3	Hijacking a hook	ε	W	S	proc_root_inode_operations	lookup

The next five columns present the contents of data behavior elements: the accessing code (c); the kind of memory access (o) such as a read (R: $o = 0$) or a write (W: $o = 1$); the kind of accessed memory (m) such as a dynamic object (D: $m = 0$) or a static object (S: $m = 1$); the accessed memory's class (i), which is converted to a data type for dynamic data or a variable name for static data; and the accessed offset(s) (f). The offset is converted to a field name if it corresponds to a specific field. If the accessed object is the system-call table, a system-call number (#) is presented by dividing the offset by the size of a pointer.

- **Attacks on Process Control Blocks (PCBs).** The first category at the top of Table 5.7 lists the data behavior that targets the PCBs (type: `task_struct` in Linux). This is a core data structure that maintains administrative information about processes. Therefore it is a major target of rootkits, that manipulate such information.

  Table 5.7 shows that seven rootkits read the process ID numbers in PCBs during attacks. The flags of the processes are accessed by six rootkits. Several rootkits, such as the family of `adore` rootkits, the `kbdv3` rootkit, and the `knark` rootkit, provide a back-door that gives root privileges for an ordinary user. The `hp` and `linuxfu` rootkits show an attack pattern that manipulates the pointers connecting PCBs. This behavior can hide PCBs from the view inside the operating system.

- **Attacks using /dev/kmem.** The second category shows the rootkit behavior that manipulates kernel memory by using a memory device (e.g., `/dev/kmem`). This device allows a user program to read and write kernel memory, putting the kernel integrity at risk. The kernel runs compromised by `fuuld`, `hide_lkm`, `SucKIT`, and `superkit` rootkits commonly show specific data behavior that the kernel functions related to memory devices access `file` kernel objects.

Table 5.8: Configuration of benchmarks

Name	Version	Command
Kernel compile	2.4.18	`time make`
UnixBench	5.1.2	`./Run -i 1`
nbench	Version 2	`time nbench`
bzip2	1.0.2	`time tar cvfj linux.tbz2 linux-2.4.18`
find	4.1.7	`time find /etc | xargs grep noexist`

- **Attacks on the Kernel Module List.** The next category lists rootkit attacks on the kernel module list. The `next` pointer field of `module` objects are written by the `cleaner`, `modhide`, and `modhide1` rootkits. The `module` objects constitute the list of kernel modules and they are connected by this pointer field. The rootkit attacks that hide a module appear as direct manipulation of this field.

- **Attacks on Static Kernel Objects.** The last category is the manipulation of static kernel objects. Several rootkits hijack system-calls by replacing the system-call table entries with the addresses of malicious functions. This behavior is captured by the manipulation of the system-call table by several code sites, depending on the attack vector. In the case of driver-based rootkits, such behavior is captured as access by the generalized rootkit code, ε. The rootkits based on memory devices (e.g., `/dev/kmem`) use legitimate kernel code for manipulation (e.g., `__generic_copy_from_user`).

5.4.6 Monitoring Performance

We evaluated the performance of DataGene compared to the unmodified QEMU and the LiveDM system. We performed five benchmarks, and their configurations are presented in Table 5.8. In kernel compile, nbench, bzip2, and find benchmarks, we used the total runtime measured for the workload. UnixBench has several internal benchmarks in the benchmarking process. Therefore, the total benchmarking time

