FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl für Sicherheit in der Informatik

Full Virtual Machine State Reconstruction for Security Applications

Christian A. Schneider

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 23.04.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 26.08.2013 angenommen.

Abstract

System virtualization allows one to monitor, analyze, and manipulate the state of a virtual machine from the vantage point of the hypervisor. This method is known as *virtual machine introspection* (VMI). Various security mechanisms can be implemented by exercising the extensive control the hypervisor has over the virtual machines running on top of it, such as malware analysis, intrusion detection, secure logging, and digital forensics. As virtualization technology becomes increasingly ubiquitous in everyday computing, VMI represents a viable option to enhance system security from the hypervisor level. However, all systems that utilize VMI face a common challenge, namely the *semantic gap*. This challenge describes the disconnect between the low-level state that the hypervisor has access to and its semantics within the guest.

In this work, we explore the possibilities and implications of bridging the semantic gap to support security applications from the hypervisor level. We define a formal model for VMI that allows us to describe and compare such approaches in a uniform way. Using our model, we identify three common patterns for bridging the semantic gap and describe the properties of the corresponding implementations. Based on these findings, we propose a novel VMI framework that allows one to analyze the full virtual machine state in a universal way. We apply semantic knowledge of the operating system kernel to recreate the graph of kernel objects from guest physical memory. Since dynamic pointer manipulations such as type-casting and pointer arithmetic are prevalent in kernel code, we develop a type-centric, semantic source code analysis technique to identify such runtime manipulations in the C programming language. The results of this analysis augment the static type information and allow us to over-approximate the possible data types of pointer values, thus increasing the overall coverage of the kernel object graph in a fully automated fashion. However, not all instances of runtime dependent type manipulations can be detected this way. As a solution, our framework additionally supports the manual formulation of simple, yet powerful type manipulation rules. These rules allow the user to extend the type knowledge with his own expertise in a generic, re-usable way.

To give evidence for the effectiveness of our methods, we have implemented a prototype of this system as part of this work to perform virtual machine introspection. Our prototype faithfully reconstructs the graph of kernel objects of Linux guests almost perfectly. The identified objects can be exposed to arbitrary VMI applications for further analysis through a rich set of interfaces. From all the possible security applications that could work on the object graph, we choose malware detection as an example. The experimental evaluation shows that our approach is well suited to detect various kinds of kernel-level attacks, including those that are specifically designed to cover their tracks within the system.

Zusammenfassung

Systemvirtualisierung erlaubt es, den Zustand einer virtuellen Maschine auf der Ebene des Hypervisors zu überwachen, zu analysieren und sogar zu manipulieren. Diese Methode wird *Virtual-Machine-Introspection* (VMI) genannt. Basierend auf der umfangreichen Kontrolle, die der Hypervisor über eine virtuelle Maschine ausübt, können verschiedenste Sicherheitsanwendungen implementiert werden, wie etwa Malware-Analyse, Angriffserkennung, sichere Protokollierung oder digitale Forensik. Da Virtualisierungstechnik die Computerwelt zunehmend durchdringt, stellt VMI eine viel versprechende Technik dar, um die Systemsicherheit von der Ebene des Hypervisors aus zu verbessern. Eine Herausforderung, denen sich alle VMI-Ansätze jedoch stellen müssen, ist die sogenannte *semantische Lücke* (engl. *semantic gap*). Sie beschreibt die mangelnde Verknüpfung zwischen dem Systemzustand auf Hardware-Ebene, auf den der Hypervisor Zugriff hat, und seiner konkreten Semantik innerhalb des Gastes.

In dieser Arbeit erkunden wir die Möglichkeiten und deren Implikationen, wie diese semantische Lücke auf Hypervisor-Ebene für Sicherheitsanwendungen überbrückt werden kann. Wir definieren ein formales Modell für VMI, mit welchem sich solche Ansätze allgemein beschreiben und vergleichen lassen. Mit Hilfe dieses Modells identifizieren wir drei elementare Schemata, um die semantische Lücke zu überbrücken, und stellen die jeweiligen Eigenschaften einer entsprechenden Implementierung heraus. Basierend auf diesen Erkenntnissen schlagen wir ein neuartiges VMI-System vor, das es uns erlaubt, den vollständigen Zustand einer virtuellen Maschine auf eine universelle Art zu analysieren. Wir verwenden semantisches Wissen über den Betriebssystemkern, um den Graph von Kernel-Objekten aus dem physischen Arbeitsspeicher des Gastes zu rekonstruieren. Allerdings erweisen sich dynamische Pointer-Manipulationen, wie beispielsweise Umwandlungen von Datentypen und Pointer-Arithmetik, wie sie in Betriebssystem-Code keine Seltenheit sind, als problematisch. Deshalb entwickeln wir eine Typ-zentrische, semantische Analysetechnik für Quellcode in der Programmiersprache C, um solche Manipulationen zu erkennen. Die Ergebnisse dieser Analyse ergänzen die statischen Typ-Informationen und ermöglichen es, die potentiellen Interpretationen eines Speicherbereichs zu überapproximieren. Hierdurch lässt sich die Abdeckung des identifizierten Arbeitsspeichers auf komplett automatisierte Weise deutlich erhöhen. Dennoch können nicht alle Instanzen von Laufzeitmanipulationen durch diese Analyse erkannt werden. Aus diesem Grund unterstützt unser System zusätzlich die Formulierung komplexer Regeln für die Umwandlung von Datentypen. Diese erlauben es dem Benutzer, seine eigene Expertise in einer allgemeinen, wiederverwendbaren Form in das System einzubringen.

Um die Effektivität unserer Methoden unter Beweis zu stellen, haben wir im Rahmen dieser Arbeit einen Prototypen des vorgeschlagenen Systems für Virtual-Machine-Introspection entwickelt. Unser Prototyp ist in der Lage, den Kernel-Objekt-Graph für Linux-Gäste nahezu perfekt zu rekonstruieren. Die Kernel-Objekte können dabei beliebigen VMI-Anwendungen für die weitere Analyse über verschiedene Schnittstellen zugänglich gemacht werden. Aus der Menge der daraus möglichen VMI-basierten Sicherheitsanwendungen haben wir für eine Evaluation unseres Systems die Malware-Analyse als Beispiel ausgewählt. Unsere Experimente belegen, dass dieser Ansatz sehr gut geeignet ist, um verschiedene Arten von Angriffen auf den Betriebssystemkern zu erkennen, darunter solche, die speziell darauf abzielen, ihre Spuren möglichst gut zu verwischen.

Contents

List of Figures

List of Tables

Danksagung

Während meiner Forschung in den vergangenen Jahren hatte ich das große Vergnügen, in einer sehr angenehmen und konstruktiven Forschungsgruppe zu arbeiten. Gleichzeitig habe ich viele interessante und wunderbare Menschen kennengelernt, die mich auf die eine oder andere Weise in dieser Zeit unterstützt haben. Ich möchte die Gelegenheit nutzen und ihnen an dieser Stelle für ihre Unterstützung danken.

Zunächst möchte ich mich bei der Betreuerin meiner Arbeit, Frau Prof. Dr. Claudia Eckert, ganz herzlich bedanken für die großartige Unterstützung, für die freundschaftliche Atmosphäre, für die konstruktiven Diskussionen, für all die Ermutigungen und für ein allzeit offenes Ohr. Claudia, es war mir eine Ehre und ein großes Vergnügen, mit Dir zu arbeiten.

Weiterhin möchte ich Herrn Prof. Dr. Thorsten Holz ein herzliches Dankeschön dafür aussprechen, dass er meine Arbeit ohne Umschweife unterstützt und als Zweitgutachter bewertet hat. Seine hilfreichen Kommentare und unsere interessanten Diskussionen waren sehr förderlich und haben signifikant dazu beigetragen, diese Arbeit noch weiter zu verbessern.

Als nächstes möchte ich meinen ehemaligen Kollegen danken, zuletzt an der TU München am Lehrstuhl für Sicherheit in der Informatik und davor an der TU Darmstadt am Fachgebiet für IT-Sicherheit. Unsere fruchtbaren Diskussionen und der Ideenaustausch haben meine Arbeit maßgeblich unterstützt und geprägt: Jonas Pfoh, Sebastian Vogl, Thomas Kittel, Krista Grothoff, Florian Kohlmayer, Fatih Kilic, Frederic Stumpf, Christoph Krauß und Thomas Stibor. Darüber hinaus bedanke ich mich bei den Studenten, die verschiedene Beiträge zu meinem InSight-Projekt geleistet haben, nämlich Cornelius Diekmann, Hagen Fritsch und Dominik Meyer.

Nicht zuletzt gilt mein besonderer Dank meiner liebevollen Familie für ihre Unterstützung. Für meine Frau Jacqueline, vielen Dank für Dein Verständnis, Deine Ermutigung und Deine Liebe. Für meine Eltern, vielen Dank für Eure Liebe, Eure Unterstützung, und dass Ihr mir diesen Weg ermöglicht habt.

Chapter 1

Introduction

1.1 Motivation

Computer systems have proven to be lucrative targets for attackers. The increasing demand for and supply of valuable data and virtual assets has formed an underground economy with a considerable revenue [FJ08, ZHS+09, Nam09, HEF09]. As a consequence, we see a constant growth in the number of malicious software, also known as *malware*, as well as other forms of attacks on computer systems [CXS+05, Ric08, Kas08, Woo12]. Some attacks target specific applications or services on a system—for example, cross-site scripting attacks on web applications or SQL injection attacks. The most severe types of attacks, however, allow an adversary to execute arbitrary code on a target system.

If an attacker is able to execute code on a system, he typically crafts this code to serve two purposes [PO10, Eck12]: First, he tries to escalate the privilege level of the execution context if his current privileges do not suffice. Second, he tries to sustain his presence on the target system by changing the system configuration, by installing malware on the target, or by dynamically creating a process that performs malicious tasks on his behalf.

While it is trivial to detect that the configuration has been modified, the situation is much more difficult for malware processes running on a system. Such programs are designed to run permanently in the background while concealing their activities from the legitimate user. They avoid detection by hiding their files within the file system and by disguising their communication with the outside world as inconspicuous network traffic, for example. In order to achieve concealment, sophisticated malware often tampers with the *operating system* (OS) to change the behavior of system (library) functions.

The most sophisticated stealth techniques are implemented by so-called *rootkits* (cf. Section 2.5). The sole purpose of a rootkit is to hide its presence from the user by nesting itself deeply into the system [HB06, KS07, BKI07, Blu09, Eck12]. According to recent Internet security threat reports and malware trends, rootkits have become an increasing danger to system security [BDG+12]. In fact, a large fraction of real-world malware already contains some rootkit functionality [Woo12]. If this code is executing at the highest privilege level (e. g., as administrative user "root"), it is able to control and manipulate any part of the OS and even deceive or deactivate security software running on the system, such as virus scanners or firewalls [BY07]. As a consequence, the system in its entirety has to be considered as potentially compromised. In such cases, the legitimate user can no longer trust in any output or rely on any functions of the system, including security software such as anti-virus products. This is what makes rootkits particularly hard to detect and remove.

With the help of a technology called *system virtualization*, we can improve this situation for the defending side. In system virtualization, a thin software layer, namely the *hypervisor*, provides a virtual hardware interface which allows a *virtual machine* (VM) to run an entire operating system along with further user processes on top of it (cf. Section 2.1). Per definition, the hypervisor remains in control of all virtual hardware resources at all times [SN05]. Even if an adversary compromises the guest OS and gains full system privileges within a virtual machine, the hypervisor still has unrestricted and untainted access to the state of the guest (i. e., disk, memory, CPU registers, etc.).

The legitimate user can leverage this fact and bypass the untrustworthy guest. Using the hypervisor's privileges, the virtual machine can be monitored and to some extent even protected against attacks without having to rely on the guest OS itself, a technique known as *virtual machine introspection* (VMI) [GR03]. If rootkit detection for the guest is performed on the hypervisor level, it does not depend on potentially compromised functionality of the guest and can thus be more reliable.

In order to perform VMI, another challenge must be addressed first, namely the *semantic gap* [CN01]. It describes the disconnect between the low-level state the hypervisor has access to and its semantics within the guest. As a consequence, the analysis of a guest OS from the vantage point of the hypervisor requires the application of some sort of semantic knowledge to the low-level state of the virtual machine.

In this thesis, we will explore new ways of analyzing the entire state of a virtual machine from the hypervisor level using VMI. Our goal is to reconstruct the state of an OS kernel in the form of the kernel's data structures within the guest physical memory. The state of the guest's kernel is relevant to a number of security applications, such as malware analysis, intrusion detection, secure logging, or digital forensics. For example, if we understand how these kernel objects drive the kernel's behavior, we can compare it to the intended functionality in order to detect any undesired or even malicious modifications of the state, as they are performed by rootkits. Thus, it is imperative to bridge the semantic gap between hypervisor and guest OS to enable hypervisor-based security applications.

1.2 Problem Statement

Our ultimate goal is to reconstruct the kernel state using VMI in order to enable detection of system level compromises of a guest OS. This task provides several challenges that we address throughout this thesis. We describe each of these challenges in the following.

(P1) Extraction of Kernel Objects from Physical Memory

An OS kernel keeps its state in form of specific data structures in a dedicated part of the system's physical memory. We call these instances of data structures *kernel objects*.[1] The actual implementation varies among different hardware platforms, OSs, and kernel versions. One feature that almost all prevalent OSs and hardware platforms share is the concept of virtual memory. This technique introduces an abstraction layer to the physical memory of the hardware to ease memory management and isolate the address spaces of concurrently running processes.

In order to extract a kernel object from physical memory, the size, layout, and address of the object must be known. However, a modern OS kernel is a very complex piece of software that uses a large amount of data types and variables. For example, the Linux kernel version 3.5, which was released in July 2012, consists of more than 15.5 million lines of code [Lee12], and the Windows XP operating system was compiled from 45 million lines of code [Mic]. These numbers illustrate that managing type information in an efficient and usable way becomes a challenge in itself.

In addition to managing the data types and variables, we must be able to perform the same virtual-to-physical address translation the memory management unit of the kernel or hardware platform applies in order to actually access the data values in the (guest) physical memory.

(P2) Identification and Recreation of Dynamic Pointer and Type Manipulations

Even today, large parts of modern OS kernels are written in the C programming language. C is known to provide the programmer with a high degree of freedom and allows him to write highly optimized code; however it lacks convenient features of modern programming languages, such as native variable length arrays, object orientation, polymorphism, templates or generics, et cetera. It is common practice amongst C programmers to simulate these features with custom code that makes use of pointer arithmetic, type casts, and '`struct`' or '`union`' data types, often wrapped in preprocessor macros for easier usage.

[1]In object-oriented programming languages, an "object" is an instance of a class type. However, large parts of OS kernels are written in C. In this thesis, we extend the term "object" to any structured chunk of memory, even if it only contains a single 32-bit integer value.

This code leads to dynamic, runtime dependent kernel behavior which makes it much harder to reliably determine the type and location of kernel objects in memory. Traditional type information as used by debuggers, for example, gives us the strictly typed view of a compiler that only allows one single unambiguous interpretation of memory. Such a static view is insufficient for VMI applications, as the kernel may use the same objects in different ways depending on the context. The challenge here is to identify the objects and contexts of dynamic behavior and to recreate this behavior when extracting kernel objects from memory.

(P3) Exposing the Kernel State to a Flexible VMI Framework

Our goal of detecting system compromises does not dictate to follow a particular approach. In fact, we want to be able to perform various kinds of analyses and apply novel methods to the system state that support our goal. Additionally, our detection methods should be portable between different kernel versions and hardware architectures. Thus, they must be able to cope with differences in the type information and memory extraction process.

These requirements pose some engineering challenges on a VMI framework that need to be addressed. Since the resulting outcome of the detection methods is crucial for security, the VMI framework must not only be reliable and fault tolerant; even more critically, the resulting framework must not introduce new vulnerabilities to the host system.

1.3 Contribution

In this thesis, we explore the challenges presented in the previous section and contribute to their solution. As part of this work, our efforts have lead to a flexible and powerful VMI framework for easy and safe development of VMI applications that we call *InSight* [ins]. The focus of its application is the reliable detection of kernel-level attacks within virtual machines. Unlike previous work in the field of VMI, we strive toward making the entire kernel state available with a high degree of automation and as little required domain knowledge as possible.

Throughout this work, we describe the methods we developed for achieving a high kernel state coverage, both with and without incorporating the knowledge of human experts. We also highlight the interesting aspects of the implementation of these methods within InSight. In addition, we evaluate our methods and show the effectiveness and efficiency of our implementation for malware detection. The following list summarizes the contributions of this thesis:

(C1) Definition of a Formal Model for VMI

Based on a joint work with Jonas Pfoh, we define a formal model to describe and compare arbitrary VMI approaches. Our model shows that there is only a limited number of ways to actually bridge the semantic gap, each of which has unique properties. Using our model helps to understand these properties and their implications for a particular VMI application. This contributes to the solution of the challenges (P1) and (P3).

(C2) Identification of View Generation as a Unique Challenge

We see a demand for a VMI framework that provides some basic functionality for common introspection tasks. With our work, we are among the first to address the semantic gap challenge in a generic way. We call the act of bridging the semantic gap *view generation* (cf. Section 3.2.2). This demand has inspired us to create such a VMI framework and thus indirectly supports challenge (P3).

(C3) Development of a Universal VMI Framework

As part of our research, we have created a universal VMI framework, called InSight. It maintains a type database of data structures to read corresponding kernel objects from a physical memory image for Linux kernels on the Intel IA-32 and AMD64 architecture. These kernel objects are accessible by the user through a rich set of interfaces, including a command shell and a JavaScript engine. With these features, InSight supports the safe and easy development of VMI applications and contributes to challenges (P1) and (P3).

(C4) Identification of Dynamic Pointer and Type Manipulations

We describe a method for a static analysis of full C source code. The focus of our analysis is the implicit and explicit type flow within C statements. We call this type-centric code analysis the *used-as analysis*. For every global variable and structure or union field in the code, our analysis provides the answer to two questions: First, is this variable or field used as a type that differs from its declaration? And, second, how do we need to transform a source value (a field or variable) to derive the next object's address? The resulting answers help to overcome challenge (P2).

(C5) Supporting Rule-based Type Knowledge

We have implemented the used-as analysis as part of our VMI framework, InSight, to establish used-as relations between data types. The information resulting from analyzing the source code of the introspected kernel are expressed as *type knowledge rules* that augment the type database. With this extension, InSight is able to retrieve objects that are referenced by generic pointers and even '`void`' pointers in a completely automatic and reliable fashion without requiring any manual annotation.

As a supplement to the derived type knowledge, the user can express his own expertise in the form of hand-written type rules. The type rule engine of InSight identifies the applicable used-as and manual rules for an object at runtime and evaluates them based on their priority and context. With only very few manual additions, the memory coverage can be improved even further while resolving possible ambiguities at the same time. This approach contributes to the solution of challenge (P2).

(C6) Demonstration of the Effectiveness and Efficiency of our Framework for Kernel-level Malware Detection

The implementation of the proposed methods generates a meaningful view of an introspected OS kernel's state. We give evidence of this claim and describe several experiments we conducted to detect various kernel-level attacks using the state information that InSight provides. The results show that with our VMI framework, it is possible to reliably detect different kinds of attacks with minimal efforts.

1.4 List of Publications

This thesis contains some new and previously unpublished material, though it is substantially based on work that has been published in scientific, peer-reviewed conferences and workshops. We will briefly introduce these publications and their contributions to this work. For the sake of completeness, we will include some papers relevant to the field of VMI that we have contributed to, but which are not covered in this thesis.

The formal model described in Chapter 3 is based on joint work with Pfoh and Eckert [PSE09]. This paper directly relates to the contribution (C1). We presented our VMI framework InSight for the first time in a publication together with the same authors [SPE11]. It includes the contributions (C2) and (C3). We cover the contents of this paper in much greater detail in Chapter 6. Chapter 7 is also based in large part on published work [SPE12]. This paper describes our used-as analysis for detecting pointer and type manipulations on the source code level. Its contributions are summarized in (C4) and (C5).

In addition to the previously listed contributions, we have done research on different methods of bridging the semantic gap which are not included in this thesis. Together with Pfoh and Eckert, we investigated the possibilities and limitations of generating views when relying solely on hardware state information [PSE10]. Based on these findings, we proposed an efficient VMI-based tool for system call tracing [PSE11]. We used the collected system call information to detect malware with machine learning methods based on support vector machines and a specially adapted string kernel function [PSE13].

In joint work with Vogl et al., we followed another approach to address the semantic gap challenge: We injected kernel modules from the hypervisor level into the context of a guest operating system very efficiently [VKSE13]. By using sophisticated isolation techniques, the modules were able to access all state of the guest OS and even call kernel functions without sacrificing their overall security.

References

[PSE09] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec '09)*, pages 1–10, Chicago, Illinois, USA, November 2009. ACM Press.

[PSE10] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Exploiting the x86 architecture to derive virtual machine state information. In *Proceedings of the 4th International Conference on Emerging Security Information, Systems and Technologies*, pages 166–175, Venice, Italy, July 2010. IEEE Computer Society. Best Paper Award.

[PSE11] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, volume 7038 of *Lecture Notes in Computer Science*, pages 96–112. Springer, November 2011.

[SPE11] Christian Schneider, Jonas Pfoh, and Claudia Eckert. A universal semantic bridge for virtual machine introspection. In Sushil Jajodia and Chandan Mazumdar, editors, *Information Systems Security*, volume 7093 of *Lecture Notes in Computer Science*, pages 370–373. Springer, December 2011.

[SPE12] Christian Schneider, Jonas Pfoh, and Claudia Eckert. Bridging the semantic gap through static code analysis. In *Proceedings of EuroSec'12, 5th European Workshop on System Security.* ACM Press, April 2012.

[PSE13] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Leveraging string kernels for malware detection. In *Proceedings of the 7th International Conference on Network and System Security*, Lecture Notes in Computer Science. Springer, June 2013.

[VKSE13] Sebastian Vogl, Fatih Kilic, Christian Schneider, and Claudia Eckert. X-TIER: Kernel module injection. In *Proceedings of the 7th International Conference on Network and System Security*, Lecture Notes in Computer Science. Springer, June 2013.

1.5 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2, we lay out some foundations that are relevant for our work. In addition, we establish some terms and definitions used in the thesis.

Chapter 3 introduces our formal model for virtual machine introspection. This model is helpful for comparing different VMI approaches and reasoning about their properties. In particular, we explain the out-of-band delivery pattern for view generation which is the focus of this work.

With this basis, we give an overview of work related to this thesis in Chapter 4 and classify the different approaches according to our model. The comparison of our problem statement and our contributions with such work separates this thesis from previous research.

In Chapter 5, the challenges for out-of-band delivery based view generation are presented. In particular, we give several examples of Linux kernel code fragments that render external view generation a hard problem.

Chapter 6 introduces our VMI framework, *InSight*. This framework adopts the out-of-band delivery pattern and overcomes several of the challenges presented in Chapter 5. It makes development of new VMI approaches an easy and intuitive task. We detail several interesting aspects of InSight and sketch some useful applications.

In Chapter 7, we propose a new pointer analysis technique for full C source code, called the *used-as* analysis. It identifies instances of dynamic pointer and type manipulations on the source code level. The purpose of the analysis is to address additional challenges outlined in Chapter 5 by extracting additional type knowledge from the code base that would otherwise require expert knowledge and manual work. The implementation of this method as an extension to InSight involves some engineering challenges which are described along with our solution.

We evaluate the effectiveness and efficiency of our VMI framework in Chapter 8. We use InSight to construct a graph of kernel objects for the Linux kernel and compare this graph to the objects the kernel has actually allocated. In addition, we use the external view generation to detect several Linux rootkits which are not directly detectable from within the compromised system itself.

Finally, Chapter 9 summarizes the thesis and includes an outlook for further research in this field.

Chapter 2

Foundations

This chapter lays out some foundations that are used throughout this thesis. We cover several technologies that we build upon and familiarize the reader with the terminology required to describe our work.

Chapter overview. We begin with on overview of the concept of system virtualization in Section 2.1. Our goal is to reason about the state of a virtual machine from the outside, a technique called virtual machine introspection. This technique is introduced in Section 2.2. Closely related to virtual machine introspection is the so-called semantic gap challenge. We explain this gap and its origin in Section 2.3. Eventually, we want to be able to detect rootkits and kernel-level attacks that are installed or carried out inside of a virtual machine. Therefore, we describe what kernel-level attacks are in Section 2.4 and distinguish them from rootkits in Section 2.5.

2.1 System Virtualization

In computer systems, there exist various types of virtualization depending on the resources or components that are being virtualized: memory, storage, an interface library, or an entire hardware platform. Virtualization always introduces an abstraction layer for an existing interface to real resources or components of a system.[1] These abstract resources are then accessible within the virtualized system. We refer to the real system that provides the resources as the *host*, while the virtualized system that uses the virtualization interface is called the *guest*.

Popek and Goldberg [PG74] define virtualization formally as an isomorphism $f : C_r \to C_v$ which maps any real (host) state $S_i \in C_r$ to a corresponding virtual (guest) state $S_i' \in C_v$ as follows. For any instruction sequence e_i that transforms the host's state S_i to $e_i(S_i) = S_j$, there exists a corresponding instruction sequence e_i' for the guest such

[1]Note that this abstraction does not necessarily imply a reduced complexity or level of detail.

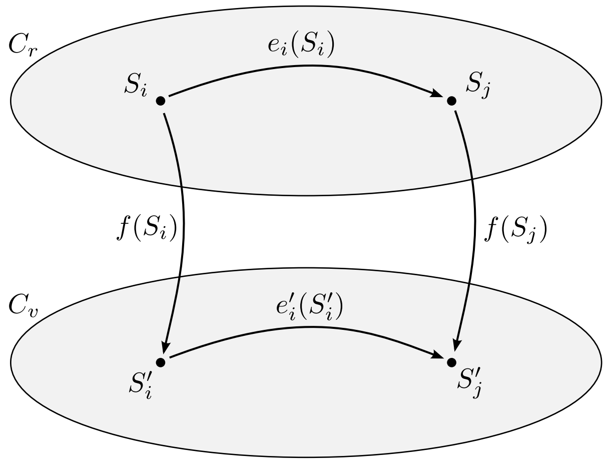


Figure 2.1: Virtualization constructs an isomorphism f which maps the set of host states C_r to the set C_v of states of the guest.

that $f(e_i(S_i)) = e'_i(f(S_i))$. In other words, all instruction sequences of the guest can be mapped to instruction sequences of the host that reproduce the corresponding state changes of the guest. This relation is illustrated in Figure 2.1.

For our work, we only concern ourselves with *system virtualization*, which describes the abstraction of a complete hardware interface of a computing platform as a *virtual machine* (VM). Smith and Nair [SN05] define system virtualization from the vantage point of the virtual machine as follows:

> "A *system virtual machine* provides a complete system environment. This environment can support an operating system along with its potentially many user processes. It provides a guest operating system access to underlying hardware resources, including networking, I/O, and, on the desktop, a display and graphical user interface."

The virtual hardware interface is provided by a software layer called the *virtual machine monitor* (VMM) or *hypervisor*. It intercepts all guest accesses to privileged resources, checks them for correctness, and executes them on behalf of the guest. Thus, the hypervisor retains control of all privileged resources and has complete access to them at all time. Those privileged resources are comprised of the CPU, the memory, and all connected I/O devices, most notably the disk storage and network interface. We will briefly describe how each of these resources can be virtualized in the following.

2.1.1 CPU Virtualization

CPUs distinguish between two (or sometimes more) privilege levels for their instructions and according modes of operation. The *user mode* is the least privileged mode and allows the execution of *non-privileged instructions*, such as basic arithmetic, branching,

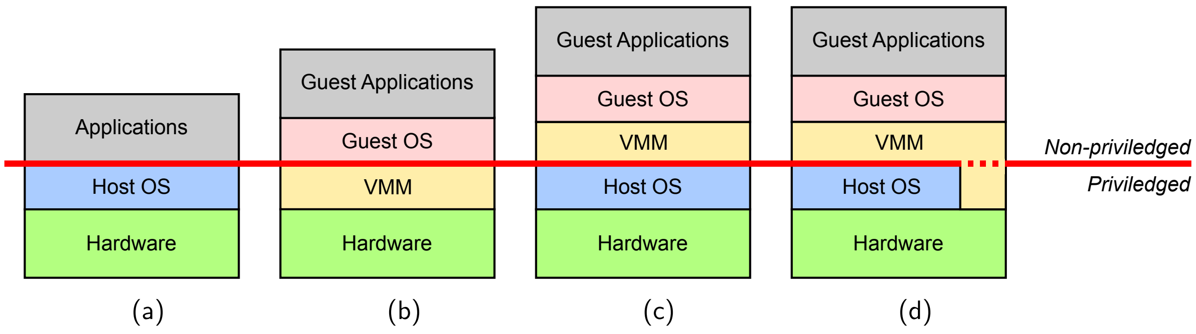


Figure 2.2: Different types of system virtualization: (a) none, (b) bare-metal virtualization, (c) user-mode hosted virtualization, (d) dual-mode hosted virtualization

and regular memory accesses. The code of user-land applications typically runs in user mode. In addition, there are the *privileged instructions* that change the behavior of the CPU, the memory management unit, or other privileged components. These management instructions are only allowed when the CPU runs in *system mode* and are reserved for operating system code. When an application tries to execute a privileged instruction while the CPU is in user mode, the CPU handles it in one of two different ways, depending on the instruction and CPU type: It can either perform a *trap* to the operating system, that is, it transitions into system mode and invokes a special handler function that the operating system has installed previously, or it effectively ignores the instruction and does not change its state.

In order to virtualize a CPU, the hypervisor has to support both privileged and non-privileged instructions of a virtual machine. A straightforward solution would be to implement the virtual CPU entirely in software and emulate all instructions. This approach obviously requires multiple instructions of the physical CPU to emulate a single instruction of its virtual counterpart, leading to a significant performance degradation. To achieve better performance, another strategy can be applied: One allows all non-privileged instructions to execute directly on the host CPU and have the hypervisor emulate only the privileged instructions. This method is called *direct native execution.*

When comparing the responsibilities of an operating system and a hypervisor, it becomes clear that they both perform similar tasks regarding resource management and handling of privileged CPU instructions. Virtualization can be classified according to the privilege level at which the hypervisor is executing, as depicted in Figure 2.2.

If no virtualization is performed, the operating system runs in privileged mode and is responsible for managing all resources of the host (Fig. 2.2a). The classic approach to virtualization is to execute the hypervisor in privileged mode as the lowest layer of the software stack instead of the operation system (Fig. 2.2b). This type of virtualization sometimes is called bare-metal virtualization because the hypervisor is running directly on the hardware.

In hosted virtualization, the hypervisor runs as an application within a host operating system. This approach allows for a much simpler code base of the hypervisor itself since it can rely on the host OS to provide essential functionality, such as bootstrapping routines, hardware drivers, and management services. If the hypervisor runs entirely in user mode, the host OS is the only code that runs in privileged mode (Fig. 2.2c). In dual-mode hosted virtualization, the hypervisor is split up into a user-land application and a (typically small) kernel module (Fig. 2.2d). Only the latter component is executed in privileged mode within the kernel. Here the work of emulating the privileged instructions is split up between the host OS and the hypervisor.

2.1.2 Memory Virtualization

Most hardware architectures and operating systems support the concept of *virtual memory*, an abstraction layer for the physical memory: The operating system splits up the physical memory in fixed size chunks, the *pages*, and maintains one *page table* per process which maps virtual pages to physical memory. This gives each process the illusion of living in its own, unique address space. During execution, the *memory management unit* (MMU) is responsible for translating virtual to physical addresses [Tan07]. For performance reasons, the MMU is usually a part of the CPU itself to avoid having to involve the operating system for every single memory access.

Hypervisors can make use of the virtual memory and the MMU to support direct native execution. The key idea is to virtualize the page table pointer register of the CPU by trapping all guest accesses to that register to the hypervisor [SN05]. For each page table that the guest OS tries to install in that register, the hypervisor maintains a *shadow page table* which maps guest-virtual to host-physical addresses. This page table is the one consulted by the MMU during execution of the corresponding process within the guest. To keep the shadow page tables in sync with the guest page tables, the hypervisor also intercepts all updates the guest performs on a page table and applies these changes to the corresponding shadow page table accordingly.

Due to the increasing popularity of system virtualization in server and desktop environments, the two big processor manufactures for x86-based CPUs, Intel and AMD, have introduced several virtualization extensions to their processors' instruction sets to better support reliable and high-performance virtual machines [UNR+05, NSL+06, AMD05]. But also in the world of embedded systems we start to see hardware extensions for virtualization to established CPU architectures, such as introduced by ARM for their Cortex-A15 processor [ARM12]. All of these extensions include support for *extended* or *nested page tables* [Maf08], sometimes also called *second-stage translation* [VH11]. The basic idea is to add a second level of indirection to the MMU's address translation scheme. This enables the MMU to translate guest-virtual to guest physical to host-physical addresses all in hardware and without intervention of the hypervisor.

2.1.3 Input/Output Virtualization

A virtual machine is equipped with a set of virtual devices that the guest OS interacts with. Depending on the virtualized hardware architecture, such interaction functions either trough specific CPU instructions or through read and write accesses of specific memory regions (memory-mapped I/O).

Accessing any hardware device represents a privileged instruction, so all device accesses must be executed in privileged mode. If they are executed in user mode, they trap to the hypervisor. Since the guest OS is always running in non-privileged mode (see Fig. 2.2), all device commands automatically trap to the hypervisor which makes their virtualization fairly straightforward. As a consequence, the guest OS never accesses the physical hardware directly; the hypervisor maps the trapped requests to physical devices or to resources of the host OS, depending on the virtualization type.

The technique that is used to map the guest's device accesses to host hardware depends on the characteristics of the device being virtualized. Some devices are intended for being used exclusively, for example, the keyboard or the display. Such dedicated devices are typically multiplexed by the hypervisor in some way. Continuing the previous example, the keyboard input can be redirected to one virtual machine at a time. The display can either be multiplexed in the same fashion, or the physical screen can be split into multiple virtual screens showing all virtual machines' displays at the same time. Other devices lend themselves to being partitioned for multiple guests. A hard disk, for instance, can be partitioned into several chunks, each of which is used by only one virtual machine. Finally, some devices can be shared at a fine time granularity, creating the illusion that each guest has exclusive access to it. A network card is an example of such a device. The hypervisor simulates a virtual network card for each VM, buffers the packets that the guests transmit, sends them out over the physical network card, receives the packets on behalf of the guests, and delivers them to the corresponding guests using virtual device interrupts.

2.2 Virtual Machine Introspection

The hypervisor controls the execution of the virtual machines running on top and provides strong isolation between them. Madnick and Donovan were among the first to describe the relevance of this isolation property for overall system security [MD73]. In addition, the hypervisor is able to observe and even modify the state and thus the behavior of the guest. Garfinkel and Rosenblum [GR03] characterize this authority as the combination of three properties:

1. *Isolation* refers to the property that the hypervisor runs independent from the guest, and (ideally) the guest cannot change other guests' or the hypervisor's behavior in any irregular way.

2. *Inspection* refers to the fact that the hypervisor is able to examine the entire state of the guest without having to rely on the guest's cooperation or trustworthiness.

3. *Interposition* refers to the hypervisor's ability to manipulate the guest's state or to inject operations into the normal running of the guest.

These properties lend themselves to implement new functionality for supervision at the hypervisor level. The act of using virtualization for the purpose of analyzing and/or manipulating a guest's behavior from the vantage point of the hypervisor is called *virtual machine introspection* (VMI) [GR03].

2.2.1 VMI for Security Applications

Security mechanisms such as digital forensic tools and intrusion detection systems rely on the ability to observe the system state. As such, these mechanisms clearly benefit from the properties of VMI [CTS+08, HN08, NBH09, DGPL11]. For example, even if an attacker has gained full control over the operating system running inside a guest, the strong isolation through the hypervisor restricts the attacker's code to this guest and leaves only a minimal attack surface to escalate his privileges across the virtual machine's boundaries. Since the hypervisor still has access to the complete and untainted guest state, all monitoring or enforcement of security policies on the hypervisor level is unaffected by any doings of the attacker and thus very hard to evade.

A hypervisor manages a guest's state as part of the state of the host system by mapping accesses to resources of the virtual machine to physical counterparts of the host. This allows the hypervisor to save and restore the state of a virtual machine and is another benefit of the inspection property. For example, it enables one to reset a compromised system to a known-good state or to preserve a machine's state for offline forensic analysis in case of an incident [DKC+02].

To retain control of all host resources, the hypervisor interposes privileged operations within the virtual machine and emulates their effect on behalf of the guest (cf. Section 2.1). This mechanism can be exploited to interpose particularly interesting instructions [PSE10] and implement various security policies [VKSE13] . For example, it allows one to monitor the interactions between a guest operating system and its userland applications by tracing the system calls that the processes invoke [PSE11]. Such interactions have proven to be relevant for assessing a system's state and for detecting system attacks [KH97, HFS98, RTWH09, PSE13].

2.2.2 Limitations

To be able to rely on the advantages that VMI brings, the hypervisor as well as any privileged VM involved in a VMI application must not be vulnerable to compromise. Of course, this is an ideal assumption that cannot be guaranteed. In fact, several vulnerabilities in various virtualization systems have already been discovered [Fer06]. Nevertheless, due to the strong isolation provided by the hypervisor, the attack surface of VMI-based approaches that execute outside of the inspected VM is significantly reduced compared to software running inside of it.

2.3 The Semantic Gap

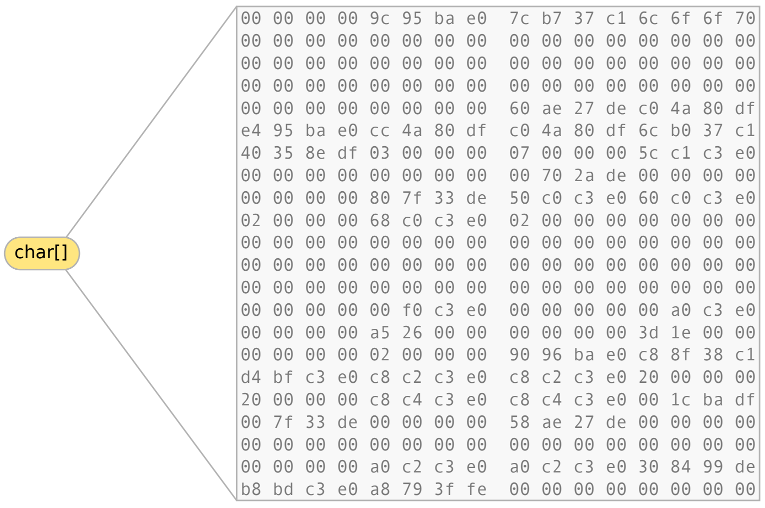
When inspecting a VM from the vantage point of the hypervisor, the entirety of the VM state is accessible. This state consists of the CPU registers, the main memory (containing data and code), the disk storage, as well as the state of all attached devices, such as network adapters or sound cards. However, since the hypervisor is only exposing a virtual hardware interface, which is a very basic, low-level interface, it has only a limited understanding of the semantics of this state. This inherent lack of semantic information in the hypervisor has been characterized by Chen and Noble as the *semantic gap* [CN01].

Considering a hypervisor providing the guest with an x86-based hardware interface[2], for instance, the semantics of the virtual CPU's control registers are precisely specified by the CPU's instruction set architecture (ISA) [Int09]. Thus, it is straightforward to interpret this part of the guest's state.

In contrast, the usage of the memory is completely left over to the operating system running inside the virtual machine. As a consequence, from the perspective of the hypervisor and without further knowledge, the semantics of the memory for the guest OS are unknown. This lack of understanding of the state is illustrated in Figure 2.3a: When accessing the VM's memory, the hypervisor only sees raw, binary data and does not now how it is used by the guest OS or how it affects its behavior. For VMI applications, it is essential to bridge the semantic gap and resemble the guest's view onto the memory, as shown in Figure 2.3b. With knowledge about the data structures used by the kernel and their layout in memory, the kernel state can be reconstructed from the hypervisor level.