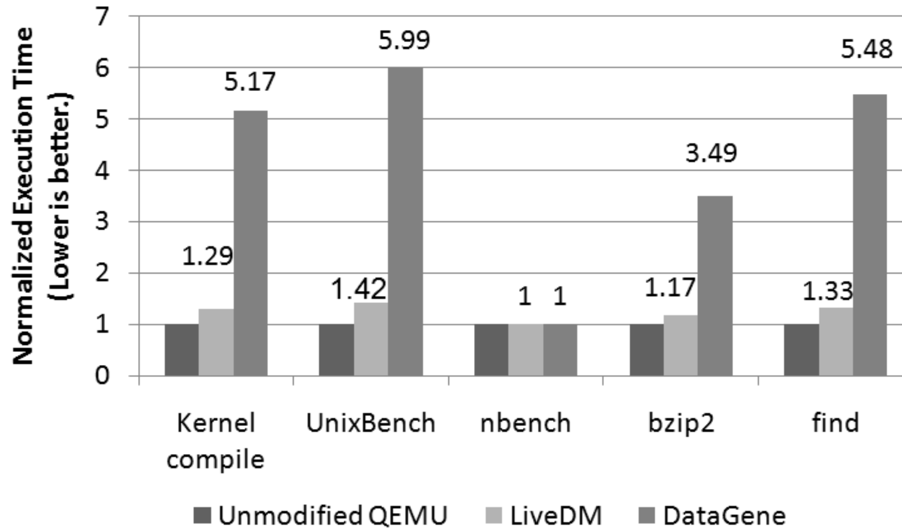


Figure 5.10.: Performance comparison of unmodified QEMU, LiveDM, and DataGene (OS: Redhat 8)

does not represent performance properly. We used the performance index from the report of the benchmark instead of its total execution time.

Figure 5.10 presents the performance overhead of unmodified QEMU, LiveDM, and DataGene. All performance numbers are normalized to the results of unmodified QEMU, and a lower number represents faster execution.

DataGene has two stages of operations: signature generation and malware detection. In the current implementation, DataGene intercedes on every kernel mode memory access. Therefore, DataGene has higher performance overhead than LiveDM, which intercedes only when the kernel executes kernel memory allocation and deallocation code. For the signature generation stage, this overhead is necessary to observe the entire malware behavior and to generate its signature. However, in the detection stage, it is necessary to monitor the memory accesses only to the kernel objects listed in a given malware signature. Malware typically has a limited number of malicious actions. Therefore the monitoring performance can be further optimized by reducing the monitoring scope. The presented result corresponds to a conservative performance

analysis of complicated malware behavior that may access any kernel objects because DataGene inspects all kernel mode memory accesses in the current experiments.

Kernel compile, UnixBench, and find benchmarks intensively use system resources such as file systems, pipes, and processes. Such activities invoke kernel services such as system calls and page fault handling, which indirectly trigger kernel-level memory activities. The nbench benchmark involves only user-level CPU workload and therefore does not cause kernel level memory accesses nor trigger kernel level services. Both LiveDM and DataGene do not have additional overhead for this case. bzip2 benchmark involves both file system access and user-level computation. Therefore it caused lower overhead compared to kernel compile, UnixBench, and find benchmarks.

5.5 Summary

In this chapter, we presented a new approach to characterize the behavior of kernel malware by using kernel data access patterns specific to the malware. We also demonstrated the effectiveness of our implementation, DataGene, in the evaluation of kernel rootkit detection.

Kernel malware signatures are constructed by using benign kernel runs and malicious kernel runs. Our experiments show that the signatures of three classic rootkits can effectively detect the kernel runs compromised by 16 kernel rootkits without triggering false positives in typical benign workloads.

We further analyzed the similarities of the signatures for 16 rootkits. Each signature shares similar access patterns with 2$\sim$10 other rootkit signatures, which are effectively exposed by DataGene, enabling the use of memory access patterns. In addition, we presented the details of common data behavior, which provide an in-depth understanding of the attack behavior of kernel rootkits.

6 CONCLUSIONS

Many malware detection mechanisms rely on the properties of malware code such as the injection of unauthorized code [6–11] and the patterns of malicious code sequences [26–30]. While these approaches are effective for classic malware, emerging malicious programs are introducing advanced techniques such as return/jump-oriented programming [14–21], code obfuscation [31–34], and code emulation [35] to elude those malware detection mechanisms. In this dissertation, we have presented a new approach for detecting kernel malware based on the properties of kernel data objects.

In Chapter 3, we first discussed the code-centric approaches based on the properties of malicious code. We then introduced data-centric malware defense architecture that models and detects kernel malware using the properties of data objects. This architecture is composed of two components: a runtime kernel object mapping system that enables the monitor to use the properties of data objects and a kernel malware detection approach based on the kernel memory access patterns.

In Chapter 4, we presented a runtime kernel object mapping approach which uses virtualization technology. This approach identifies kernel objects by capturing the execution of the kernel memory allocation and deallocation functions. It generates a view of kernel objects that is un-tampered by manipulating pointer connections, unlike related approaches based on memory snapshots. We demonstrated its effectiveness via detection of 10 kernel rootkits that hide kernel data objects. In addition, we presented the effectiveness of its temporal view by analyzing malware attacks that target dynamic kernel objects.

In Chapter 5, we demonstrated the detection a class of malware that has recurring kernel memory access patterns specific to malware attacks. We implemented a prototype system, DataGene, using the QEMU virtual machine monitor and demonstrated

its effectiveness against 16 kernel rootkits. We used dynamic kernel execution analysis to generate malware signatures which do not trigger false positives for typical server workload such as web-servers, databases, kernel compiling, and utilities. Our experiments demonstrated that this approach effectively exposes the similarity of attack behavior among rootkits. Using the signatures of three kernel rootkits, we could detect not only the rootkits with signatures but also the other 13 kernel rootkits that share kernel memory access patterns in their attacks. The cross-comparisons among the 16 rootkit signatures showed that each rootkit shared memory access patterns with 2∼10 other rootkits.

6.1 Discussion and Limitations

As LiveDM operates in the VMM beneath the hardware interface, we assume that kernel malware cannot directly access LiveDM code or data. However, it can exhibit potentially obfuscating behavior to confuse the view seen by LiveDM. Here we describe several scenarios in which malware can affect LiveDM and our counter-strategies to detect them.

First, malware can implement its own custom memory allocators to bypass LiveDM observation. This attack behavior can be detected based on the observation that any memory allocator must use internal kernel data structures to manage memory regions or its memory may be accidentally re-allocated by a legitimate memory allocator. Therefore, we can detect unverified memory allocations by comparing the resource usage described in the kernel data structures with the amount of memory being tracked by LiveDM. Any deviation may indicate the presence of a custom memory allocator.

In a different attack strategy, malware could manipulate valid kernel control flow and jump into the body of a memory allocator without entering the function from the beginning. This behavior can be detected by extending LiveDM to verify that the function was entered properly. For example, the VMM can set a flag when a

memory allocation function is entered and verify the flag before the function returns by interceding before the return instruction(s) of the function. If the flag was not set prior to the check, the VMM detects a suspicious memory allocation.

DataGene is a signature-based approach that detects known and unknown rootkits based on kernel data access patterns similar to the signatures of previously analyzed rootkits. If a rootkit's attack behavior is not similar to any behavior in existing signatures or it does not involve kernel data accesses, such malware is out of coverage of DataGene because such behavior does not match the DataGene's signature.

Many existing rootkits that share the attack goals often exhibit similar data access patterns because essentially these malicious programs generate a false view by manipulating legitimate kernel data structures relevant to the goals. Our approach can detect rootkits by focusing on the common attack targets described in the malware signatures even though such rootkits have different functionalities.

Obfuscating data access patterns involves comparatively more sophistication than code obfuscation because malware requires to use alternate legal code to access kernel data beyond the diversification of malware's own code patterns. These attack attempts can be detected by employing the defense approaches against control flow anomaly.

In the environment whose typical workload can be determined, it is possible to produce malware signatures that can avoid frequent false alarms as presented in our experiments. However, if this technique is further directed towards a production environment where a diversity of workload could be generated, false alarms may occur because of the foundation of our technique on dynamic execution.

This dissertation focuses on malware detection and analysis targeting operating system kernels. However, some of methodologies can be applied to user level applications with changes in the implementation details.