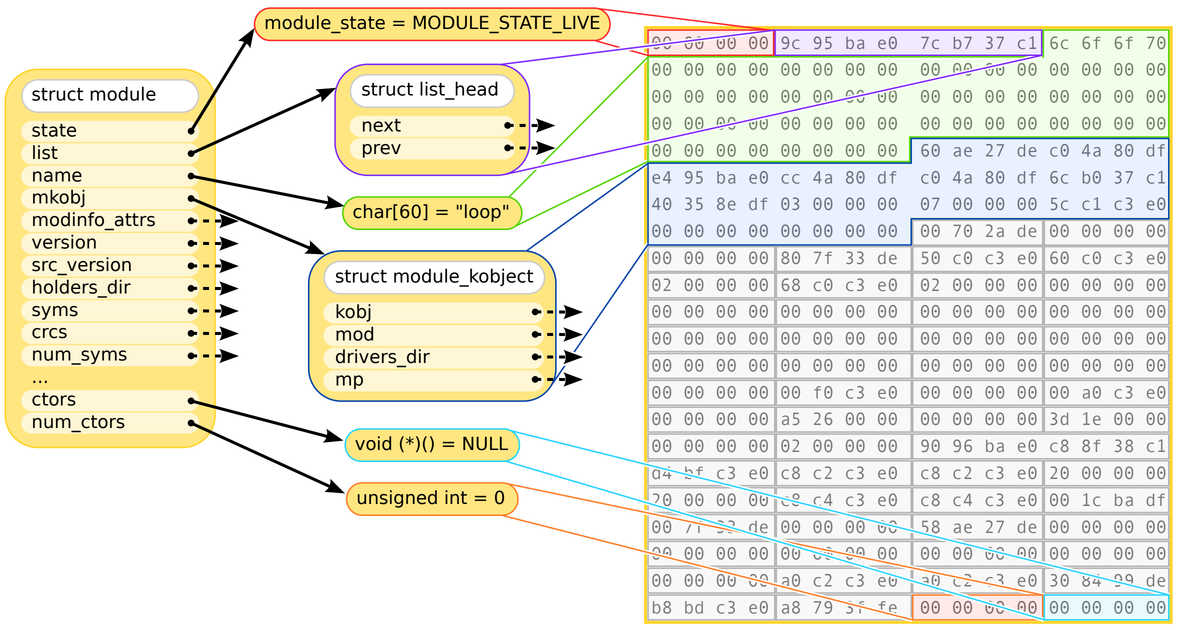
It is obvious that bridging the semantic gap requires application of knowledge about the virtual hardware architecture, the guest OS, or both. Recall that the hypervisor remains in control of and has access to all virtual hardware and resources at any time (cf. Section 2.1). From this follows that, with *complete* knowledge about the virtual hardware, the guest OS, and all software running inside the virtual machine, one can theoretically generate a view of the guest's state that is as complete as the internal view of the guest OS itself. In practice, however, even with all that information at hand this is hardly feasible: The view that can be acquired from the perspective of the hypervisor will always be a compromise between completeness, performance, and other implementation dependent trade-offs.

[2]When talking about the x86 architecture in this thesis, we include both the 32-bit architecture Intel IA-32 as well as the 64-bit extension Intel 64/AMD64, unless otherwise stated. However, we do not consider the Intel Itanium architecture (IA-64).



```
00 00 00 00 9c 95 ba e0  7c b7 37 c1 6c 6f 6f 70
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  60 ae 27 de c0 4a 80 df
e4 95 ba e0 cc 4a 80 df  c0 4a 80 df 6c b0 37 c1
40 35 8e df 03 00 00 00  07 00 00 00 5c c1 c3 e0
00 00 00 00 00 00 00 00  00 70 2a de 00 00 00 00
00 00 00 00 80 7f 33 de  50 c0 c3 e0 60 c0 c3 e0
02 00 00 00 68 c0 c3 e0  02 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 f0 c3 e0  00 00 00 00 00 a0 c3 e0
00 00 00 00 a5 26 00 00  00 00 00 00 3d 1e 00 00
00 00 00 00 02 00 00 00  90 96 ba e0 c8 8f 38 c1
d4 bf c3 e0 c8 c2 c3 e0  c8 c2 c3 e0 20 00 00 00
20 00 00 00 c8 c4 c3 e0  c8 c4 c3 e0 00 1c ba df
00 7f 33 de 00 00 00 00  58 ae 27 de 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 a0 c2 c3 e0  a0 c2 c3 e0 30 84 99 de
b8 bd c3 e0 a8 79 3f fe  00 00 00 00 00 00 00 00
```

(a) Hypervisor's perspective



(b) Guest OS's perspective

Figure 2.3: Illustration of the semantic gap when accessing the virtual machine's memory. (a) From the perspective of the hypervisor, the semantics of the memory is unknown. (b) The guest operating system has a clear understanding of the location and layout of its data structures. When this view can be reconstructed from the hypervisor level, the semantic gap has been successfully bridged.

2.4 Kernel-Level Attacks

One of the properties of VMI is the inspection property, which describes that the hypervisor is able to examine the entire state of a guest "as-is", that is, complete and untainted by any adversary who might have taken control of the guest OS. This is the reason why VMI-based approaches are most promising for defending against attacks that target the kernel directly: After the kernel itself has been compromised, any detection or removal mechanism cannot rely on kernel functions anymore since the attacker controls them already. He may modify these functions to return false reports about the kernel's state or simply deny the termination of a malicious process. If we apply a VMI-based detection or removal mechanism and choose the approach properly (cf. Section 3.3), we leave the attacker's range of influence and escape his control.

When using the term *kernel-level attack*, we refer to activities with (mostly) malicious intent that are carried out on the kernel level and leave traces in the kernel's state[3]. Examples of such attacks include modifying a function pointer in the system call table to perform key logging, removing a kernel object from a list by manipulating the list pointers to hide its presence, or overwriting the user identifiers associated with a process to elevate the process's privileges.

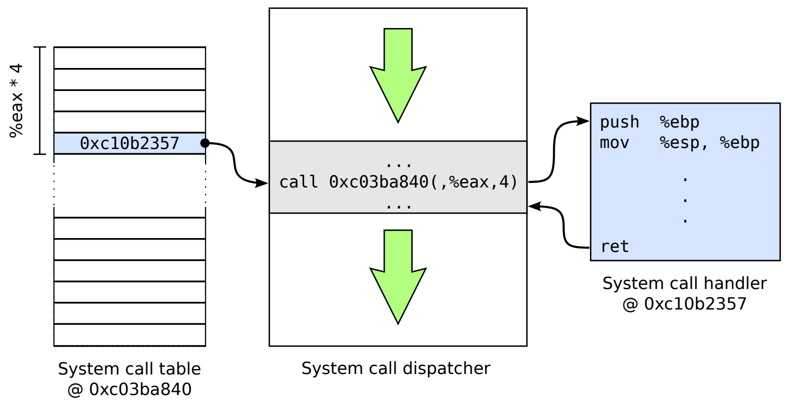
One prominent incarnation of software that performs kernel-level attacks are rootkits [HB06, Blu09, Eck12]. Rootkits are specialized in providing concealment for themselves and usually for an additional (typically malicious) payload that is supposed to run undetected on a system, for instance, to provide remote access or to collect intelligence. However, not every kernel-level attack has to be performed by a rootkit and not every rootkit has to attack the kernel. We give a clear definition of rootkits and differentiate them from kernel-level attacks in Section 2.5. Nevertheless, we will use kernel-level rootkits as a running example throughout this section since it is convenient and intuitive to explain kernel-level attacks from the perspective of a rootkit author.

The goal of a kernel-level attack is to make the kernel behave in a way the attacker desires. Since this desired system behavior is contrary to the normal operation of the kernel, it follows that the attacker has to modify the state of the kernel to achieve his goal. There are basically two elements of the kernel he can alter: kernel code or data. For both of these attack vectors, we describe some fundamental techniques that are commonly used to subvert OS kernels in the following.

2.4.1 Data-based Attacks

The kernel organizes its state in form of instances of data types in memory, the so-called kernel objects. The information contained within these objects steer the kernel's course in large parts. Other important information resides in locations dependent on the hardware architecture, such as specific CPU registers. If an attacker can tamper

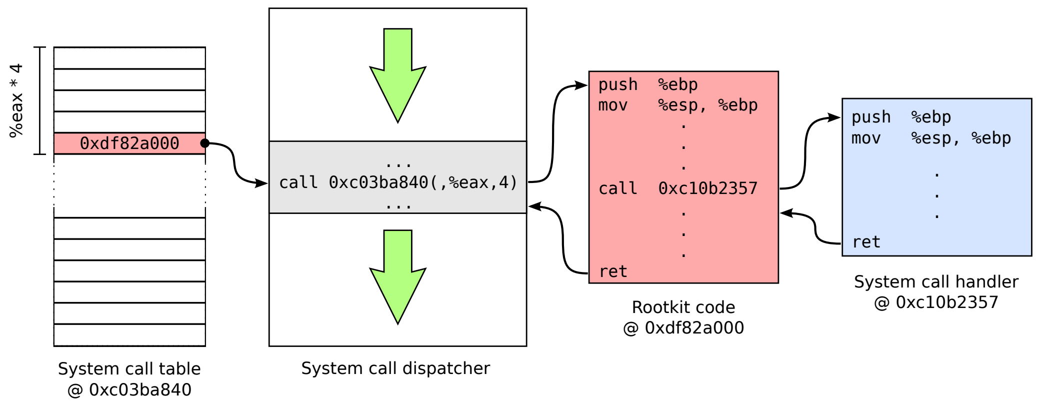
[3]Recall that the state includes not only the data but also the code of the running kernel.



```
...
call 0xc03ba840(,%eax,4)
...
```

```
push  %ebp
mov   %esp, %ebp
      .
      .
      .
ret
```

(a) Before



```
...
call 0xc03ba840(,%eax,4)
...
```

```
push  %ebp
mov   %esp, %ebp
      .
      .
      .
call  0xc10b2357
      .
      .
      .
ret
```

```
push  %ebp
mov   %esp, %ebp
      .
      .
      .
ret
```

(b) After

Figure 2.4: Control flow of a *hook* in the system call table, (a) before the modification and (b) after the hook has been installed.

with this data, he can manipulate the system's behavior to his advantage.

2.4.1.1 Function Pointer Hooking

Callback functions and call tables in the kernel are implemented using function pointers. These pointers point to the entry point of the functions in charge to handle specific events. A common technique to attack a kernel is to *hook* function pointers by swapping the address the pointer holds with the address to some code under the control of the attacker. The attacker's code will then perform the intended actions on each event, for example, logging the keystroke in case of a keyboard interrupt. Eventually, the malicious code will call the original function, probably with modified parameters or filtering the return values, to keep the system functional and to continue to operate undetected. This control flow is illustrated in Figure 2.4.

Examples for function pointers of interest include the interrupt descriptor table (IDT) and several machine specific registers (MSRs) within the CPU, both of which are specific to the x86 architecture. The system call table is an instance of an OS specific call table that has become famous among attackers, especially for creating rootkits [Blu09].

2.4.1.2 Direct Kernel Object Manipulation

Kernel objects hold other security relevant information beyond function pointers. File descriptors contain the access rights associated with an open file, process descriptors contain references to the associated users which determines their privileges, and so on. An attacker can change such values to his advantage, for example, to elevate his privileges. These types of attacks have become known as *direct kernel object manipulation* (DKOM) [But04].

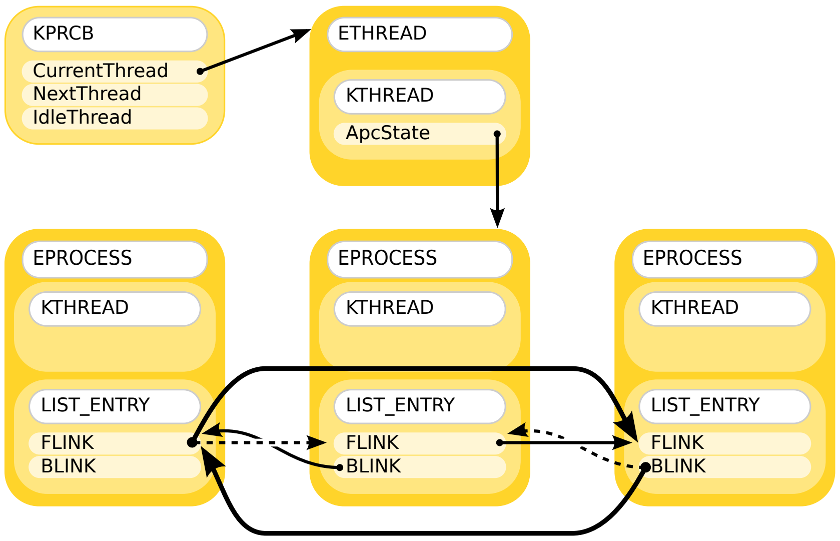


Figure 2.5: Process hiding for the Windows operating system using direct kernel object manipulation (DKOM). The process descriptor in the middle of the list is skipped by manipulating the '`FLINK`' field of the previous and the '`BLINK`' field of the next process in the list to remove the object from the list, thus hiding the corresponding process.

Aside from changing concrete values representing IDs, permissions and flags, an attacker can also tamper with pointers to other objects. An attack that manipulates the pointers to objects in a doubly-linked list was first described by Butler et al. [BUP03], even though he simply referred to it as "process hiding". The acronym "DKOM" was coined later by Butler himself [But04]. The attack works a follows: By "skipping" a particular process descriptor in the list of all processes on a Windows system, as depicted in Figure 2.5, the skipped process disappears from the output of many reporting tools while the process continues to execute normally. Even though the process is hidden from some reports, it may appear in other places where different pointers point to the skipped process descriptor, for example, when accessing the '`ETHREAD`' object and following field

'`ApcState`'. The same technique is also known to work for hiding drivers and network connections in Windows [HB06] as well as processes and kernel modules in the Linux kernel.

Note that in order to keep an object "active" and thus useful for the attacker, at least some references to this object have to persist. If *all* references to an object would be removed from the entire kernel state, the kernel itself would loose awareness of it. As a consequence, a process would no longer be scheduled, a network connection would not accept any more packets, and so on. Thus, an attacker has to be careful with the pointers he manipulates to keep his code functional.

2.4.2 Code-based Attacks

Once an attacker has gained full kernel privileges, he can not only manipulate data, but also the kernel code itself.[4] This opens almost unlimited possibilities for kernel manipulations. However, the most widely adopted use of code-based attacks is the *runtime* or *detour patching* of functions to insert malicious code at certain points of the existing control flow.

Even though the effect of this method is similar to the previously introduced function pointer hooking, runtime patching is more generic and more powerful. Both approaches allow an attacker to force the original code to take a detour and execute his code instead. With runtime patching, he replaces an existing instruction at a chosen code position with a jump instruction that will transfer control to his code. After the detour code has been executed, the control flow can be returned to the position where the initial jump instruction left of, depending on the intentions of the attack. To keep the functionality of the original code sound, the detour code executes the instructions that were replaced by the jump code before returning control back to the kernel code [HB06]. The control flow of this attack is shown in Figure 2.6.

Runtime patching is a versatile technique for multiple purposes. It can be employed to block function calls under certain conditions (e. g., to stop a malware removal tool), to trace particular calls and record their parameters, to manipulate or filter input parameters to a function as well as the corresponding output values, and even to replace entire routines with custom code [Blu09].

2.5 Rootkits

From the perspective of an attacker, gaining unauthorized access to a machine is a hard piece of work, especially via a remote channel. Once he has successfully acquired system privileges, he usually wants to sustain his presence on the compromised machine and make it easier for him to re-gain these privileges later on. However, this has to be done

[4]Even though the kernel can write-protect his code using the protection bits of the page tables, an attacker with full privileges can easily flip the protection bits to make the pages writable again.

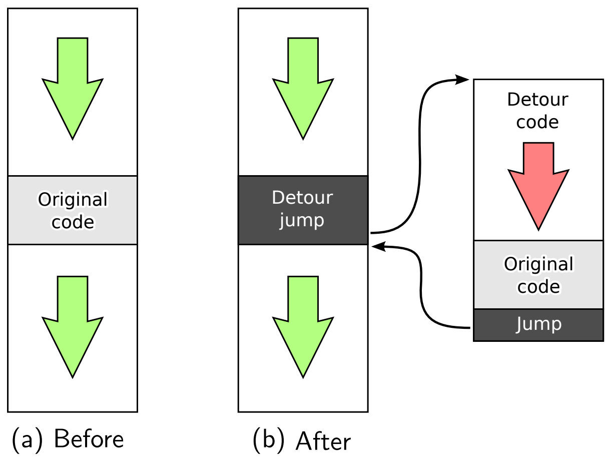


Figure 2.6: Control flow of a *runtime* or *detour patch*, (a) of the original code and (b) after the patch has been applied.

without being noticed by the legitimate owner or administrator, because otherwise the system might be cleaned and his efforts are lost. For this purpose, he typically installs a *rootkit* on the system.

A more accurate description of a rootkit is given by the following definition from Hoglund and Butler [HB06]:

> "A rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer."

This definition effectively restricts the functionality of a rootkit to providing stealth for some sort of additional payload. Blunden broadens this definition [Blu09] and characterizes rootkits as software that provides three services: concealment, command and control, and surveillance. Even though there exists legitimate software that provides parts of these services, for example, remote administration tools such as the OpenSSH daemon, it is the stealthiness that differentiates such tools from rootkits.

2.5.1 Rootkit Taxonomy

Many of the examples for kernel-level attacks given in Section 2.4 illustrate their applicability for rootkits, especially for providing concealment. In fact, many real-world rootkits that exist today employ such methods to provide their services. However, there are other ways for a program to operate undetected without attacking the kernel. For example, the first rootkits for Linux replaced important system programs that report on running processes (*ps*), files (*ls*), network connections (*netstat*), etc., with custom binaries that delivered fake reports. These replacements simply excluded a back door daemon that was installed as part of the rootkit from their reports.

To draw the connection between kernel-level attacks and rootkits, we use the malware taxonomy introduced by Rutkowska [Rut06]. This taxonomy classifies malware according to their interaction with the OS kernel. The first class is *type 0* malware which runs as a regular process within the operating system. It does not alter the kernel code or data. Consequently, the Linux rootkit consisting of replacement binaries and a back door is a type 0 rootkit. The next class is *type I* malware. A member of this class tampers with the kernel, but only modifies parts of the state that are considered to by constant during normal operation, such as code sections, the IDT or the system call table. For example, a rootkit that only uses runtime patching or MSR hooking is a type I rootkit. As the final class, *type II* malware operates on dynamic resources that are supposed to change during runtime. Rootkits that rely on DKOM are obviously representatives of the type II class.

2.5.2 Rootkits vs. Kernel-Level Attacks

In this thesis, we only concern ourselves with rootkits that involve kernel-level attacks. Borrowing the terminology from Rutkowska, we are only interested in type I and type II rootkits. Since these rootkits inevitably leave traces within the kernel state and VMI provides us with a complete and untainted view of that state, we always have a chance to detect them.

For simplicity, we use the term “rootkit” for kernel-level rootkits throughout this work, i. e., rootkits that implement kernel-level attacks, or in other words, rootkits of type I and type II.

2.6 Summary

This chapter has described the fundamental technologies that this work builds upon and established a common vocabulary that is used in the remainder of this thesis. When running an operating system inside of a virtual machine, we can make use of the hypervisor’s privileged position to inspect the state of the guest OS, a technique called virtual machine introspection (VMI). Using this untainted view of the guest’s state, VMI is very well suited to detect kernel-level attacks against the guest, especially rootkits that tamper with the kernel’s state. However, before we can perform malware detection on the hypervisor level, we first need to address the semantic gap issue, a common challenge of all VMI applications. In the chapter that follows, we introduce a formal model that can be used to describe different approaches for bridging the semantic gap and reason about their properties.

Chapter 3

Modeling VMI Approaches

In this chapter, we introduce a formal model which helps in examining and reasoning about possible VMI approaches and their resulting properties. We focus on leveraging VMI to perform intrusion detection on the OS level, that is, detection of kernel-level attacks that are executed with full system privileges, as described in Section 2.4.

The typical goal of an intruder performing such attacks is to alter the OS state in such a way that he may remotely access the system, abuse system resources, or collect intelligence from the system or its users. These types of attacks include—but are not restricted to—rootkits. User-level compromises that do not affect the OS state are out of the scope of our considerations. While we focus on intrusion detection, our arguments are applicable to further security applications of VMI as well, such as digital forensics, secure logging, etc.

Chapter overview. We start in Section 3.1 with a summary of the challenges that need to be addressed to perform virtual machine introspection. The basic components of our formal model are defined in Section 3.2. Based on our model, we identify three basic patterns for bridging the semantic gap in Section 3.3. Each of these patterns can be associated with certain properties. We argue that one is able to model any possible way of bridging the semantic gap with a combination of these three patterns. Finally, we summarize the essential concepts of our model, the patterns, and the model's application in Section 3.4.

This chapter is based on joint work with Jonas Pfoh and contains material previously published in: *A Formal Model for Virtual Machine Introspection* [PSE09].

3.1 VMI Challenges

Introspection is a synonym for self-examination. Virtual machine introspection describes the act of examining and monitoring the virtual machine (i. e., the state of the guest OS) from the vantage point of a hypervisor or a second privileged VM. As discussed in Section 2.2, the hypervisor naturally allows for this as all system state is inherently visible.

Recall that the properties of virtualization which support intrusion detection from the hypervisor level can be summarized as isolation, inspection, and interposition. These properties lead to the following advantages of VMI which are beneficial for many security applications:

- A *complete* and *untainted* view of the system state.
- A *consistent* view of the system state, since the VM can be paused.
- The ability to *manipulate* the VM state.
- Complete *isolation* of the VMI mechanism from the guest OS.

In order to leverage the full potential provided by these advantages, the following challenges must be overcome:

1. Interpreting the immense amount of binary low-level data that comprise the system state to obtain abstract high-level state information.
2. Identifying relevant parts of that state information for a given application.
3. Classifying both known and unknown system states in an appropriate way.

These challenges are common among all VMI approaches and can be addressed in different ways. The first challenge, which is equivalent to bridging the semantic gap, is particularly interesting for the following reason: In order to bridge the gap, one needs to apply knowledge about the virtual hardware architecture, the guest OS, or both (cf. Section 2.3). Depending on the way this knowledge is acquired and applied, the resulting approach will have characteristic properties. Consequently, it becomes beneficial to use a generic model that allows to describe and compare any VMI approach and reason about its characteristics.

3.2 VMI Model

In this section, we define a formal model for describing the inspection property of arbitrary VMI approaches in a generic way. This model complies with the challenges that we identified in the previous section. Each of these challenges can be addressed by a function that, based on the appropriate input, produces the correct output for a particular combination of virtual hardware and software architecture.

Before we introduce the functions themselves, we define the input and output sets that these functions operate on. More precisely, we introduce the notion of a *state*, a *view*, and a *profile*. The corresponding sets represent different levels of abstraction of the data accessible by the hypervisor and may include semantic knowledge about the introspected guest or a history of previous information. Finally, the resulting information can be evaluated and tagged with *class labels* to distinguish application specific classes of guest states, such as “compromised by a rootkit” and “not compromised”.

When all of these sets have been specified, we model their relationships with generic functions that each take at least one of the sets as input and output another. Every VMI application implements several or sometimes all of these functions in order to bridge the semantic gap. Since the actual implementation varies for each application, operating system, and hardware architecture, we do not detail how such functions can be realized in this chapter, but limit the description to the information flow in our model. (In Chapter 6, we present our own implementation for one of these functions.)

3.2.1 States, Views, and Classes

All VMI approaches inspect guest system states. A guest system state is comprised of all the guest system information that is either visible to the hypervisor or the guest OS, that is, CPU registers, volatile memory, stable storage, as well as the internal state of connected devices. For our model, we begin by defining two sets of states:

Definition 1 *We define $S_{int} \subseteq \{0,1\}^n$ as the set of all possible VM states visible to the guest OS being monitored, and we define $S_{ext} \subseteq \{0,1\}^m$ as the set of all possible VM states visible to the hypervisor. A single state is represented as a finite length string of bits, where n and m are the number of bits required to represent an entire state from the respective vantage point.*

In other words, a state is simply the concatenation of all visible values within the VM. Concretely, a state is the concatenation of all CPU register values, with the entire contents of memory, with the entire contents of stable storage, and so on.

One challenge of creating a VMI-based security mechanism is classifying any given machine state based on the specific application. For example, an intrusion detection system in the hypervisor might label a given guest state as “compromised” or “uncompromised”. This is formalized in the following definition:

Definition 2 *We define C as the set of all possible class labels for machine states for a specific application.*

Given a machine state $s \in S_k$, $k \in \{int, ext\}$, ideally one would like to have a function $f : S_k \to C$ that directly outputs the correct class label for s. However, this may not be possible in practice. First and foremost, the semantic gap challenge must be solved in order to extract meaningful information out of the binary state.

In addition, a system state contains lots of information that is irrelevant to the classification. It is advantageous to extract only the relevant parts of the state for a respective application. For example, modern OSs tend to reserve a large amount of memory to cache data from stable storage. This cache represents a portion of memory which has no valuable contribution for many security mechanisms, such as intrusion detection.[1]

Finally, some $s, s' \in S_k$ with $s \neq s'$ may represent the same abstract system state for a specific application, but differ in their binary representation due to the randomness in a running kernel, such as memory locations of dynamic data structures, inodes, process IDs, etc. For example, the order in which processes appear in the process list may not be important from a system security perspective. Consequently, all permutations of a process list would be considered the same.

In order to represent these factors in our model, we introduce the notion of a view:

Definition 3 *We define V_{int} and V_{ext} as the set of all possible views constructible from S_{int} and S_{ext}, respectively. A view is a concise representation of the partial or entire state and can be constructed by any computational means.*

Ideally, a view contains only relevant data and is completely independent of any kernel randomness, such as variable memory locations, effective process IDs etc. In other words, a view is the "essence" of a system state. In order for this "essence" to be useful, the semantics of every single bit included in the view must be well-known. When we say that the semantics of a bit is known, we mean that its meaning within the context of the guest system is known. For example, the bits x to y within the view represent the list of running processes in the form for n consecutive process descriptors.

To give the reader a better understanding of the view generation process, suppose a VMI application requires to know the number of processes of the user "root". To generate this view, we can locate the kernel data structure that manages the processes in kernel memory, extract the list of process descriptors, count the number of processes belonging to "root", and finally return the total number as the resulting view. This illustrates that a view may be as simple as a single numeric value.

Now that we have defined the input and output sets that VMI approaches operate on when bridging the semantic gap, we are ready to introduce the functions that, in combination, classify a system state according to a given application.

3.2.2 View Generation, Aggregation, and Classification

Each operation presented in Definition 3 serves as a means for bridging the semantic gap. Consequently, they all must incorporate knowledge of the virtual hardware architecture and/or the guest system software architecture (i. e., the combined architecture of the guest OS and all software running on the guest OS). This knowledge of the respective architectures is denoted by the following:

[1]In fact, there do exist viruses that infect disk caches, e. g., Darth_Vader or W95/Repus, but due to the inefficiency of their replication method, such viruses are not prevalent in the wild [Szo05].

Definition 4 *We define a virtual hardware architecture description λ and a guest system software architecture description μ as:*

$$\begin{aligned} \lambda &\in \{\langle HW \rangle \mid HW \textit{ is a hardware architecture}\} \\ \mu &\in \{\langle SW \rangle \mid SW \textit{ is a software architecture}\} \end{aligned}$$

At this point it is possible to define a model that is capable of representing the transformation of a single state to a view and a view to a class label. The problem with such a model is that it is not able to describe an approach that considers several sequential system states or the change from one state to another in order to classify it. However, this is an important aspect for many scenarios. Considering the case of a VMI-based intrusion detection system (IDS) for illustration, a sequence of small state changes spread over multiple views might not be considered harmful by themselves, but may be malicious in this sequence.

The ability to model a stateful system also allows the measurement of certain activities against a threshold. For example, a compromised process may sequentially open a large number of sockets. A single view may seem benign since the process only opens a single socket at a time. However, it may become clear that malicious activity is transpiring when looking at a sequence of views. Thus, we introduce the notion of aggregating past views into a profile:

Definition 5 *We define P as the set of all possible profiles. A profile is an aggregation of several consecutive views of a system run.*

With the definition of a profile to model stateful introspection tasks, we are prepared to specify a simple, yet universal formal model for virtual machine introspection:

Definition 6 *For a virtual hardware architecture description λ and a guest system software architecture description μ, we define the following functions:*

$$\begin{aligned} f_{\lambda,\mu} &: S_{int} \rightarrow V_{int} && \textit{(Internal view generation)} \\ g_{\lambda,\mu} &: S_{ext} \rightarrow V_{ext} && \textit{(External view generation)} \\ a &: V_{int} \times V_{ext} \times P \rightarrow P && \textit{(Aggregation)} \\ d &: P \rightarrow C && \textit{(Classification)} \end{aligned}$$

These functions in combination are capable of modeling the information flow in any given VMI application.

The process of generating a view is essentially the act of bridging the semantic gap. In other words, the functions $f_{\lambda,\mu}$ and $g_{\lambda,\mu}$ can be seen as “semantic bridges” to close this gap. We refer to these view-generating functions as internal and external as well as use the subscripts *int* and *ext* for the sets of states and views. We may also refer to a process as happening internally or externally. These terms reflect the view point of the VM being monitored. That is, "internal" refers to a function or process that occurs within the monitored VM, while "external" refers to a function or process that occurs within the hypervisor or a privileged VM.

The flow of information in the model is depicted in Figure 3.1. The view-generating function $f_{\lambda,\mu}$ constructs a view internally, while $g_{\lambda,\mu}$ constructs a view externally. The aggregation function a takes these views along with the current profile to create a new profile of the system state. This profile is, in turn, processed by the classification function d to classify the current system state. Both the aggregation function and the classification function are executed externally.

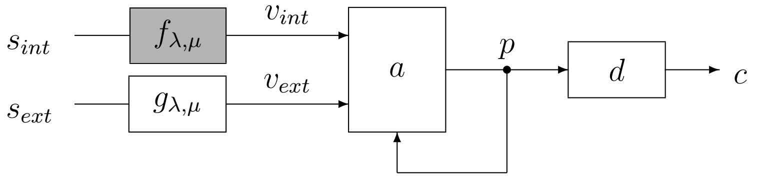


Figure 3.1: Visualization of the information flow in our VMI model, where $s_{int} \in S_{int}$, $s_{ext} \in S_{ext}$, $v_{int} \in V_{int}$, $v_{ext} \in V_{ext}$, $p \in P$, $c \in C$. The function $f_{\lambda,\mu}$ in the gray box is the only function that is evaluated from within the monitored guest.

The functions in Definition 6 map sets to sets. In order to distinguish the functions from a concrete implementation of such a function, we use the term *component*, denoted as $[\![f]\!]$, when we talk about a computer program that implements a function f. Consequently, we call implementations of the view-generating functions f and g *view-generating components* (VGCs). In contrast to a function, a VGC is constrained by the machine it is executing on with regard to processing power, memory, numeric precision, and so on.

The subscripts that the view-generating function labels carry is a pair λ, μ, which denotes that each function constructs its output based on the information contained in μ and λ. For a particular VMI approach, it is possible that either $\mu = \varepsilon$ or $\lambda = \varepsilon$, where ε is the empty word. In such a case, the approach makes use of a view-generating function that constructs its output based solely on the knowledge of either μ or λ. Going forth, rather than explicitly stating that either $\mu = \varepsilon$ or $\lambda = \varepsilon$, we simply leave the appropriate subscript out of the notation. For example, we denote the function $f_{\lambda,\mu}$, where $\lambda = \varepsilon$ as f_μ.

Additionally, it is possible that for a particular VMI approach that either $\forall s_{int} \in S_{int} : f_{\lambda,\mu}(s_{int}) = \varepsilon$ or $\forall s_{ext} \in S_{ext} : g_{\lambda,\mu}(s_{ext}) = \varepsilon$. In a practical sense, this represents the lack of a corresponding component in the implementation. For example, a VMI approach may only contain a single VGC within the hypervisor. In order to denote this, we simply omit the appropriate view-generating function as well as its input into the aggregation function when describing such an approach.

3.2.3 Purpose

This model breaks a VMI system into its basic components and their interactions, thus reflecting important design decisions. This is very helpful for understanding and reasoning about a VMI system design. As we will see, there are different ways to reach the same goal in a VMI system which result in different security, safety, and portability properties. Taking the time to describe a system with our model forces one to look at all aspects of the system design. In doing so, one may find a flaw in parts of the system that were not the focus of the initial design.

Our model reveals properties for individual view-generating functions in a fine-grained manner. These properties are explained in the next section. The utility of our model becomes especially evident in cases where two functions result in contradicting properties, for example, a hardware portable function in combination with a function which is not hardware portable. Considering how these contradicting properties interact and coming up with the ensuing property of the system as a whole takes some thought. This thought process is easily overlooked or taken for granted in some circumstances. A further discussion on this issue can be found in Section 3.3.5. If a design does not conform to the properties it was set out to achieve, it may be tweaked and re-checked.

We have found this model to be a very helpful tool for our research and continue to use it in the remainder of this thesis.

3.3 View Generation Patterns

From the vantage point of the hypervisor, the guest system state is seen in a strictly binary form. Creating a view in this situation becomes quite challenging due to the fact that the external view-generating function $g_{\lambda,\mu}$ has no native knowledge about what this binary data represents, for example, which data structures lie where in memory.

As stated before, view generation is a matter of bridging the semantic gap. That is, discussing the different methods of generating views is, in fact, a discussion of ways to bridge the semantic gap. This is an excellent example of how our model aids in the discussion of a particular aspect of VMI.

In order for the hypervisor-based view-generating function $g_{\lambda,\mu}$ to create an appropriate view where the semantics of every bit is known, the function needs to make use of knowledge about either the guest system software architecture μ, the virtual hardware architecture λ, or a combination of the two. We refer to this knowledge as *semantic knowledge* because it gives the function knowledge about the meaning of the observed system state for the guest.

Using our model, we present three *view generation patterns* that may be used alone or in any combination as a means for bridging the semantic gap, namely:

- the *out-of-band delivery* pattern,
- the *in-band delivery* pattern, and

- the *derivative* pattern.

They differ in two ways, first, where within the architecture the view generation takes place (i. e., internally or externally) and, second, how semantic knowledge is incorporated. The patterns can then be used alone or in combination to describe *all* possible methods for bridging the semantic gap. What follows is an in-depth discussion of each pattern along with its corresponding properties.

3.3.1 Properties

Before we start with the description of the view-generating patterns, we first introduce the properties that may result from their application.

3.3.1.1 Hardware Portability

A view-generating function that is *hardware portable* can be used across multiple hardware platforms without modification. In general, this property is only achieved if the function does not require any knowledge of the hardware architecture λ, but is solely based on the software architecture μ. Such an approach saves the user from the effort of having to re-implement the same functionality for each hardware architecture.

3.3.1.2 Guest Operating System Portability

Similar to hardware portability, a view-generating function is said to be *guest OS portable* if it supports multiple guest operating systems without change. Consequently, a function that does not rely on any semantic knowledge about the software μ but only on hardware knowledge λ is guaranteed to be portable among all OSs that run on that hardware platform in compatible configurations. Portability does not come with any security relevant advantages per se, but it allows the same function to be easily reused among several OSs.

3.3.1.3 Binding Knowledge

The type of knowledge a view-generating function is based upon determines how well this knowledge is dovetailed with the guest. Needless to say, a function $g_{\lambda,\mu}$ that is designed to apply semantic knowledge of the software architecture μ and hardware architecture λ will most likely produce wrong results for some different architecture μ' and λ'.

If a malicious entity changes the software architecture, for instance, by inserting some hidden data structure, it effectively changes the appropriate software description from μ to μ'. In that case $g_{\lambda,\mu}$ is no longer fit to provide an accurate view. Bahram et al. have demonstrated that such an attack is feasible in practice [BJW$^+$10]. This attack is possible because the semantic knowledge that the view-generating function uses is in no way bound to the actual software architecture of the running guest. For this reason, such an approach is called *non-binding* [LLCL08].

In contrast, consider a function g_λ that relies on the hardware architecture alone. The hardware architecture description λ is bound to the hardware architecture of the virtual machine because there is no way for a malicious entity that compromises the guest OS to change the virtual hardware architecture. A malicious entity has no way of changing λ, and whatever changes it may apply to μ will not effect the ability of g_λ to create the correct view. Figure 3.2 illustrates this relation. We call such an approach *binding*.

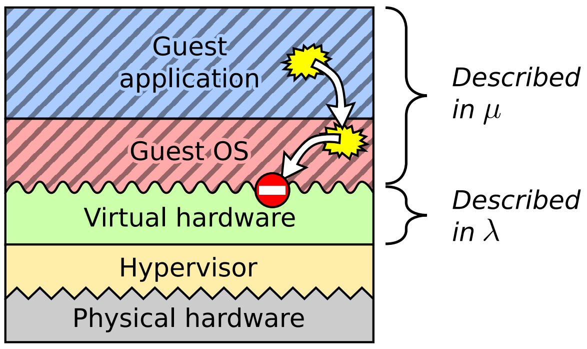


Figure 3.2: Even an attacker who has gained full control over a virtual machine (the hatched area) cannot modify the virtual hardware architecture described in λ to evade view generation. In addition, the strong isolation of the hypervisor prevents him to break out of the virtual machine.

3.3.1.4 Isolation

The process of generating a view can be executed in one of two possible locations: either within the guest that is to be monitored or outside of the guest. We say the view generation process is *isolated* when it is running outside of the guest. This does not imply that the function has to be implemented as part of the hypervisor itself. It only means that the function is executed externally from the vantage point of the guest. Such functions can just as well be carried out within a separate, privileged VM monitoring the target VM without affecting our discussion.

The isolation property leads two to beneficial consequences. First, even if the kernel of the monitored guest has been compromised, the view generation is not directly affected since it runs independent of the guest, as shown in Figure 3.2. This is a major advantage for security applications such as malware analysis or intrusion detection. Second, the view can be generated without having any undesired side effects for the guest. For example, a VMI-based tool for digital forensics that features the isolation property will have no observer effect on the guest and thus will not distort any possible evidence.

3.3.1.5 Offline Analysis

For some of the patterns we are going to describe, the VM is not required to be running while the view is generated. In those cases, the pattern supports *offline analysis*. It allows the view generation to be performed either while the VM is temporary suspended or based on snapshots that contain the entire state of the guest and are stored as files on the hard disk. Note that the offline analysis property always implies that the view generation process is isolated from the guest and vice versa; both properties are mutual dependent.

If the view is generated while the VM is not running, the observed state will be in itself consistent. It cannot occur that one part of the state changes while another part is still examined. A second advantage stems from the fact that this property allows one to mitigate external effects of keeping a compromised system running. For example, it is favorable to suspend a system which acts hostile to neighbors on a production network while still being able to analyze it.

3.3.1.6 Full State Visibility and Applicability

A view-generating function that is executed in isolation of the guest possesses *full state visibility*. That is, every bit that makes up the guest system state is visible to such a function, even areas which are not accessible from within the VM itself. With this complete and untainted access to the state, the function has the potential to observe anything that happens within the guest, regardless at which layer of the hardware/software stack. This is a very strong and encouraging argument from a security perspective.

However, the ability to access the complete state is different from being able to interpret that information. The later also requires a complete understanding of the hardware and software architecture. To distinguish between these two qualities, we say that a function has *full state applicability* if the entire state is not only visible, but in addition the function potentially has the semantic knowledge to analyze it. In other words, full state applicability requires access to the full state as well as both λ and μ.

3.3.2 Out-of-Band Delivery

Now that we have described the properties that characterize the different view generation patterns, we present the first pattern, which is the out-of-band delivery pattern. This is the most commonly used method for bridging the semantic gap in the literature (cf. Section 4.1). Here semantic knowledge is *delivered* to the external view-generating function in an *out-of-band* manner. That is, the external view-generating function receives the semantic knowledge in advance, before VMI begins. For example, the hypervisor may make use of a previously delivered symbol table based on the guest OS kernel to determine the position of key data structures, which is illustrated in Figure 3.3.

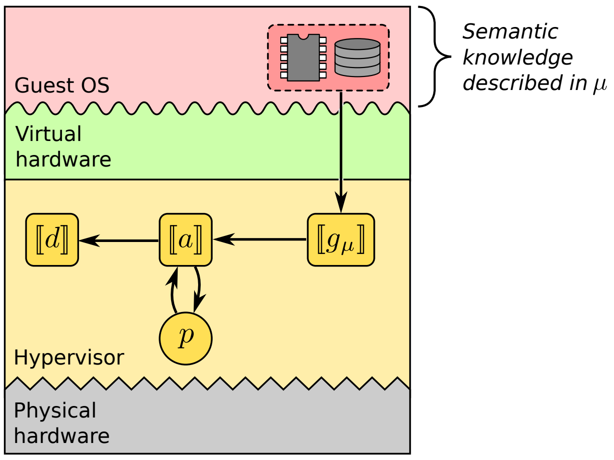


Figure 3.3: Out-of-band delivery: The semantic knowledge about the guest software architecture μ is delivered out-of-band to the view-generating component $[\![g_\mu]\!]$.

Formally, this pattern is described as follows:

$$\begin{aligned} g_\mu &: S_{ext} \to V_{ext} \\ a &: V_{ext} \times P \to P \\ d &: P \to C \end{aligned}$$

From this description, it can be seen that the view-generating function makes sole use of knowledge of the software architecture μ. Since this knowledge is built into the function, this function is only fit to create a view based on a software architecture for which μ is the description. Consequently, this approach is non-binding.

As g_μ depends entirely on μ, it lacks guest portability. That is, if the guest system software is changed or replaced completely, the software architecture description must also be changed to reflect this and the view-generating function must be constructed anew to accommodate the new system software. However, it does not require λ, which results in a hardware portable approach.

As seen in the formal description, this pattern makes use of the external view-generating function, g_μ. The fact that the view generation is performed externally leads to three additional properties, which are isolation, support for offline analysis, and full state visibility. Unfortunately, even though the state is fully visible, it is not completely applicable because g_μ misses the semantic knowledge λ to interpret the hardware state.