In the case of a user program, dynamic memory is managed by external system components. For example, `malloc` and `free` functions are part of C library, which internally uses system calls to map and unmap memory pages into the memory space

of the program. Therefore, the information about user level data objects can be obtained by intercepting these memory management interfaces. Memory accesses to dynamic objects can be intercepted by using several techniques. The page tables for the process can be used to intercept memory accesses by setting page permissions as read-only. Similar to `gdb`, debugging registers also can be used.

User level programs have underlying system software layers that include the C library, system calls, and the kernel code. Such layers have higher privileges than the monitored user program; therefore, they are suitable for implementing a monitor with protection from the monitored program. Compared to kernel level data monitoring, user level monitoring offers more convenience in the implementation because of these underlying layers.

Our approach provides advanced detection and analysis of kernel malware activities based on the information regarding kernel data objects. It is primarily designed for malware analysis environments such as honeypots and malware profilers. In the current implementation, the advanced features based on data properties incur nontrivial performance overhead from fine-grained monitoring of kernel level memory accesses. There are several approaches to address this concern and improve the current implementation.

AfterSight [88] is a research prototype from VMware that decouples dynamic program analysis from a production run by using virtual machine record and replay technology. A light-weight log is generated from a production virtual machine. Then security checks are applied to another virtual machine that replays the recorded log in the backend. As expensive security inspection is performed on the replay machine, it does not affect the production run. If this technique is leveraged, our approach can be applied to environments that require production level performance.

In addition, hardware virtualization can be used to build a more efficient implementation of our approach. By setting the page permission for the inspected kernel memory as read-only, the VMM can intercept the memory accesses to the monitored

kernel data objects. By reducing the scope of interceptions to the data objects listed in the signature, we can further optimize the monitoring performance.

DataGene uses kernel mode memory access patterns to detect kernel malware. Therefore, its monitoring efficiency varies, depending on the kernel mode workload. As presented in Section 5.4.6, if the workload is mostly CPU bound, it could incur trivial overhead. However, if an application intensively uses kernel services such as system calls, such events can indirectly trigger kernel mode memory accesses and incur non-trivial overhead. Malware may attempt to exploit this characteristic to lower application performance and cause denial of service attacks.

In this dissertation we highlighted the handling of dynamic kernel objects because monitoring dynamic memory has more challenges than monitoring static objects. For instance, the addresses of dynamic objects are determined at runtime and the number of runtime instances varies.

Compared to dynamic kernel objects, static objects have memory addresses that are predetermined at the compilation time. The manipulation of static objects is observed as write accesses to their unique addresses. If such memory access patterns are observed specifically during malware execution, they are extracted as malware signatures. For example, system call hijacking is implemented as the manipulation of the system call table that is a static object. The manipulation of this object by other than the legitimate initialization code is rare in benign execution. Thus, this attack pattern is automatically extracted as a signature.

If the manipulated memory is executed and used in a different way from the overwritten memory, DataGene can extract it as malicious behavior. If the overwritten memory corresponds to a data object before an attack, its execution is specific to the attack because the memory is not executed in the benign run. Typically, the overwritten code by malware exhibits memory access behavior different from the original code. Otherwise the attacker could have reused the original code.

If the injected code accesses the data objects in the same way with the overwritten code, this access pattern is not specific to this malware; and this attack behavior

therefore does not belong to our malware behavior criteria and can evade our approach. While this is a possible attack scenario, it can be detected if our approach is deployed with code integrity checking or control flow integrity checking. These approaches detect any manipulation of code which is not meant to be modified in typical cases (except self-modifying code or dynamic recompilation). The combination of these code-centric approaches and our data-centric approaches places significant constraints on the capability of attackers. The attacker can be effectively limited not only in what can be executed (i.e., the integrity of code), but also in what can be accessed (i.e., the integrity of the memory access targets by the code).

Another attack mechanism towards our approach is to avoid a recurring pattern. Our approach assumes that the malware mechanism in the signature occurs when the malware is active. However, it is not necessarily true for all malware. Malware can have adaptive adversary behavior. For example, malware can have logic that activates or deactivates malicious behavior at certain conditions (e.g., holidays or when there is no user logged in). In general, it is a challenging problem to understand the hidden malicious logic that can be a combination of a variety of system variables and formulas. One potential strategy to detect this type of malware is to expose the hidden behavior by setting various configurations of system variables in the signature generation stage. The search space of such combinations would be a significant challenge.