3.3.3 In-Band Delivery

The in-band delivery pattern describes an approach in which an internal component creates a view and delivers this view to the hypervisor. Since the view-generating function is internal, it may make use of the guest OS's inherent knowledge of the software

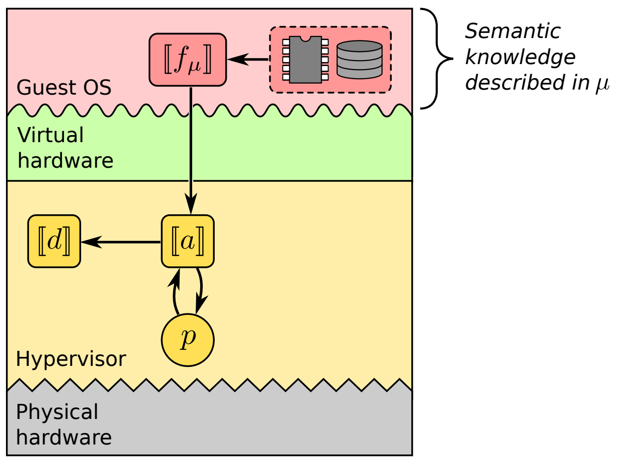


Figure 3.4: In-band delivery: The internal view-generating component $[\![f_\mu]\!]$ uses the inherent semantic knowledge μ of the guest OS and delivers the view in-band to the aggregation component $[\![a]\!]$.

architecture, as shown in Figure 3.4. That is, semantic knowledge is being *delivered* in an *in-band* manner.

Upon closer inspection, this pattern does not so much bridge the semantic gap, but rather avoids it. In fact, one could rightfully argue that this is not a method for bridging the semantic gap in the purest sense since the view is generated from within the monitored guest OS. As such, there is actually no gap that would require bridging. However, we include this pattern for the sake of completeness and because it can be successfully combined with other patterns.

This method is formally described as follows:

$$\begin{aligned} f_\mu &: S_{int} \to V_{int} \\ a &: V_{int} \times P \to P \\ d &: P \to C \end{aligned}$$

Having an internal view-generating function results in a few disadvantages. First, it is not isolated from a potentially hostile environment since view generation takes place within the monitored guest OS. As a consequence, the component that implements view generation is susceptible to compromise from any malicious entity that has compromised the monitored guest OS. In essence, it is necessary to trust a component which may be not be trustworthy. Second, an internal function does not support the examination of a suspended system, creating possible inconsistencies and resulting in the inability to mitigate external effects. Third, an internal function will always interfere with the system being monitored and finally, there may be parts of the state not visible to the view-generating function. As with the out-of-band delivery pattern, this pattern is non-binding. In-band delivery also shares the lack of guest portability with the out-of-band delivery pattern.

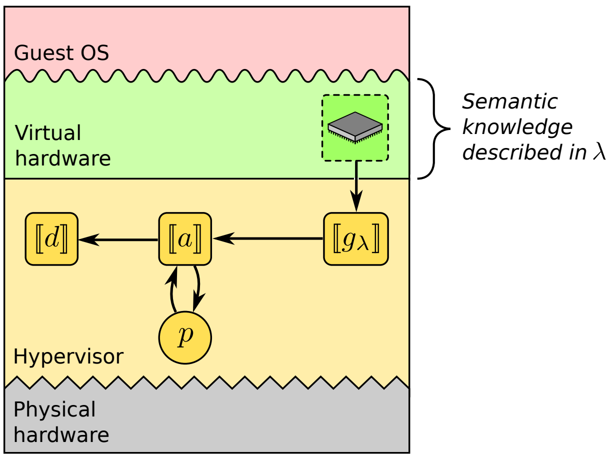


Figure 3.5: Derivation: The external view-generating component $[\![g_\lambda]\!]$ derives the view based on the semantic knowledge λ of virtual hardware architecture.

It seems appropriate to have a short discussion about why the in-band delivery pattern is not being considered with a view-generating function that makes use of hardware knowledge (i. e., $f_{\lambda,\mu}$, $\lambda \neq \emptyset$). While we admit it is practically possible to create such a view-generating function within the monitored guest OS, view generation using knowledge of the hardware architecture is *always* best done externally for one primary reason: Any view-generating function that makes use of hardware architectural knowledge can be implemented from within the hypervisor and gains all of the benefits associated with such an approach (i. e., isolation, offline analysis, etc.). On the other hand, implementing a view-generating function in the guest OS brings no advantages over an approach in which the function is implemented externally. For this reason, we argue that while implementing such a function internally is practically possible, there is absolutely no added value and is therefore not discussed further.

3.3.4 Derivation

The final VMI pattern moves away from the more common approaches and has the hypervisor *derive* information through semantic knowledge of the hardware architecture. For example, understanding a particular architecture and monitoring control registers within a CPU provides us with some semantic knowledge. Figure 3.5 depicts the application of this pattern.

Formally, this approach is described as follows:

$$\begin{aligned} g_\lambda &: S_{ext} \to V_{ext} \\ a &: V_{ext} \times P \to P \\ d &: P \to C \end{aligned}$$

From our formal description, it is seen that this pattern makes use of the external view-generating function g_λ, thus leading to the same advantages that are discussed in Section 3.3.1 with respect to the externality.

Further, it can be seen that g_λ constructs its output based on knowledge of the hardware architecture alone. This results in two additional advantages. First, this pattern is completely guest OS portable as the view-generating function makes no use of the guest software architecture. Second, this is a binding approach in contrast to the two delivery methods. That is, the hardware architecture description is bound to the hardware architecture of the virtual machine because even an attacker with full system privileges has to play according to the rules of the hardware. Consequently, the view generation process of g_λ is unaffected by any doings of a malicious entity.

This seems like the ideal pattern, though in practice there is a large drawback to such an approach: The view that is generated is generally very constrained. A lot of information that is present in the state is lost through the view-generating function. This is due to the fact that there is information that cannot be extracted by only making use of hardware architectural knowledge and that information which can be extracted often needs much expertise and effort to be extracted.

Finally, this approach is obviously not portable across various hardware platforms. If the view-generating function is built to interpret the state of a particular hardware architecture, then this function will clearly not be suited for use with another hardware architecture.

3.3.5 Combination of Patterns

The patterns that have been discussed up to this point are in their purest form. It is, of course, possible to combine such patterns. In fact, it is sometimes beneficial to use a drawback of one pattern in combination with another pattern to one's advantage: A component of rootkit detection often involves trying to determine whether two mechanisms, one usually at a higher level than the other, report the same status about some OS data structure. Such an approach is possible by combining the in-band and out-of-band delivery patterns and relying on the fact that a capable malicious entity will compromise the internal view-generating function or the guest OS itself, thus leading to an inconsistency between the two views created, which is then detected. Such an approach can also be described with our model:

$$\begin{aligned} f_\mu &: S_{int} \rightarrow V_{int} \\ g_\mu &: S_{ext} \rightarrow V_{ext} \\ a &: V_{int} \times V_{ext} \rightarrow P \\ d &: P \rightarrow C \end{aligned}$$

Using this formal description, it is clear which view-generating functions are used and that this is a combination of the in-band and out-of-band delivery patterns. Note that

Property	*Delivery*		*Derivation*
	in-band	*out-of-band*	
Hardware portable	✓	✓	—
Guest OS portable	—	—	✓
Binding knowledge	—	—	✓
Isolated from guest	—	✓	✓
Offline analysis	—	✓	✓
Full state visible/applicable	—/—	✓/—	✓/—

Table 3.1: Comparison of the properties of the different view generation patterns.

in this case, the aggregation function a does not depend on a previous profile, thus P is not used as an input to a. This information along with a thorough understanding of the the properties associated with the three patterns allow for a more structured analysis of a particular approach.

We have summarized the properties of the different patterns in Table 3.1. When combining such patterns, it is important to carefully consider how the properties of these patterns will interact with each other. For example, an external view-generating function $g_{\lambda,\mu}$ inherits the full state visibility property from the out-of-band delivery and the derivation pattern. In addition, it becomes full state applicable because it incorporates both hardware and software knowledge. This property cannot be achieved with any of the three patterns alone.

However, deciding what approach is best for a given application is not as simple as combining all patterns in order to get the benefits of each. An advantage of one pattern will not necessarily negate the disadvantage of another. For example, a combination of a delivery pattern with the derivation pattern will not result in an approach that is guest portable simply because one of the patterns has such a property. In fact, in this case, the opposite is true: The disadvantage of one pattern negates the advantage of another and the resulting component is neither hardware nor guest OS portable.

There are additional cases in which the properties are independent, for example, a combination of a delivery and a derivation pattern, though this time with respect to the binding property. A combination of two such patterns can neither be classified as binding nor as non-binding. There is simply some information that is the result of bound knowledge and some information that is the result of unbound knowledge. How this information is to be used needs to be carefully considered.

Finally, some patterns are simply not appropriate for a given application. For example, using the derivation method alone to monitor changes in a file on a disk is a poor choice as it is impossible—without knowledge of the file system structure in use—to derive the exact sector of a disk on which the appropriate portion of a file lies.

3.4 Summary

We have presented a universal formal model for virtual machine introspection. It is universal in the sense that it can be applied to all security applications of VMI we are aware of. This model has shown its usefulness throughout our research. To demonstrate this usefulness, we have classified all possible view-generating approaches into three patterns, namely in-band and out-of-band delivery, and derivation. Furthermore, we presented a thorough discussion of each pattern's properties as well as their advantages and disadvantages for view generation.

In addition, we identified and discussed the three major challenges of VMI-based security applications, namely, interpreting the state information, identifying relevant state information, and classification of states. We have found our model to be an invaluable tool to discuss and to better understand all the intricacies of building VMI-based security applications. The model and the associated patterns are able to represent all VMI approaches in a consistent way, making them comparable and allowing one to reason about their properties and appropriateness for a given application.

Chapter 4

Related Work

Since the first computer viruses started to spread at the beginning of the 80s [Szo05, Kas08], computer systems have been integrating dedicated security mechanisms in hardware or software to counter the threat of malicious code or mitigate the possible impact. Hence, it is no surprise that a lot of different approaches have been proposed over time to protect computer systems. Due to the increasing popularity and availability of virtualization in desktop and server environments [Ros04], researchers start to re-think established, host-based mechanisms and combine them with virtualization techniques to strengthen their resilience and effectiveness.

Virtualization provides interesting properties for implementing security applications on the hypervisor level, as already discussed in Section 2.2. Garfinkel and Rosenblum have pioneered this idea and coined the term virtual machine introspection [GR03]. Their work has inspired many other researchers and lead to a whole new area of research.

The formal model for virtual machine introspection which was subject of Chapter 3 is a helpful means to describe and compare the proposed mechanisms in this field. Since a substantial part of our contribution in this thesis is about bridging the semantic gap, we classify the following related work according to their way of overcoming the semantic gap challenge when creating the guest view. In addition, we detail the level of automation and sophistication the proposed view generation provides. Note that many approaches use hybrid techniques for view generation, that is, a combination of patterns. In such cases, the classification is based on the view generation pattern that contributes most to the proposed work.

Later in this thesis, we cover a topic that might seem rather unrelated to the subject of this thesis at first glance: static pointer analysis of source code in the C programming language. This topic also has been the focus of extensive studies. However, we postpone to look into related work in this field until Section 7.5 where our motivations and goals for this kind of analysis should be more clear to the reader.

Chapter overview. We begin with a description of related delivery-based approaches in Section 4.1. Since these approaches provide a broad spectrum of view generation techniques, we sub-divide them further with respect to the level of sophistication of the employed view-generating component. In Section 4.2, we examine some work that uses the derivative pattern for view generation. This includes two of our own publications about hardware-based derivation of state information which are not part of this thesis. Finally, we summarize this chapter and highlight the key differences between previous work and our own research in Section 4.3.

4.1 Delivery-based Approaches

Delivery-based approaches are the most commonly used way for bridging the semantic gap. Recall that the delivery-based VMI approaches deliver the required semantic knowledge either in-band, that is, at runtime, or out-of-band, when the introspection starts (see Section 3.3.2 and 3.3.3, respectively).

We partition the related work in this field into two sub-categories:

- The first category contains approaches that bridge the semantic gap in a way that is highly tailored towards the proposed application. Typically, the view generation process is supported by manual annotations that require in-depth domain knowledge of the observed operating system. While such restricted view generation may suffice for simple examples, it often cannot be generalized for other applications and is very sensitive to changes of any parts of the hardware and software stack of the guest.

- The second category consists of approaches that strive toward capturing a more complete view of the guest kernel state in a generic way. This process requires much more comprehensive semantic knowledge than the approaches in the first category do, for example, in form of the kernel's debugging symbols or source code. Such a complete view has two advantages: First, it allows for much more detailed analysis of a guest's state, and second, the view is potentially more universal and can be used for various kinds of analysis.

In Chapter 6, we introduce our framework for VMI applications, called InSight. It uses a combined delivery-based and derivate approach to bridge the semantic gap. InSight identifies kernel objects in memory and exposes them through several interfaces to applications building upon the framework. Thus, it falls into the second category of the two previously mentioned. In the following, we compare the methods used in our framework to other approaches where applicable.

4.1.1 Approaches using Restricted Semantic View Generation

The first VMI-based IDS was presented by Garfinkel and Rosenblum [GR03]. In their system, called Livewire, the IDS runs in a separate VM and monitors another VM for possible intrusions. The security mechanisms are enforced by a policy engine that is able to handle various kinds of policies. An overview of Livewire's system architecture is shown in Figure 4.1.

The proposed policy modules range from a file integrity checker over a memory integrity checker for critical data structures up to a raw socket detector. One of the modules performs *lie detection*, also known as *cross-view validation* [JADAD08], by comparing an inside view created by an agent running within the guest OS to an outside view generated at the hypervisor level. This module is based on the observation that malware often tries to hide its presence on a compromised system, for example, by hooking critical OS functions and reporting forged information about the system state. Consequently, if the inner view differs from the one created outside, something within the guest lies about the OS's state.

Livewire bridges the semantic gap in several ways. On one hand, it collects required information by reading kernel objects from the guest physical memory using the "crash" utility [Mis]. Crash itself uses the kernel debugging information to locate kernel symbols, which clearly is an out-of-band delivery approach. However, in order to read the symbols' values from guest physical memory, it also requires at least some semantic hardware knowledge, more precisely, an understanding of the virtual-to-physical address translation performed by the virtual hardware. Hence, the crash tool represents a combined

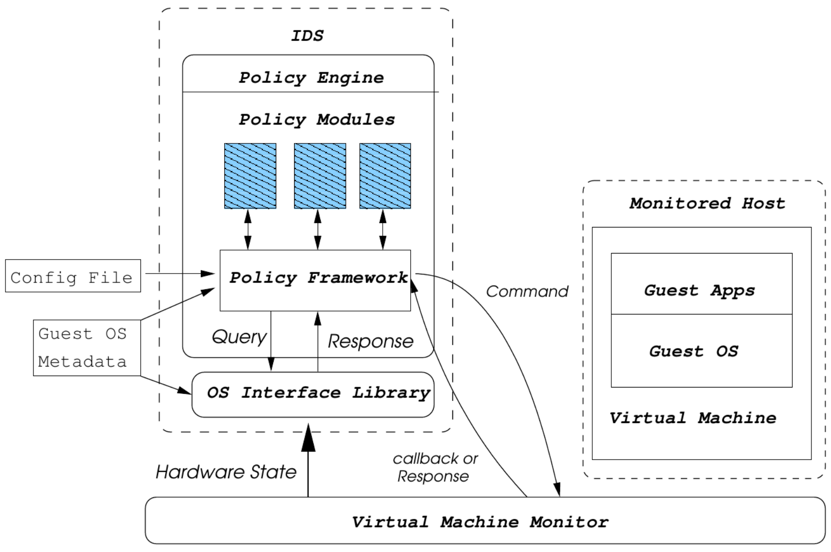


Figure 4.1: Overview of the system architecture of Garfinkel's and Rosenblum's VMI-based IDS called "Livewire" (taken from [GR03]).

approach for external view generation, written as $g_{\lambda,\mu}$. It is part of the "OS interface library" in the architectural overview in Figure 4.1.

The agent running inside of the guest OS, on the other hand, actually does not need to bridge the semantic gap as it is using the inherent guest knowledge to deliver the desired information in an in-band fashion to the hypervisor. Also, it does not depend on any hardware knowledge since it relies completely on guest OS functions. In our formal model, the agent is expressed as internal view generation function f_μ (not depicted in the architecture).

In summary, Livewire implements a combination of all three patterns for view generation: in-band and out-of-band delivery as well as derivation. The aggregated profiles do not depend on previous profiles, but only on the internal and external views V_{int} and V_{ext}. Using the notation of our model, this approach can be expressed as follows:

$$\begin{aligned} f_\mu &: S_{int} \to V_{int} \\ g_{\lambda,\mu} &: S_{ext} \to V_{ext} \\ a &: V_{int} \times V_{ext} \to P \\ d &: P \to C \end{aligned}$$

Almost all policy modules of Livewire that monitor critical parts of the system state and verify their integrity rely solely on $g_{\lambda,\mu}$ in the form of the OS interface library. Basing in IDS on external view generation is an excellent choice, since it provides isolation and support for offline analysis, along with all advantages that were covered in Section 3.3.1.4 and 3.3.1.5. Through the combination of both hardware and software knowledge in $g_{\lambda,\mu}$, this component not only has access to the entire state, but also has a chance to fully apply it (see Section 3.3.1.3). The downside of this combination is that their system is neither hardware nor guest OS portable. The internal view generation f_μ is only used for implementing the lie detection module. This is a clever way of using a drawback of the in-band delivery approach, namely the lack of isolation, to one's benefit: Any inconsistency between the internal and external view is an indication for possibly malicious activities. However, since both view generation functions mostly depend on delivered knowledge, they are non-binding and susceptible to attacks that render μ incomplete [BJW+10].

We have seen much work that follows the ideas of Livewire in one way or the other to implement security mechanisms on the hypervisor level. Among these approaches are various intrusion detection systems and system monitors [JWX07, PCL07, PCSL08, SSG08, XJZ+10, SWJX11], intrusion prevention systems [JKDC05, SLQP07, RJX08, RRXJ09, WJCN09], tools for forensic or malware analysis [DKC+02, HN08, MFPC10, RX10, DGPL11], an application-aware firewall [SG08], and a kernel-level debugger in the hypervisor [FPMM10].

All of these approaches have in common that they use delivery-based methods for view generation. However, the general tendency is to bridge the semantic gap just enough for one particular application, often with support of manually added expert knowledge. For example, a common introspection task for a Linux guest is to list the running processes. Many approaches implement this task by first retrieving the virtual address of the '`init_task`' variable—which represents the head to the process list—from the '*System.map*' file (out-of-band delivered knowledge) and then accessing the relevant fields within this data structure with hard-coded offsets (expert knowledge). While this leads to a straightforward implementation, it is very sensitive to changes to the kernel configuration, the source code, or the hardware architecture. In addition, all important data structures and fields have to be known in advance in order to extract their data. Obviously, this technique does not scale well for a greater number of data structures, kernel versions, configurations, and architectures.

In contrast to the work mentioned so far, our own view-generating component, InSight, strives towards full state applicability, that is, capturing the entire kernel state with the highest possible degree of automation (cf. Section 3.3.1). By using the debugging symbols and the source code of the guest OS kernel, InSight extracts the required semantic information independently and thus easily adapts to different kernel versions and configurations on the x86 architecture. Our framework is not designed for one particular use-case, but as a framework that supports arbitrary VMI applications and provides several interfaces for their implementation. As a consequence, the view that is generated by InSight is much more sophisticated and elaborated than the views of any of the previously listed approaches.

One approach that sticks out of the masses is Virtuoso which was developed by Dolan-Gavitt et al. [DGLZ+11]. It automates the generation of an entire view-generating component by tracing the execution of a given in-guest program and learning how to retrieve the same information from the hypervisor level. For the learning phase, multiple traces have to be generated and analyzed to extract the correct set of instructions for one particular application.

In terms of our model, Virtuoso is capable of transforming an in-band delivery into an out-of-band delivery approach using derived knowledge. The resulting external view-generating component inherits most of the beneficial properties from out-of-band delivery and automatically bridges the semantic gap by learning from in-band knowledge. However, as the external component is generated from an in-band delivery approach that lacks full state visibility, it is arguable whether the external component itself has the potential to access the full state in practice.

While the proposed technique provides a high degree of automation even across multiple OSs, the training programs for which Virtuoso was demonstrated to work were all very simple in nature and small in code size. Given the limitations of this approach that the authors have identified themselves, it is clear that automating the generation of a view of the guest kernel that is as complete as the one created by InSight is well beyond Virtuoso's scope.

4.1.2 Approaches using Kernel Object Mapping

The following approaches aim to generate a complete view of the kernel state using out-of-band delivered knowledge for the purpose of detecting kernel-level attacks. They monitor the integrity of different parts of the kernel's state that has the potential of being abused by an attacker. The view-generating components of these approaches share some similarities with InSight.

Petroni et al. propose a system that checks the integrity of kernel objects based on user-specified policies [PFWA06]. Their system is able to detect inconsistencies resulting from direct object manipulations performed by real-world rootkits. Even though their system is not based on virtualization but is implemented as a separate add-in PCI card [PFMA04], it shares all properties of a VMI-based approach with minor limitations.

Petroni furthers this work together with Hicks and verifies the integrity of the kernel-level control flow [PH07]. With semantic information gathered from a source code analysis of the Linux 2.4 kernel, their implementation constructs and traverses the kernel object graph and checks the integrity of all identified function pointers. However, their code analysis requires manual annotations of the source code in order to cope with type casts and generic pointers. Thus, it does not achieve the high degree of automation that InSight provides.

Other researchers suggest to automatically derive invariants on kernel data structures and assure that these invariants hold during runtime. Baliga et al. introduce Gibraltar [BGI08], a system that dynamically infers invariants of variables and data structures. In a training phase, it learns the typical usage of these objects, while it checks that these invariants hold during the monitoring phase. Wei et al. take a different angle and analyze the source code of the Linux 2.4 kernel to identify variables, structure fields, and array elements that are constant [WZS11]. During runtime, they monitor these variables and fields for changes and consider any occurring change to be an attack on the kernel. Similar to the approach of Petroni [PH07], Wei's source code analysis depends on manual annotations of type casts and generic pointers.

When InSight analyzes the source code of an introspected kernel, it also derives type invariants for data structures. However, they are not used to reveal attacks, but only to validate the mapped kernel objects and to rule out bogus objects resulting from dangling pointers or uninitialized data. This is because InSight is not an IDS, but a view-generating component. Thus, it is not the task of InSight to detect intrusions, but only to generate a usable view for other VMI applications, including—but not limited to—intrusion detection.

Another very interesting approach was introduced by Carbone et al. in the form of the KOP system [CCL^+09]. KOP performs a source code analysis of the Windows Vista kernel[1] to derive a detailed understanding of the kernel memory layout, the used data structures, and their fields. In contrast to the source code based approaches mentioned

[1]Even though the source code of Windows is not publicly available, some of the authors working for Microsoft Research had access to it.

so far, their analysis automatically recognizes pointer type casts and derives the correct data type for most pointers, even for '`void*`' pointers, without having to rely on manual annotations. From all related work in the field of VMI, this work is the one most closely related to our tool, InSight. We will revisit the KOP system in the related work section of Chapter 7 to discuss its implementation in greater detail and to compare it to our own semantic code analysis.

As a final representative of the delivery-based approaches, we describe the work of Inoue et al. [IADB11]. Their introspection framework, named min-c, consists of an introspection library built on top of Xen in combination with an interpreter for the C programming language. Introspection software for their framework is written in simplified C using a syntax similar to the one of a guest-native program. As a consequence, all introspection code has to incorporate domain knowledge to follow generic pointers or to read dynamic arrays. By consulting the compiler-generated type information for the introspected kernel, their C interpreter is able to resolve the types and symbols that refer to the guest kernel. During execution of the code, the interpreter translates pointer addresses to the memory range of the mapped guest physical memory, thus reading the objects from the guest kernel's address space. With these features, their system seems to be comparable to our own VMI framework that we introduce in Chapter 6 without the source code analysis technique and type rule engine described in Chapter 7.

All the approaches mentioned above make use of both hardware and software knowledge λ and μ, thus having the potential for full state applicability. However, they fail to achieve this property in a generic, guest OS-independent way due to their restrictive system design.

4.2 Derivative Approaches

In order to move away from an approach with the non-binding property, it is necessary to consider derivative approaches. Litty et al. present a system they call Manitou [LL06]. It uses the paging mechanism of x86-based processors from Intel and AMD to perform integrity checks on code pages before their execution. They further this work to create a system called Patagonix [LLCL08] for reporting running processes and maintaining the integrity of code pages. A similar approach has also been proposed by Wessel and Stumpf [WS12]. Such work uses knowledge of the hardware architecture, specifically the paging mechanism, to perform the intended task. They clearly are an implementation of the derivation pattern and are therefore bound to this hardware architecture.

Jones et al. take a slightly different approach, but also present a system that implements the derivation pattern [JADAD06, JADAD08]. Their system makes use of the paging mechanism and the MMU on x86 and SPARC hardware architectures to identify running processes.

The Ether system introduced by Dinaburg et al. [DRSL08] makes use of a page fault based mechanism to trap system calls for Windows XP guests running on 32-bit x86 platforms. In addition, there system is able to perform instruction-level tracing using the debugging functionalities of the CPU. However, since this mechanism results in a trap to the hypervisor for every single CPU instruction, the incurred overhead is tremendous and the resulting system hardly usable for real-world applications.

Another interesting approach from Vogl et al. leverages the performance monitoring counters within modern CPUs for trapping specific instruction types to the hypervisor [VE12]. This method is suited for multiple purposes, for example, to reconstruct the entire instruction stream of a specific process, including all kernel-level instructions, by tracing all branches in the code. In a different experiment, only '`call`' and '`return`' instructions are traced to defend the system against return-oriented programming attacks [Sha07] as follows. On each '`call`', the current instruction pointer is pushed onto a shadow stack, while on each '`return`' the return address on the process's stack is verified against the top-most address on the shadow stack. Even though this approach is very elegant from an engineering perspective, the high performance degradation of this method limits its usage to small programs.

In order to foster the usage of the derivative pattern, we also have investigated on the possibilities to derive virtual machine state information using the x86 architecture in joint work with Jonas Pfoh [PSE10]. We generalize from the simple derivate mechanisms of the previously mentioned approaches and identify numerous ways to bridge the semantic gap using hardware features. These novel and innovative methods led to the development of Nitro, a hardware-based system call tracer [PSE11]. Nitro supports tracing of all three hardware-assisted system call mechanisms for the x86 architecture, as shown in Figure 4.2, and has proven to produce excellent results for Windows and Linux guests, both for 32-bit and 64-bit flavors.

4.3 Summary

In this chapter, we have given an overview of related work in the field of virtual machine introspection. The relevant approaches have been classified according to our formal model that was introduced in the previous chapter. In addition, we distinguished the individual view generation processes based on their level of sophistication.

When reviewing the existing work in this area, it becomes apparent that the view generation process of almost all VMI approaches has been designed to support only one particular application, for example, the detection of specific intrusions or the implementation of a hypervisor-based firewall. However, previous work fails to identify the semantic gap as an inherent challenge for VMI and to address it accordingly. Rather, each approach implements its own, restricted way for bridging the semantic gap with manual annotations and hard-coded offsets.

The only three exceptions here are the Virtuoso [DGLZ+11], the min-c [IADB11], and the KOP system [CCL+09]. Nevertheless, these approaches either provide a high degree of automation but do not scale well for complex views (Virtuoso), or they are

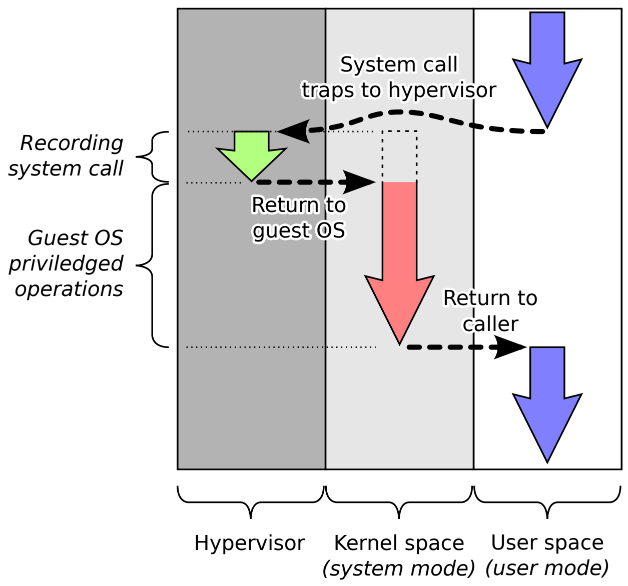


Figure 4.2: Control flow of a system call being intercepted by the hypervisor.

suitable for complex tasks but are entirely dependent on expert knowledge (min-c). Of the aforementioned systems, KOP provides the best compromise between automation of methods and scalability. Unfortunately, KOP relies on Windows specific internals to achieve the high precision and coverage of the generated views. Thus, these methods cannot readily be applied to generate views for different OSs.

With InSight, our own VMI framework, we focus on a versatile view-generating component (VGC) that supports arbitrary VMI applications and scales well for complex tasks with a high degree of automation independent of the introspected OS. While InSight currently targets Linux guests, it does not make any Linux specific assumptions about the guest and is expected to work for other OSs written in the C programming language with only minor modifications. Even though InSight makes use of multiple sources of semantic knowledge, it is highly unlikely that any VGC will be able to generate a perfect view without some manual adjustments to the view generation process. For this reason, our framework features a dedicated interface for user-supplied expert knowledge to guide the VGC in its decisions. This combination of automated methods, support for centralized and reusable expert knowledge, OS independence, and flexible interfaces sets InSight apart from all previous approaches in the field of VMI.

Chapter 5

View Generation with Full State Applicability

Bridging the semantic gap requires knowledge about the virtual hardware architecture, the guest operating system, or both. Recall that a VMI component that bridges the semantic gap by applying such knowledge is referred to as a view-generating component (VGC), as this component generates a useful “view” of the VM state that other VMI components can work with.

In Chapter 3, we discuss that an external VGC has access to the entire, untainted state of the introspected VM, a property called *full state visibility*. We also argue that, with *complete* semantic knowledge about the hardware and the guest OS, a VGC can theoretically generate a view of the guest's state that is as complete as the internal view of the guest OS itself. We call this property *full state applicability*. It can be achieved by combining the out-of-band delivery and derivation pattern.

Full state applicability is a powerful enabler for security applications and a strong argument for a combining the out-of-band delivery and the derivative approach. It is the main reason we chose this combination for our VMI research.

This chapter investigates the challenges that one must overcome in order to achieve full state applicability. We focus on the problem statements and do not describe how we address them here. Our approaches to handle these challenges are introduced in Chapter 6 and Chapter 7.

Chapter overview. We begin in Section 5.1 by taking a look at all parts of the state information available to an external VGC and their relevance for detecting system-level attacks. In addition, we estimate the efforts required to perform a semantic interpretation of each part. As we will see, the guest's physical memory is the most difficult of these parts. For this reason, all of Section 5.2 is dedicated to the discussion of the related challenges. We explain which semantic knowledge needs to be applied to the guest's memory to retrieve all kernel objects and why it remains a hard problem even with all this information at hand. Finally, we summarize the chapter in Section 5.3 and show the relevance of the achieved results for our own VMI approaches.

5.1 Interpretation of State Information

The initial problem statement in Section 1.2 says that we want to detect system level compromises of a guest OS using VMI. The most essential part of this problem is to create a VGC that provides full state applicability. That is, the component should be able to interpret the entire VM state with the same semantics as the guest kernel would without having to rely on any cooperation with the guest OS.

As described in Section 2.3, the state of a VM consists of the CPU registers, the main memory, the disk storage, as well as the state of all attached devices. Let us look at each of these portions of state information, the effort required to interpret it from the vantage point of the hypervisor, and their relevance for detecting system-level attacks.

5.1.1 CPU State

A modern CPU contains lots of different types of registers. Some of them are only used for arithmetic computations and are of no interest from a security perspective. Other registers configure the behavior of the CPU and how it interfaces with the running OS. They are usually called system registers, control registers, or MSRs and are crucial for the system's security.

The configuration options available through these registers is well specified as part of the hardware documentation. Consequently, all the semantic information required to interpret this part of the state can be delivered out-of-band in the hardware description λ. For example, the software developer's manual for Intel's IA-32 and 64 architecture [Int09] describes that the control register 3 (CR3) holds the physical address of the top-level page directory for the running process. It also defines the format of the page tables at different levels and for different addressing schemes.

This example illustrates that interpreting the system registers of a CPU is straightforward. Since this step requires only hardware information, this knowledge is binding, as described in Section 3.3.1.3. The virtual hardware interface is well-defined, and an attacker has no means to change it. In addition, this knowledge is independent of the type of OS running on the CPU.

5.1.2 Device State

Similar to a CPU, the state of a device may contain configuration values that control the device's behavior. Depending on the device type and the way it is virtualized (cf. Section 2.1.3), this state information might be relevant for security.

Unlike CPU vendors, device manufacturers tend to provide their own software that interfaces with the device in form of drivers rather than publishing a detailed specification of their hardware's inner working. If a specification of the semantics of the device's state is available, it can be included in the hardware description λ. If such information is not available, interpretation of the state becomes very hard, if not impossible.

Fortunately, we have not come across any system-level attack so far that is based on the manipulation of a device's state, so this limitation has no serious consequences for detecting such attacks.

5.1.3 Disk Storage

The disk storage contains the file system of the guest OS and all configuration files and system binaries, including the kernel itself. Thus, it is an important part of the state with regard to system security.

Accessing a VM's file system is not a problem in most cases. If it resides on a dedicated physical disk or partition, it can be directly mounted by the host OS to access its files, provided the host supports the guest's file system. The Linux kernel, for example, comes with support for most popular file systems, including Windows' NTFS and Mac OS's HFS. In case the disk storage is provided by means of some virtual disk container, this process gets only slightly more complicated. Most hypervisors come with tools to access the file system within their containers and to convert between different container formats used by other hypervisors. Using these tools, interpreting the disk storage of a VM becomes effortless.

In terms of our model, mounting a VM's disk using the host's implementation of the guest's file system effectively represents a form of external view generation. The knowledge about the file system's internal structure and semantics is part of the software architecture description μ. However, since the corresponding file system code within the guest OS might be subject to attacks, this information is non-binding. A malicious entity might deceive view generation by changing this code to a different file system semantics. For example, the TDSS rootkit family comes with its own implementation of a custom, encrypted file system [She09, GR10]. Thus, its files cannot be included in the view unless the corresponding semantic information is added to μ.

5.1.4 Physical Memory

The guest OS kernel keeps its state in the system's main memory in form of kernel objects, as previously described in Section 1.2. These objects and their contained values drive the kernel's behavior through configuration options and jump tables, keep book of the resources available and in use, define the privileges of processes and access rights on files, and so on. Obviously, this state information provides an enormous potential for an adversary when being tampered with, as the system-level attacks described in Section 2.4 bear witness. As a consequence, it is absolutely essential for system security to be able to read the kernel objects from memory and to interpret them as the guest OS kernel would.

In order to read the kernel objects, the following semantic knowledge of the guest's hardware and software architecture description λ and μ must be applied to the guest physical memory:

1. the layout of the memory used by the kernel, i. e., the kernel address space ($\subset \mu$)
2. the layout and size of the kernel's data structures ($\subset \mu$)
3. the location of data objects in the kernel's address space ($\subset \mu$)
4. the function of the virtual-to-physical address translation ($\subseteq \lambda$)

Given that this information is complete, the whole set of kernel objects can be retrieved by starting at the global objects listed in μ and from there recursively follow all pointers to further objects until all objects are found. This strategy, when carried out perfectly, will indeed reveal *all* objects: As we already argue in Section 2.4.1.2, the kernel itself must keep at least one link to each object in order to consider it for its decisions. If the kernel has lost track of an object, it looses awareness of that object, and thus the object has no more influence on the kernel's behavior.

Following that line of thought, we model the state of the kernel as a directed graph in which the kernel objects represent vertices and the pointers that link to other objects represent directed edges. If a vertex represents an elementary data type, such as an integer value, this value becomes an additional property of this vertex. Structure fields of such elementary data types become vertices themselves, linked to their embedding object. An example of such a graph is illustrated in Figure 5.1.

Using the semantic knowledge listed above to build the complete and correct kernel object graph is a hard task. The next section helps to understand the challenges related to that task and to the application of the semantic knowledge.

5.2 Challenges of Kernel Object Extraction

The guest physical memory is the most difficult part of a VM's state information to interpret. The previous section lists the semantic knowledge that is required to read kernel objects from memory and to eventually build the graph of kernel objects. In the following, we describe the challenges involved in applying this knowledge to retrieve the entire set of kernel objects.

5.2.1 Memory Layout and Address Translation

The kernel objects reside in a dedicated part of the main memory. Most modern OSs support the concept of virtual memory, typically utilizing a hardware MMU (see Section 2.1.2). The MMU is in charge of translating virtual to physical addresses whenever a virtual memory address is accessed. This translation is done on a per-process basis, so every process lives in its own virtual address space separated from other processes.

When virtual memory is supported, the kernel itself also stores its objects in some virtual address space. How the kernel's virtual address space is implemented differs across various OSs. For Windows and Linux systems, for example, the kernel code and

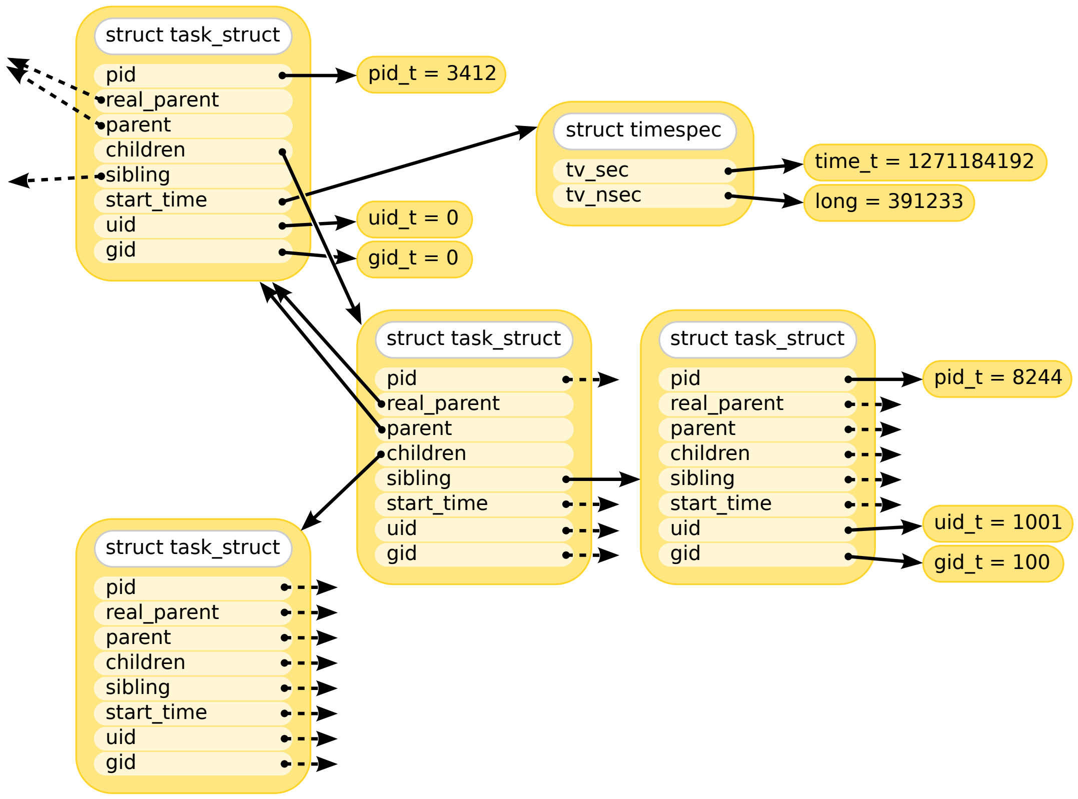


Figure 5.1: A part of a kernel object graph.

data occupy a fixed virtual address range that is shared among *all* processes' address spaces. In contrast, most BSD-based systems such as FreeBSD or MacOS X dedicate a separate virtual address space to the kernel.

5.2.1.1 Virtual-to-Physical Address Translation

Regardless of the OS specific implementation, the VGC always accesses the VM's state at the level of the virtual hardware. This interface limits the component to reading (and probably writing) the guest's memory at the guest physical level, as Figure 6.1 illustrates. Consequently, the VGC needs to perform the same address translation as the MMU to read kernel objects from virtual addresses. To accomplish this translation, the details of the address space layout are required along with the locations of the necessary information in memory. For the x86 platform, for example, this information consists of:

- the *physical* address of the top-level page directory of the kernel's address space
- the operation mode of the CPU, which can be one of:
  - 32-bit mode without physical address extension (PAE)

  - 32-bit mode with PAE
  - 64-bit mode (i. e., “long mode”)

Part of this information can be obtained by checking various control registers of the virtual CPU. However, the CPU state might not always be available, for example, when performing an offline analysis of a previously stored memory dump. In such cases, this information can be delivered as a part of the hardware knowledge λ that the VGC requires to interpret a given state.

5.2.1.2 Swapped-out Memory

When utilizing virtual memory support, most OSs are able to make more total memory available to their processes than there is physical memory installed on the system. This is achieved by identifying memory that has not been accessed for a while and swapping its contents out to a dedicated *swap space* on the hard disk. The swapped memory region can then be handed out to a different process in need of more memory. Each process’s page table records which pages are swapped out and another table keeps track of where in the swap space these pages have been saved.