Our approach is based on kernel memory access patterns. As an extreme case against our approach, malware can be constructed by only using arithmetic instructions and the accesses to registers. This malware can achieve some computations. However, this attack strategy will be significantly limited in making changes in kernel behavior as most existing kernel malware does.

6.2 Conclusions

In this dissertation, we presented an approach to detect a class of malware using recurring memory access patterns that are specific to malware attacks.

The data-centric malware defense architecture (DMDA) is effective at detecting this class of malware without causing a high number of false positive cases. This is because many kernel rootkit attacks exhibit kernel data access patterns specific to their attacks to change legitimate kernel behavior and such memory access patterns are rare in benign kernel execution. In experiments with 16 kernel rootkits, we could generate a non-empty set of recurring memory access patterns specific to rootkit attacks for 16 rootkits. These patterns successfully match the presence of rootkit execution.

In our experiments, the signatures managed to avoid triggering false positives in 15 typical workloads, such as production applications and utility programs. We contend that our signature derivation process can produce reasonably effective malware signatures after successfully pruning out frequent benign memory access patterns from signature candidates. However, it does not guarantee that these signatures do not trigger false positive cases in other workloads because it is a challenging task to inspect all possible benign memory access patterns in an OS kernel. The generated malware signatures usually have a limited size because typical kernel malware interacts with the OS in a limited number of ways. Therefore, if a false positive error is triggered, it is feasible that a human expert who understands OS kernel code would manually inspect the case and confirm the malware behavior.

This approach does not use code-centric properties such as the injection of unauthorized code or malicious control flow patterns to detect malware. In the experiments, our prototype could successfully detect kernel rootkits that do not inject code into kernel memory for attacks. Therefore, this data-centric approach can complement code-centric approaches by not depending on solely code information.

The data access patterns in our approach are of a general form that can match other malware if it targets similar kernel data objects. Our experiments have demonstrated that the generated signatures are effective in matching not only the rootkits of the signatures, but also malware variants that share data access patterns. This char-

acteristic demonstrates the potential of this approach to detect new malware based on the similarity of the data access behavior.

6.3 Future Work

In this section we present future work to improve our current results or to apply our techniques to new areas.

- **Improving Performance via Hardware Virtualization**. Our approach requires information regarding data objects in the OS kernel, which caused non-trivial performance overhead in the current implementation. With the introduction of hardware virtualization techniques, we are interested in developing a new prototype with improved monitoring performance. Major hardware virtualization technologies provide page-table virtualization (a.k.a. nested page tables) to improve the performance of hardware virtualization. For instance, Intel's VT [93] provides Extended Page Tables (EPT) [94]. A similar technique in AMD virtualization technology (AMD-V [95]) is referred to as Rapid Virtualization Indexing (RVI) [96]. These features can be utilized to implement a reference checking mechanism of kernel memory accesses in an environment where guest operating systems are executed natively.

- **Kernel Debugging and Vulnerability Assessment**. Our approach provides in-depth information about data objects at runtime. It could be used to validate kernel operations and identify kernel vulnerabilities related to kernel memory. For instance, the information on heap objects previously was not available for an external monitor; therefore, validating a proper memory access to kernel heap memory was a challenging task. With the identification of kernel objects including heap objects, our system can check proper memory accesses and inspect memory-related vulnerabilities (e.g., kernel heap overflow).

LIST OF REFERENCES

LIST OF REFERENCES

[1] Eugene H. Spafford. The Internet Worm Program: An Analysis. *Computer Communication Review*, 19, 1989.

[2] Fred Cohen. Computer Viruses: Theory and Experiments. *Computers & Security*, 6:22–35, February 1987.

[3] Greg Hoglund and James Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[4] James P. Anderson. Computer Security Technology Planning Study, Volume I. Technical Report ESD-TR-73-51, ESD/AFSC, October 1972.