When reading a kernel object from a virtual address, a VGC might encounter memory pages that are swapped out to disk. To provide full access to all kernel objects, either the VGC must be able to interpret those tables and read the memory that has been swapped out from disk, or swapping of kernel memory must be avoided.

5.2.2 Type Information

In order to read kernel objects from memory, information about the data types that the kernel uses, that is, the *kernel symbols*, are required. If the source code of the OS kernel is available, as is the case for Linux, FreeBSD or OpenBSD, the kernel can be compiled with specific compiler flags to generate debugging symbols for it. The symbols contain, amongst other information:

- the names of all data types, structures, unions, enumerations, and type aliases (‘`typedef`’),
- the fields, offsets and memory alignments for structures and unions,
- the names, virtual addresses, and segments of global variables, and
- the names and parameters of functions and the virtual addresses of their entry points.

This knowledge is sufficient for reading kernel objects from a known location within the virtual address space.

Going forth, when talking about "data structures" in general, this term refers to both '`struct`' and '`union`' types as specified by the ANSI C standard [ISO90], unless otherwise stated.

5.2.2.1 Consolidation of Type Information

For code written in C, it is common practice to include the same data types and function declarations through header files in multiple translation units (i. e., source files). Most compilers, however, create self-contained debugging symbols for every translation unit. As a consequence, the debugging symbols for the whole kernel consisting of thousands of source files contain a lot of redundant type information. This results in two problems: First, the type information database becomes bloated and thus less efficient, and, second, objects of the same type found in different translation units appear as having a different type. For building a correct graph of kernel objects, the VGC must be able to recognize objects of the same type as such. Consequently, we must be able to detect and merge equivalent data types into one single type each.

Microsoft takes a different approach with their Visual C/C++ compiler for the Windows OS. For the purpose of debugging, they offer type information for most of their core libraries in the form of separate "program database" (PDB) files. Those files can be obtained from Microsoft's Windows Symbol Server by using the Debugging Tools for Windows [win]. The advantage of this approach is that the linker automatically merges type information originating from the same header file when linking multiple objects to one binary. This solves the aforementioned problem of type equivalence at least on a per-binary level.

5.2.3 Dynamic Pointer and Type Manipulations

The debugging symbols for the kernel contain information about the interconnection of kernel objects in form of pointer fields of structures to further data structures. However, the C language in which OS kernels are mostly written allows to dynamically change pointer locations and types by means of pointer arithmetic and type casts. This runtime behavior is not reflected in the debugging symbols, rendering this information not only incomplete but sometimes even misleading. In the following, we give several examples for such "pointer magic" to illustrate the fundamental problem.

The examples given in this section are based on the Linux kernel and are easily reproducible due to the openness of Linux. When considering the general design of modern operating systems regarding performance and efficient usage of system resources [Tan07], it becomes evident that similar instances can most likely be found in any other comparable OS kernel such as Microsoft Windows or other UNIX derivates. As a consequence, the conclusions we draw from the Linux examples also apply to other OSs in one way or the other.

5.2.3.1 Linked Lists

OS kernels organize objects in efficient data structures such as linked lists or hash tables, for example. A common implementation of these lists and tables works by defining generic "node" data structures along with auxiliary functions and macros to operate on them. These nodes must then be embedded as a field of the kernel object's definition to be organized in a list or table. To iterate over such a list or retrieve hashed objects based on a key, the auxiliary functions only operate on the embedded nodes, but still make the embedding object available through sophisticated address manipulations and type casts.

Example 5.1 In the Linux kernel, the node structure for doubly linked lists is called '`struct list_head`'.[1] This structure only contains two pointers, '`next`' and '`prev`', and is defined as follows:

```
/********** Source: <include/linux/list.h> ***********/
struct list_head {
    struct list_head *next, *prev;
};
```

▲

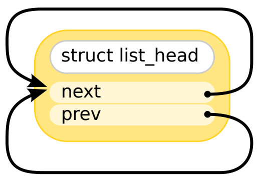
One of the many data structures that makes use of the '`list_head`' node is '`struct module`', which describes a loadable kernel module. However, the head to this list is not an object of type '`struct module`', as the following example shows:

Example 5.2 The definition of the '`module`' structure embeds a field of type '`struct list_head`' to arrange the corresponding objects in a doubly linked list. The global variable '`modules`' serves as the head to this list and is initialized to point to itself:

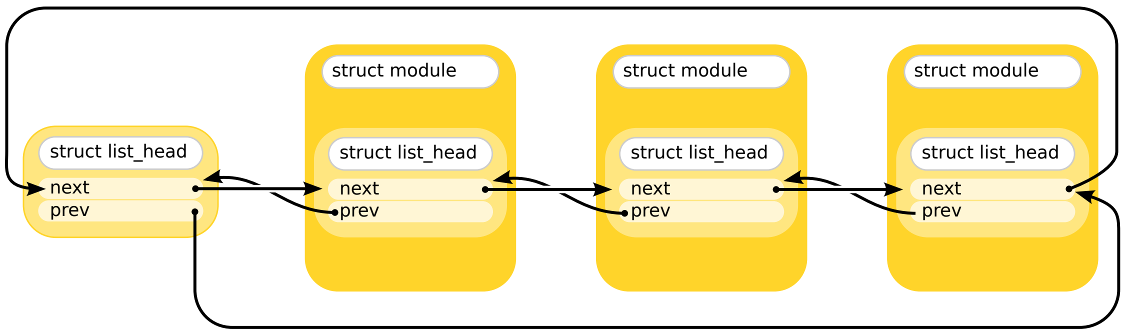
```
/********** Source: <include/linux/module.h> ***********/
struct module {
    enum module_state state;
    struct list_head list;
    char name[MODULE_NAME_LEN];
    /* ... */
};

/********** Source: <kernel/module.c> ***********/
static struct list_head modules = { &modules, &modules };
```

▲

[1]For Windows, the node structure for doubly linked lists is called '`LIST_ENTRY`' and works exactly in the same way as described for Linux's '`list_head`'.



(a) Empty list



(b) List with three elements

Figure 5.2: A doubly linked list of ‘module’ objects utilizing ‘struct list_head’.

This list is then accessed using the variable ‘modules’ and operated on by the provided auxiliary functions and macros. They only operate on the ‘next’ and ‘prev’ pointers of field ‘list’ which hold the addresses of field ‘list’ *within* the next and previous ‘module’ objects, respectively. The last object points back to the head of the list. An illustration of this pointer linkage is shown in Figure 5.2. To get access to the ‘module’ object that embeds the ‘list’ field, other macros are provided that subtract the offset of field ‘list’ from the pointer value and perform a cast to the requested type ‘struct module*’.

5.2.3.2 Functions and Macros Operating on Lists

In order to better understand the inner workings of lists and similar data structures and the resulting problems for view generation, it is worth looking at the auxiliary functions that operate on them. Typically, these operations are implemented as preprocessor macros or as in-line functions.

Example 5.3 Consider the simple function ‘find_module’ in Listing 5.1 that finds a loaded kernel module by its name. The loop starting at line 6 iterates over all ‘module’ objects, compares the name to the given one, and returns the matching object, if found.

As it turns out, the loop initialization ‘list_for_each_entry()’ actually is a macro for iterating over a list and expects three arguments: a temporary variable which points the object at each loop iteration (which is ‘mod’ in our example), a pointer to the head of the list (‘&modules’), and the field of type ‘list_head’ in the embedding data structure

Listing 5.1: Searching for a module by name.

```c
/********** Source: <kernel/module.c> ***********/
struct module* find_module(const char *name)
{
    struct module *mod;

    list_for_each_entry(mod, &modules, list)
    {
        if (strcmp(mod->name, name) == 0)
        return mod;
    }
    return NULL;
}
```

(‘`list`’).

Listing 5.2 shows the same function after being preprocessed by the compiler. The original macro ‘`list_for_each_entry`’ has been expanded to a lengthy ‘`for`’ loop which performs the list iteration. This loop requires some explanation:

- The initialization in line 6 assigns the value of ‘`(&modules)->next`’ − *offset* to the ‘`mod`’ local variable[2], where *offset* is the offset of field ‘`list`’ within ‘`struct module`’, calculated using the compiler-specific function ‘`__builtin_offsetoff`’. This calculation involves the additional temporary variable ‘`__mptr`’. Finally, the resulting pointer is cast to type ‘`typeof(*mod)*`’ in line 8, which evaluates to ‘`struct module*`’.

- The loop condition in line 10 compares the address of the current object’s ‘`list`’ field to the address of the list’s head (‘`&modules`’). The end of the list has been reached when the list wraps around, as depicted in Figure 5.2.

- The loop advancement in line 11 is almost identical to the initialization, the only difference being that ‘`mod`’ is now assigned ‘`mod->list.next`’ − *offset*. The temporary variable ‘`__mptr`’ and the type cast to ‘`struct module*`’ remain the same.

▲

This example involves several instances of dynamic pointer and type manipulations which makes it difficult to find all objects within this list. First, the static type information for variable ‘`modules`’ only indicates that the ‘`next`’ and ‘`prev`’ fields point to

[2]The right-hand side of expression ‘`mod = ({...})`’ is an extension of the GNU C compiler called *compound braces statement*. It expects a list of (almost) arbitrary definitions and statements between the curly braces. The resulting value of the compound braces statement is the value of the last statement within this expression. [St10]

Listing 5.2: Macro expansion of '`list_for_each_entry()`' from Listing 5.1.

```
struct module* find_module(const char *name)
{
    struct module * mod;

    /* expansion of: list_for_each_entry(mod, &modules, list) */
    for (mod = ({
            const typeof(((typeof(*mod) *) 0)->list) * __mptr = ↩
                ((&modules)->next);
            (typeof(*mod) *) ((char *) __mptr - ↩
                __builtin_offsetof(typeof(*mod), list));
        });
        &mod->list != (&modules);
        mod = ({
            const typeof(((typeof(*mod) *) 0)->list) * __mptr = ↩
                (mod->list.next);
            (typeof(*mod) *) ((char *) __mptr - ↩
                __builtin_offsetof(typeof(*mod), list));
        }) )
    {
        if (strcmp(mod->name, name) == 0)
        return mod;
    }
    return NULL;
}
```

objects of type '`list_head`', but there is no hint regarding the embedding objects of type '`struct module`'. Second, even if the VGC knows that '`modules.next`' points to a '`module`' object, the value of *offset* to calculate the correct base address of that object is unknown. One might assume that *offset* is the offset of a field of type '`list_head`' within '`struct module`', however the complete definition of this structure includes three such fields, rendering this heuristic useless. Third, once the VGC has access to a '`module`' object and follows the pointers of the embedded '`list`' field, it again loose awareness of the embedding objects of type '`module`' and the correct value of *offset*.

5.2.3.3 Data Structures with Multiple Lists

The Linux kernel contains even more problematic use cases of lists that have not yet been covered. For instance, many structures contain fields of type '`list_head`' as the heads to lists of objects with different types.

Listing 5.3: Structures for managing PCI devices in the Linux kernel.

```
/********** Source: <include/linux/pci.h> ***********/
struct pci_bus;

struct pci_dev {
    struct list_head bus_list;   /* node in per-bus list */
    struct pci_bus *bus;         /* bus this device is on */
    /* ... */
};

struct pci_bus {
    struct list_head node;       /* node in list of buses */
    struct pci_bus *parent;      /* parent bus this bridge is on */
    struct list_head children;   /* list of child buses */
    struct list_head devices;    /* list of devices on this bus */
    /* ... */
};
```

Example 5.4 In Listing 5.3, '`struct pci_bus`' describes the state of a PCI bus, which is organized in a linked list using its field '`node`'. In addition, each PCI bus manages a list of devices through its field '`devices`'. The state of each PCI device is contained in objects of type '`struct pci_dev`', however the fields '`node`' and '`devices`' both are of type '`list_head`'. ▲

This example demonstrates that the correct type and offset for '`list_head`' fields cannot be trivially guessed based on the embedding type. Instead, detailed information about the dynamic pointer and type manipulations is required per field.

To make matters worse, the offsets that are subtracted from the '`next`' and '`prev`' pointer values are not always consistent. For example, the data structure '`task_struct`' arranges the process descriptors in a tree to reflect the parent, children, and sibling relations between the processes using two '`list_head`' fields '`children`' and '`sibling`'. These relationships are illustrated in Figure 5.3. Contradicting to the common usage pattern, the pointer '`children.next`' does not point to the embedding field '`children`' of the first child's process descriptor but rather to field '`sibling`' of the first child. Analogous, '`children.prev`' points to the '`sibling`' field of the last child's process descriptor.

The kernel not only uses doubly linked lists; there are several other data structures that work in a similar way, for example, hash tables and red-black trees. Identifying all of them and handling all special use cases manually is cumbersome and highly error-prone.

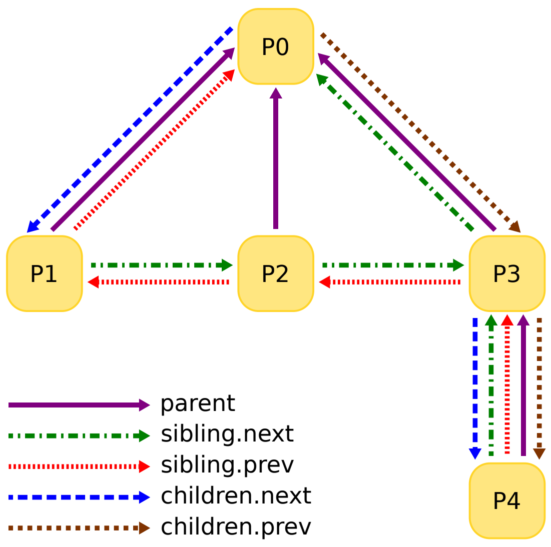


Figure 5.3: The Linux kernel organizes its process descriptors in a tree to reflect the parent, children, and sibling relations between the processes (illustration adapted from [BC05]).

5.2.3.4 Generic Pointers

Another problem results from objects that are pointed to by generic pointers such as '`void*`' or '`char*`', or pointers that are cast at runtime from integer values. In case of a generic pointer, the VGC can at least assume that the pointer should hold a memory address that could point to another object, a buffer, or a function. For integer values that are cast to pointers, there is no easy way to determine if a numeric value represents a memory address or not.

In addition, some integers hold an address together with some flags or status bits stored in the least significant bits. The nodes of a radix tree having type '`struct radix_tree_node`', for instance, use one address bit to encode an internal, binary property of the node without wasting additional memory. This bit is masked out at runtime before the resulting pointer is dereferenced. For such cases, the values of these masks and the involved arithmetic operations are required knowledge to access the referenced objects.

5.2.4 Runtime Dependencies

Kernel code not only contains static type ambiguities as described in the previous subsection, but also runtime dependencies which cause dynamic ambiguities. We describe these situations and explain why they cannot be handled statically, that is, at compile time.

5.2.4.1 Ambiguous Types

The ANSI C standard [ISO90] allows the specification of inherently ambiguous types using a ‘union’ definition. Such a data type is specified in the same way as a ‘struct’, however it only occupies memory equal to the size of its largest field. That is, only one if its fields may be used at any given time. It is left up to the programmer to know which field is valid at which position of the control flow.

Example 5.5 The ‘page’ data structure, defined in Listing 5.4, makes heavy use of ‘union’ fields. The Linux kernel employs it to describe the current usage of a physical memory page (see Section 2.1.2). All fields having the keyword “SLUB” in their comment are only valid when the corresponding page is currently used by the SLUB memory allocator [Cor07]. Thus, the ‘union’ fields help to reduce the memory footprint of the kernel’s management data. ▲

When a VGC builds up the kernel object graph, the objects found are not bound to any control flow point. From the vantage point of the VGC, the objects—including ‘union’ objects—have no context that allows to determine their current usage. Consequently, without additional information, all fields of a ‘union’ object are valid with equal likelihood.

5.2.4.2 Dynamic Arrays

Another source of ambiguity are pointers that are used as arrays. When the kernel requires variable length arrays, they are usually defined as pointers within a structure.

Example 5.6 An data structure might be used as a variable length array using a pointer field and an integer field:

```c
struct dynamic_array {
    struct foo* array;
    unsigned int len;
};

void array_alloc(struct dynamic_array *a, unsigned int len)
{
    a->array = kmalloc(sizeof(struct foo) * len);
    a->len = len;
}
```

The memory for this array is allocated at runtime with the base pointer stored in ‘array’ and the number of array elements stored in ‘len’. ▲

When encountering an object of this type, the debugging symbols only show a pointer to a single structure ‘foo’ and an integer. The relationship between ‘array’ and ‘len’

Listing 5.4: The ‘page’ structure contains several nested ‘union’ fields to reduce the memory footprint of the corresponding objects.

```
/********** Source: <include/linux/mm_types.h> ***********/
struct page {
   unsigned long flags;         /* Atomic flags */
   atomic_t _count;             /* Usage count */

   union {
      atomic_t _mapcount;       /* Count of ptes mapped in mms */
      struct {                  /* SLUB */
         u16 inuse;
         u16 objects;
      };
   };

   union {
      struct {
         unsigned long private; /* Mapping-private opaque data */
         struct address_space *mapping; /* Points to address_space */
      };                                /* or to anon_vma object */
      spinlock_t ptl;
      struct kmem_cache *slab;  /* SLUB: Pointer to slab */
      struct page *first_page;  /* Compound tail pages */
   };

   union {
      pgoff_t index;            /* Our offset within mapping. */
      void *freelist;           /* SLUB: freelist req. slab lock */
   };

   struct list_head lru;        /* Pageout list */
};
```

is in no way obvious at this level. Without further knowledge, a VGC is limited to following the pointer '`array`' to a single element of '`foo`', effectively ignoring the fact that '`len`' might be set to zero, in which case the VGC would follow a dangling pointer, or to a number greater than one, resulting in an incomplete view.

There are other instances of structure fields that are guarded by flags inside of a kernel object. If this field happens to be a pointer and the VGC is unaware of that flag indicating whether that pointer is valid, it will eventually follow a dangling pointer in the same manner as for an array of length zero.

5.2.4.3 Dynamic Object Sizes

The kernel has no resources to waste and tries to save memory whenever possible. The Linux developers go even so far to allocate less memory for an object than its definition specifies, if it can be assured that not all of its fields are used.

Example 5.7 Listing 5.5 shows an example of a data structure whose size is defined dynamically at runtime. The data type '`struct kmem_cache`' is used by the SLUB memory allocator. As the listing shows, its last field '`cpu_slab`' in line 10 is an array of length '`NR_CPUS`' of pointers to per-CPU caches. '`NR_CPUS`' is a compile-time constant value for the maximum number of supported CPU cores on a system and typically ranges between 32 (IA-32) and 512 (AMD64) on PC platforms.

The size of a '`kmem_cache`' object is stored in the global variable '`kmem_size`' at line 14 and is first assumed to equal the declared size of '`struct kmem_cache`'. At boot time, the SLUB allocator is initialized and '`kmem_size`' is adjusted in line 19 according to the actual number '`nr_cpu_ids`' of CPUs that were detected. When later on new '`kmem_cache`' objects are allocated in line 29, only the required amount of memory is reserved, since all array fields with index '`nr_cpu_ids`' $\leq i <$ '`NR_CPUS`' will never be accessed anyway. ▲

A VGC that encounters a pointer to a '`kmem_cache`' object is unaware of the memory savings that the kernel performs. Given that '`nr_cpu_ids`' $<$ '`NR_CPUS`', it might find objects that seem to overlap based on their declared size. What is worse, it might even try to follow pointers of the array field '`cpu_slab`' with indices $i \geq$ '`nr_cpu_ids`', resulting in bogus objects within the object graph.

5.3 Summary

External view generation with full state applicability is the goal we are striving for. The state of the attached virtual devices is hard to interpret, however it has little relevance for VMI-based security applications. The situation is different for the state information describing the virtual CPU and the hard disk: It is both important for security as well as

Listing 5.5: The size of a '`struct kmem_cache`' object is dynamically determined at boot time, independent of its defined size.

```c
/********** Source: <include/linux/slub_def.h> ***********/
struct kmem_cache {
    /* Used for retriving partial slabs etc */
    unsigned long flags;
    int size;          /* The size of an object including meta data */
    int objsize;       /* The size of an object without meta data */
    int offset;        /* Free pointer offset. */
    struct kmem_cache_order_objects oo;
    /* ... */
    struct kmem_cache_cpu *cpu_slab[NR_CPUS];
};

/*********** Source: <mm/slub.c> ***********/
static int kmem_size = sizeof(struct kmem_cache);

void __init kmem_cache_init(void)
{
    /* ... */
    kmem_size = offsetof(struct kmem_cache, cpu_slab) +
                         nr_cpu_ids * sizeof(struct kmem_cache_cpu *);
    /* ... */
}

struct kmem_cache *kmem_cache_create(const char *name, size_t size,
              size_t align, unsigned long flags, void (*ctor)(void *))
{
    struct kmem_cache *s;
    /* ... */
    s = kmalloc(kmem_size, GFP_KERNEL);
    /* ... */
    return s;
}
```

straightforward to interpret. The most challenging part for a VGC is the guest physical memory of the virtual machine.