[5] James P. Anderson Co. Computer Security Threat Monitoring and Surveillance. Technical Report Contract 79F296400, February 1980.

[6] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, January 1998.

[7] H. Etoh. GCC Extension for Protecting Applications From Stack-smashing Attacks. `http://www.trl.ibm.com/projects/security/ssp/`. Accessed May 2011.

[8] Vendicator. Stack Shield: A "Stack Smashing" Technique Protection Tool for Linux. `http://www.angelfire.com/sk/stackshield/info.html`. Accessed May 2011.

[9] Ryan Riley, Xuxian Jiang, and Dongyan Xu. An Architectural Approach to Preventing Code Injection Attacks. In *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2009.

[10] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*, 2008.

[11] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of 21st Symposium on Operating Systems Principles (SOSP'07)*. ACM, 2007.

[12] c0ntex. Bypassing Non-executable-stack During Exploitation Using Return-to-libc. *Phrack Magazine*.

[13] Nergal. The Advanced Return-into-lib(c) Exploits: PaX Case Study. *Phrack*, 11(58). Article 4.

[14] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, pages 552–561. ACM, 2007.

[15] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of CCS 2008*, pages 27–38. ACM Press, October 2008.

[16] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium (Security'09)*, 2009.

[17] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting Return-Oriented Programming Malicious Code. In *Proceedings of the 5th International Conference on Information Systems Security (ICISS '09)*, 2009.

[18] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense against Return-oriented Programming Attacks. In *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing (STC'09)*, 2009.

[19] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending Embedded Systems against Control Flow Attacks. In *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code (SECUCODE'09)*, 2009.

[20] M. W. Lucas Davi and Ahmad-Reza Sadeghi. Ropdefender: A Detection Tool to Defend against Return-oriented Programming Attacks. 2010. Technical Report HGI-TR-2010-001.

[21] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating Return-oriented Rootkits with "Return-Less" Kernels. In *Proceedings of the 5th European Conference on Computer Systems (EUROSYS'10)*, 2010.

[22] Phrack Magazine. Linux On-the-fly Kernel Patching without LKM. `http://www.phrack.com/issues.html?issue=58&id=7`. Accessed May 2011.

[23] MITRE Corporation. Common Vulnerabilities and Exposures. `http://cve.mitre.org/`. Accessed May 2011.

[24] The Month of Kernel Bugs (MoKB) Archive. `http://projects.info-pull.com/mokb/`. Accessed May 2011.

[25] US-CERT. Vulnerability Notes Database. `http://www.kb.cert.org/vuls/`. Accessed May 2011.

[26] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, 2010.

[27] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauscheck, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Symposium on Network and Distributed System Security (NDSS'09)*, 2009.

[28] Mihai Christodorescu, Christopher Kruegel, and Somesh Jha. Mining Specifications of Malicious Behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, pages 5–14, New York, NY, USA, 2007. ACM Press.

[29] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and Efficient Malware Detection at the End Host. In *Proceedings of the 18th Usenix Security Symposium (Security'09)*, 2009.

[30] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.

[31] Mihai Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pages 169–186. USENIX Association, USENIX Association, August 2003.

[32] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the Principles of Programming Languages 1998 (POPL'98)*, San Diego, CA, January 1998.

[33] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[34] Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. Protection of Software-Based Survivability Mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN'01)*, pages 193–202, Washington, DC, USA, 2001. IEEE Computer Society.

[35] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009.

[36] Junghwan Rhee, R. Riley, Dongyan Xu, and Xuxian Jiang. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES'09)*, 2009.

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

[38] Troy Nash. An Undirected Attack against Critical Infrastructure. Technical report. http://www.us-cert.gov/control_systems/pdf/undirected_attack0905.pdf. Accessed May 2011.

[39] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL-injection Attacks and Countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering*, 2006.

[40] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Automatic Inference and Enforcement of Kernel Data Structure Invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC'08)*, pages 77–86, 2008.

[41] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, 2009.

[42] Nick L. Petroni, Jr., Timothy Fraser, AAron Walters, and William A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the 15th Conference on USENIX Security Symposium (USENIX-SS'06)*, 2006.

[43] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory. In *Proceedings of the 13th International Symposium of Recent Advances in Intrusion Detection (RAID 2010)*, Ottawa, Canada, September 2010.

[44] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Multi-Aspect Profiling of Kernel Rootkit Behavior. In *Proceedings of the 4th European Conference on Computer Systems (Eurosys'09)*, April 2009.

[45] Chaoting Xuan, John A. Copeland, and Raheem A. Beyah. Toward Revealing Kernel Malware Behavior in Virtual Execution Environments. In *Proceedings of 12th International Symposium on Recent Advances in Intrusion Detection (RAID'09)*, pages 304–325, 2009.

[46] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[47] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17:412–421, July 1974.

[48] Blaise Barney. *Introduction to Parallel Computing*. https://computing.llnl.gov/tutorials/parallel_comp/. Accessed May 2011.

[49] Thomas M. Chen and Jean marc Robert. The Evolution of Viruses and Worms. In *Statistical Methods in Computer*, 2004.

[50] Aleph One. Smashing The Stack for Fun and Profit. *Phrack*, 7(49). Article 14.

[51] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection from Printf Format String Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[52] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard™: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium*, 2003.

[53] Intel. Execute Disable Bit and Enterprise Security. `http://www.intel.com/technology/xdbit/index.htm`. Accessed May 2011.

[54] AMD. AMD Technologies: Enhanced Virus Protection (EVP). `http://www.amd.com/us/products/technologies/enhanced-virus-protection/Pages/enhanced-virus-protection.aspx`. Accessed May 2011.

[55] ARM. Instruction Set Architectures. `http://www.arm.com/products/processors/technologies/instruction-set-architectures.php`. Accessed May 2011.

[56] PAX PAGEEXEC Documentation. `http://pax.grsecurity.net/docs/pageexec.txt`. Accessed May 2011.

[57] Arjan van de Ven. New Security Enhancements in Red Hat Enterprise Linux v.3, Update 3. August 2004. `http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`. Accessed May 2011.

[58] OpenBSD. The OpenBSD 3.3 Release. May 2003. `http://www.openbsd.org/33.html`. Accessed May 2011.

[59] A Detailed Description of the Data Execution Prevention (DEP) Feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. `http://support.microsoft.com/kb/875352`. Accessed May 2011.

[60] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot–A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[61] Nick L. Petroni and Michael Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[62] Elia Florio. When Malware Meets Rootkits. `http://www.symantec.com/avcenter/reference/when.malware.meets.rootkits.pdf`. Accessed May 2011.

[63] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: Defeating Return-oriented Programming through Gadget-less Binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*, 2010.

[64] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[65] Ping Chen, Xiao Xing, Bing Mao, and Li Xie. Return-Oriented Rootkit without Returns (on the x86). In *Information and Communications Security (ICICS'10)*, 2010.

[66] Ping Chen, Xiao Xing, Bing Mao, Li Xie, Xiaobin Shen, and Xinchun Yin. Automatic Construction of Jump-oriented Programming Shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*, 2011.

[67] Tim Kornau. Return Oriented Programming for the ARM Architecture. 2010. Master's Thesis. Ruhr-Universität Bochum.

[68] Microsoft. Driver Signing for Windows. `http://technet.microsoft.com/en-us/library/cc784714.aspx`. Accessed May 2011.

[69] Deirdre Mulligan and Aaron K. Perzanowski. The Magnificence of the Disaster: Reconstructing the Sony BMG Rootkit Incident. 2008.

[70] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for Data Structures. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[71] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust Signatures for Kernel Data Structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, 2009.

[72] Zhiqiang Lin and Junghwan Rhee and Xiangyu Zhang and Dongyan Xu and Xuxian Jiang. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, February 2011.

[73] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software, Practice and Experience*, 1988. John Wiley & Sons, Inc.

[74] Marina Polishchuk, Ben Liblit, and Chloë W. Schulze. Dynamic Heap Type Inference for Program Understanding and Debugging. In *Proceedings of the 34th Annual Symposium on Principles of Programming Languages*. ACM, 2007.

[75] Jamie Butler. DKOM (Direct Kernel Object Manipulation). `http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf`. Accessed May 2011.

[76] Junghwan Rhee and Dongyan Xu. LiveDM: Temporal Mapping of Dynamic Kernel Memory for Dynamic Kernel Malware Analysis and Debugging. Technical Report CERIAS TR 2010-02, Purdue University, West Lafayette, Indiana, 2010.

[77] Andrea Lanzi, Monirul Sharif, and Wenke Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, 2009.

[78] Diego Zamboni. *Using Internal Sensors for Computer Intrusion Detection*. PhD thesis, Purdue University, West Lafayette, Indiana, 1995.

[79] Xuxian Jiang and Xinyuan Wang. "Out-of-the-Box" Monitoring of VM-based High-interaction Honeypots. In *Proceedings of Recent Advances in Intrusion Detection*, pages 198–218, September 2007.

[80] Greg Hoglund. Kernel Object Hooking Rootkits (KOH Rootkits). http://www.rootkit.com/newsread.php?newsid=501. Accessed November 2008.

[81] Jinpeng Wei, Bryan D. Payne, Jonathon Giffin, and Calton Pu. Soft-Timer Driven Transient Kernel Control Flow Attacks and Defense. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC'08)*, December 2008.

[82] Free Software Foundation. The GNU Compiler Collection. http://gcc.gnu.org/. Accessed May 2011.

[83] VMware. VMware Workstation, Multiple Operating Systems Including Linux on Windows. http://www.vmware.com/products/ws/. Accessed May 2011.

[84] Innotek. Virtualbox. http://www.virtualbox.org/. Accessed May 2011.

[85] Parallels. Parallels. http://www.parallels.com/. Accessed May 2011.

[86] Fabrice Bellard. QEMU: A Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[87] Nick L. Petroni, AAron Walters, Timothy Fraser, and William A. Arbaugh. FATKit: A Framework for the Extraction and Analysis of Digital Forensic Data from Volatile System Memory. In *Digital Investigation Journal 3(4):197-210*, 2006.

[88] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proceedings of 2008 USENIX Annual Technical Conference (USENIX'08)*, 2008.

[89] Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. Polymorphing Software by Randomizing Data Structure Layout. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'09)*, 2009.

[90] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium (Security'09)*, 2009.

[91] Hsin Pan and Eugene H. Spafford. Towards Automatic Localization of Software Faults. In *Proceedings of the 10th Pacific Northwest Software Quality Conference*, Oct 1992.

[92] Hsin Pan and Eugene H. Spafford. Heuristics for Automatic Localization of Software Faults. Technical Report SERC-TR-116-P, Purdue University, 1992.

[93] Intel. Intel®VT. http://www.intel.com/technology/virtualization. Accessed May 2011.

[94] Intel. Intel®Virtualization Technology: Hardware Support for Efficient Processor Virtualization. http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art01.pdf. Accessed May 2011.

[95] AMD. AMD-V™. http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx. Accessed May 2011.

[96] AMD. White Paper: AMD-V™Nested Paging. http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf. Accessed May 2011.

VITA

VITA

Junghwan Rhee obtained a B.E. degree from Korea University in 2003 and a M.S. degree from the University of Texas at Austin in 2005. He also pursued a Ph.D. degree in the Department of Computer Science at Purdue University under the direction of Professor Dongyan Xu. He is also affiliated with CERIAS, the Center for Education and Research in Information Assurance and Security. His research efforts focus on operating system security, malware analysis, virtualization, and cloud computing, specifically in the areas of kernel malware defense, virtualized infrastructure management, and reliability of distributed systems. In the Fall of 2011 he joined NEC Laboratories America at Princeton, New Jersey as a Research Staff Member.