For defending a system against system-level attacks, we focus on the memory that contains the kernel's state in form of kernel objects. Our approach is to build the graph of all objects used by the kernel, interconnected by pointers. If this is to be done with a high level of completeness and accuracy, this task becomes very challenging, as we have shown throughout this chapter. The C programming language that most OS kernels are written in allows developers to create highly efficient code. However, this efficient code tends to make view generation very difficult. Based on the source code of the Linux kernel, we have shown several instances of such dynamic pointer and type manipulations as well as other runtime behavior. In order to cope with this dynamic behavior in an automated fashion, a VGC requires much more information than the kernel debugging symbols alone.

The VMI framework that we introduce in Chapter 6 is a first step towards view generation with the potential for full state applicability. In Chapter 7, we propose a novel method for automatically collecting additional semantic information based on the kernel's source code to solve many of the challenges that were detailed in this chapter. To handle the remaining cases which cannot be solved autonomously, we extend our VMI framework with a flexible rule engine which allows an easy incorporation of knowledge from a human expert in addition to the collected semantic knowledge. With this combined approach, we are able to address all instances of dynamic pointer and type manipulations as well as resolve runtime dependencies such as we described here.

Chapter 6

Towards Full State Applicability

In this chapter, we introduce our VMI framework, *InSight*, created as part of our research for bridging the semantic gap. The heart of InSight consists of an external view-generating component. Recall that according to our model, any VMI approach can generate an internal view using $f_{\lambda,\mu}$ and an external view using $g_{\lambda,\mu}$. The aggregation function a combines these views over time into a profile, which in turn is classified by the decision function d.

InSight focuses on the external view-generating component $[\![g_{\lambda,\mu}]\!]$ which makes use of both hardware knowledge λ and software knowledge μ, as illustrated in Figure 6.1. With the complete state visible, it has the potential for full state applicability. The framework provides several interfaces that allow other components to further process the generated view. In other words, InSight decouples the view generation from the aggregation and classification step and strives towards making the implementation of the components $[\![a]\!]$ and $[\![d]\!]$ as easy and straightforward as possible.

In contrast to our approach, previous VMI applications generally do not aim for full state applicability. Their view generation typically targets exactly the portion of the VM state necessary for a particular use case. This is due to the fact that bridging the semantic gap is an admittedly difficult challenge [CN01]. Consequently, the way the resulting systems bridge the semantic gap is very much tailored to only provide access to the guest data relevant for the current task. The employed VGCs often rely on manual annotations and expert knowledge such as OS version specific offsets, data types, and sizes. Nevertheless, each of these approaches basically "reinvents the wheel" with regard to view generation, when in fact the primary goal was to introduce new VMI applications. We have outlined several such approaches with restricted view generation and manual annotations in Section 4.1.1 and 4.1.2, respectively. Recently, some research has begun to investigate stand-alone view generation [CCL+09, DGLZ+11, IADB11]. Such work has, to this point, addressed aspects such as automation, but in its current state, it lacks the completeness and flexibility necessary for the wide range of possible VMI applications.

InSight is one of the first approaches to focus on view generation with full state ap-

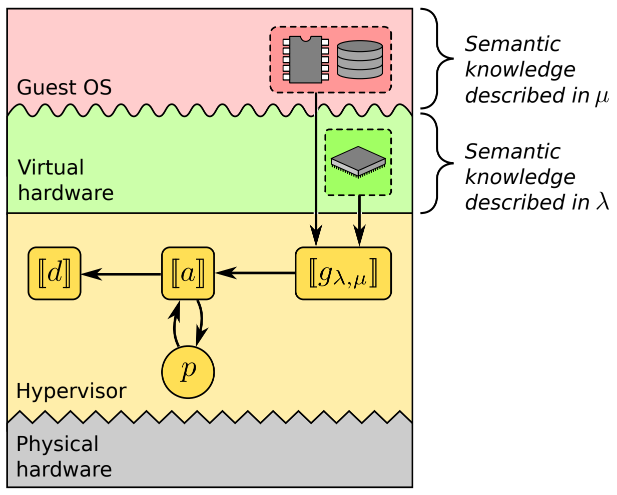

Guest OS
Virtual hardware
Hypervisor
Physical hardware
Semantic knowledge described in μ
Semantic knowledge described in λ
$[\![d]\!]$
$[\![a]\!]$
$[\![g_{\lambda,\mu}]\!]$
p

Figure 6.1: The *InSight* VMI framework uses a combination of the out-of-band delivery and derivation patterns for view generation in $[\![g_{\lambda,\mu}]\!]$. The components $[\![a]\!]$ and $[\![d]\!]$ can be tailored towards a given VMI application using the framework's interfaces.

plicability, independent of the VMI application at hand. Instead of relying on manual annotations and expert knowledge, InSight is designed to automatically adapt to differences among guest kernel versions and virtual hardware platforms. For example, InSight supports view generation for Linux guests on the x86 platform for all three addressing schemes (cf. Section 5.2.1) and all Linux kernel versions from 2.6 to 3.5, the latest version by the time of this writing. A VMI application that is implemented using the interfaces InSight provides will run without changes for all supported guest kernels.[1]

Chapter overview. In the following, we describe the design and implementation of InSight. We begin with a short overview of the features of our framework and how it may be applied for various VMI tasks in Section 6.1. This is followed by an in-depth description of the techniques applied in InSight's implementation in Section 6.2. This covers the areas of managing the type information, accessing the kernel memory, and representing and handling kernel objects, as well as the interfaces which provide access to the types and kernel objects. We sketch some applications VMI applications that make use of our framework in Section 6.3 and give some concluding remarks in Section 6.4.

6.1 Overview of InSight

InSight is a framework meant to ease the implementation of VMI-based security applications. The core of this framework is a versatile view-generating component with the

[1]It runs without changes subject to the caveat that the VMI application itself does not rely on internals of the kernel that have been changed in newer kernel versions, of course.

potential for full state applicability. However, InSight by itself is neither an intrusion detection system nor any other type of VMI application; rather it provides everything required to build such applications. Our framework is implemented in the C++ programming language and follows the object-oriented paradigm. It is based on the Qt 4 libraries [qt] for better host portability. We have successfully tested InSight to run in Linux and Windows host environments and expect it to work on MacOS X and other BSD derivatives with at worst minimal code changes.

6.1.1 Supported Guests

InSight allows the creation of external views of Linux guests running inside of VMs that provide an x86 hardware interface (i. e., Intel IA-32 or Intel 64/AMD64). It has been successfully tested with Linux 2.6, 3.0, and 3.2 kernel versions and is expected to work for any other kernel of the 3.X series. View generation focuses on the guest physical memory of the VM, which is the most challenging part of the available state information to interpret (cf. Section 5.1). InSight is able to read any kernel object from physical memory and supports all three addressing schemes of the virtual x86 hardware: 32 bit with and without PAE as well as 64 bit.

6.1.2 Supported Hypervisors

Our framework does not limit itself to working in conjunction with a particular hypervisor. Since it operates on guest physical memory, it supports any hypervisor that provides linear read access to the physical memory of the inspected VM. In our experiments, the kernel-based virtual machines for Linux (KVM [KKL+07, Sha08]) have proven to work well with InSight. In addition, a VM's physical memory image saved as a dump file (i. e., a *memory snapshot*) can be analyzed just as well. Because of this, InSight is especially useful for offline analysis of virtual machine states.

6.1.3 Modes of Operation

To support different usage scenarios, InSight is split up into two executables: a daemon called *insightd*, and a front-end called *insight*. The daemon contains all the logic and intelligence for view generation and normally runs as a regular terminal application in the foreground. But as the name suggests, the daemon can also detach itself from the controlling terminal to continue execution in the background. When running detached, the front-end executable allows control of the daemon and attachment to its command line interface. This mode of operation is meant for continuous, periodic introspection of a VM during its regular execution.

6.1.4 Interfaces

The view that is generated by the framework's VGC can be accessed in three different ways, described below.

6.1.4.1 Command Line Interface

InSight comes with a powerful command line interface, similar to a UNIX shell. This interface is most useful for interactive inspection, browsing through the available types, variables, and functions, and reading single kernel objects and their values from memory. This mode of operation is similar to the traditional command line interface of a debugger. In addition to that, the InSight shell allows one to execute scripts, to manage the type information, and to load different memory snapshots of the introspected VMs.

6.1.4.2 JavaScript Interface

A scripting engine built into the framework allows developers to implement complex analysis tasks in JavaScript. In the scripting engine, kernel objects read from memory are encapsulated in native JavaScript objects, thus allowing an intuitive interaction with the kernel state they represent. For example, suppose '`t`' is a JavaScript variable holding a process descriptor of a Linux guest, then the code '`print(t.comm)`' will output the '`comm`' field of the corresponding data structure, which holds the command line of the executing binary. Due to the nature of scripting languages, all errors within the script code are contained within the scripting environment and do not propagate to InSight. This makes the development of new VMI applications both convenient and safe.

6.1.4.3 Library Interface

For even more flexible control of the analysis, InSight comes with a shared library that allows an application to interface with the framework using C++ code. By linking against this library, both the aggregation and classification components $[\![a]\!]$ and $[\![d]\!]$ can be implemented efficiently as host-native executables while still preserving the benefits provided by the excellent view generation component of the framework.

6.1.5 Availability

Using the VGC of our framework through any of these interfaces, the actual VMI application is completely decoupled from the view generation process, making InSight a universal tool for various applications such as intrusion detection, forensic analysis, as well as kernel debugging.

We have released the InSight source code under an open source license [ins] to foster the fast and intuitive development of new VMI and forensic approaches.

6.2 Design and Implementation

After having introduced the basic features of InSight in the previous section, this section details the design and implementation of the VMI framework. Note that some aspects of the implementation we describe here do not reflect the latest development status of InSight. Rather, they represent evolutionary steps that were taken while striving towards our goal, which is view generation with full state applicability. Nevertheless, we include them here as a motivation for our further work in Chapter 7.

6.2.1 Type Information

In order to read kernel objects from memory, InSight requires information about the data types that it expects to find there as well as a way to describe these types internally.

6.2.1.1 Representation of Data Types

The data types that the introspected kernel uses require an internal representation describing their properties, such as the name and the fields of a data structure. In our framework, all data types are represented as objects of one of the classes shown in Figure 6.2. Each class is a descendant of the abstract class *BaseType* which holds elementary information such as a unique identifier, the type's name, and its storage size. For the purpose of view generation, many data types have similar properties and differ only in details.

Example 6.1 All numeric types represent numbers, but they differ in their storage size and their encoding. The following table shows the numeric types that occur within the Linux kernel compiled with the GNU C compiler for the x86 platform, along with the names of the corresponding classes representing these data types internally:

	Encoding			
Size	*signed*	*unsigned*	*boolean*	*float*
8 bit	*Int8*	*UInt8*	*Bool8*	
16 bit	*Int16*	*UInt16*		
32 bit	*Int32*	*UInt32*		*Float*
64 bit	*Int64*	*UInt64*		*Double*

▲

The inheritance diagram in Figure 6.2 shows further abstract classes that introduce new functionality shared by all descendants. Most of them do not require any further explanation, with the exception of *ReferencingType*. This class provides a reference to another *BaseType* object. As a consequence, all classes inheriting from *RefBaseType*,

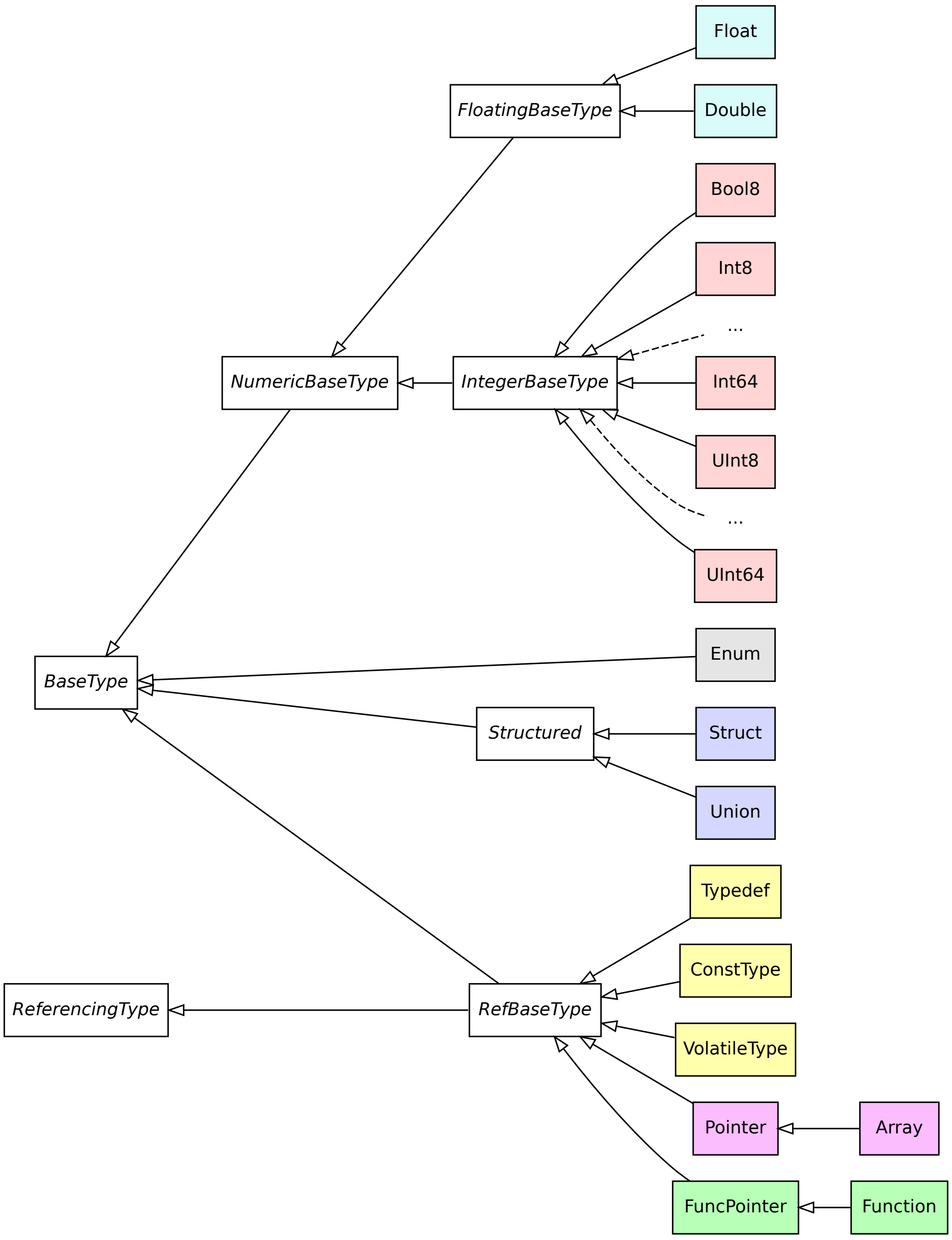


Figure 6.2: Inheritance diagram of the classes that are used to internally represent all data types. The class names in *italic* font represent abstract classes with shared functionality. The color coding indicates similar properties for view generation.

which is a descendant from both *ReferencingType* and *BaseType*, refer to another type on either the syntactic or semantic level.

Example 6.2 Consider the following declarations:

```c
typedef unsigned long long size_t;
const char* s = "foo";
```

The first line represents a syntactic reference of a *Typedef* with name ‘size_t’ to the integer type *UInt64*. In the second line, the type of variable ‘s’ establishes two references: a *Const* type referencing a *Pointer* type, which in turn references the type *Int8*. The former is a syntactic reference while the latter represents a semantic reference. ▲

For *FuncPointer* and *Function* objects, the referenced type is the data type of the function’s return value. In addition, there are other classes that inherit from *ReferencingType* which are not shown in the diagram, namely *StructuredMember* (representing a ‘struct’ or ‘union’ field), *FuncParam* (a function parameter), and *Variable* (a global variable). Each of the corresponding objects holds a reference to the type of their declaration.

6.2.1.2 Parsing Debugging Symbols

The VGC bases its knowledge about the types and global instances of kernel objects on the kernel’s debugging symbols. Before InSight is fit to generate an external view of a guest OS, it has to parse and process the corresponding debugging symbols first. This step takes several minutes to complete. Afterwards, the type information can be stored in a custom format to minimize the effort for future use, so this process is only required once per kernel.

For a Linux guest, the debugging symbols can be obtained by recompiling the kernel with specific compiler flags for symbol generation. This step is straightforward and neither influences the performance nor the semantics of the resulting kernel image. The compiler and linker attach the debugging symbols to the kernel image file in a binary format according to the DWARF standard [Fre05]. InSight utilizes the *objdump* program that is part of the GNU binary utilities to transform the binary symbols into a textual representation. The textual format is much simpler to process than the binary counterpart.

Our prototype is able to correctly read the DWARF debugging symbols and is expected to work well for other OSs that provide their kernel’s debugging information in this format, such as FreeBSD and OpenBSD, with minimal modifications, if any. However, InSight is not limited to the DWARF standard, as it uses its own format to store and load the symbols after they have been parsed. In order to read the symbols for a Windows guest, for example, one only needs to add a parser for Microsoft’s debugging symbols in PDB format [win] to InSight.

6.2.1.3 Consolidation of Type Information

As explained in Section 5.2.2.1, the debugging symbols contain information about all defined types per translation unit, which leads to a high degree of redundancy. When InSight processes the debugging information for the first time, it generates a hash value for each data type. The hash is computed using the information listed in Table 6.1. Based on these hashes, duplicated data types can be quickly recognized and merged into single instances.

Example 6.3 For our experiments, we use a virtual machine with a Debian 6.0 (IA-32) installation. This distribution uses Linux 2.6 as the default kernel (the Debian specific version number is 2.6.32-5). The following table shows the number of types and function prototypes occurring in the debugging symbols of the kernel and all modules along with the number remaining after being merged together:

	No. of symbols		
Symbol	*original*	*merged*	*Reduction*
Data types	10 390 462	73 456	99.3%
Function prototypes	278 373	152 135	45.4%
Total	10 668 835	225 591	97.9%

As the numbers demonstrate, merging the data types to unique representatives reduces their number tremendously. ▲

6.2.2 Memory Access

An OS kernel stores its objects in a virtual address space. As InSight reads these objects from the guest physical memory, it needs a sound understanding of the kernel's address space along with the ability to perform the same translation from virtual to physical addresses that the hardware MMU normally does.

At this time, our prototype is able only to understand and use the kernel address space of Linux guests. As a consequence, we limit our description of how to access the guest's memory to Linux. To add support for a different OS (for instance, Windows), all that InSight requires is a specification of the address ranges that define the kernel's address space, as we explain for Linux below. The actual translation from virtual to physical addresses is the job of the MMU and is independent of the OS using it. In other words, our software-implemented MMU works for any guest when properly used.

6.2.2.1 Linux Kernel Address Space

The Linux kernel does not use a separate virtual address space for its code and data; rather, it reserves a dedicated range within every process's address space for this purpose.

BaseType				
class-specific enumeration value, storage size				
NumericBaseType	*Non-NumericBaseType*			
encoding	type name			
IntegerBaseType	*Enum*	*RefBaseType*		*Structured*
bit size, bit offset	enumeration names and values	hash of referenced type		field names, field offsets, field type hashes
		Array	*FuncPointer*	
		length	parameter names, parameter type hashes	
			Function	
			address of entry point	

Table 6.1: Information used to compute the hashes of data types. The vertical ordering of classes indicates inheritance from bottom to top (cf. Figure 6.2).

That way, a process can invoke kernel functions without having to switch the context first (i. e., via page tables).

The layout of the kernel's address range differs for the Intel IA-32 and AMD64 architecture; however, it includes similar regions in both modes. These layouts are illustrated in Figure 6.3 (a) and (b), respectively. The memory regions are defined by preprocessor macros and global variables that mark their beginning, end, or offset relative to another region. InSight incorporates knowledge about how the kernel uses these memory regions in order to provide access to the objects stored within. We briefly describe both layouts in the following.

32-bit Address Space Layout (IA-32). The lower 3 GB of the 32-bit address space represents the user space and is unique for each process. This area is dynamically mapped using the process's page tables as the process allocates more memory. As a consequence, reading objects in user space requires that the page tables used are from the same process that allocated the objects.

The upper gigabyte of the address space is reserved by the kernel. The beginning of this region is marked by the '`PAGE_OFFSET`' macro. This area starts with a linear mapping of the physical memory, up to a maximum amount of 896 MB. All physical memory beyond 896 MB is mapped dynamically, as explained below. The end of this mapping is defined by the global variable '`high_memory`' and depends on the amount

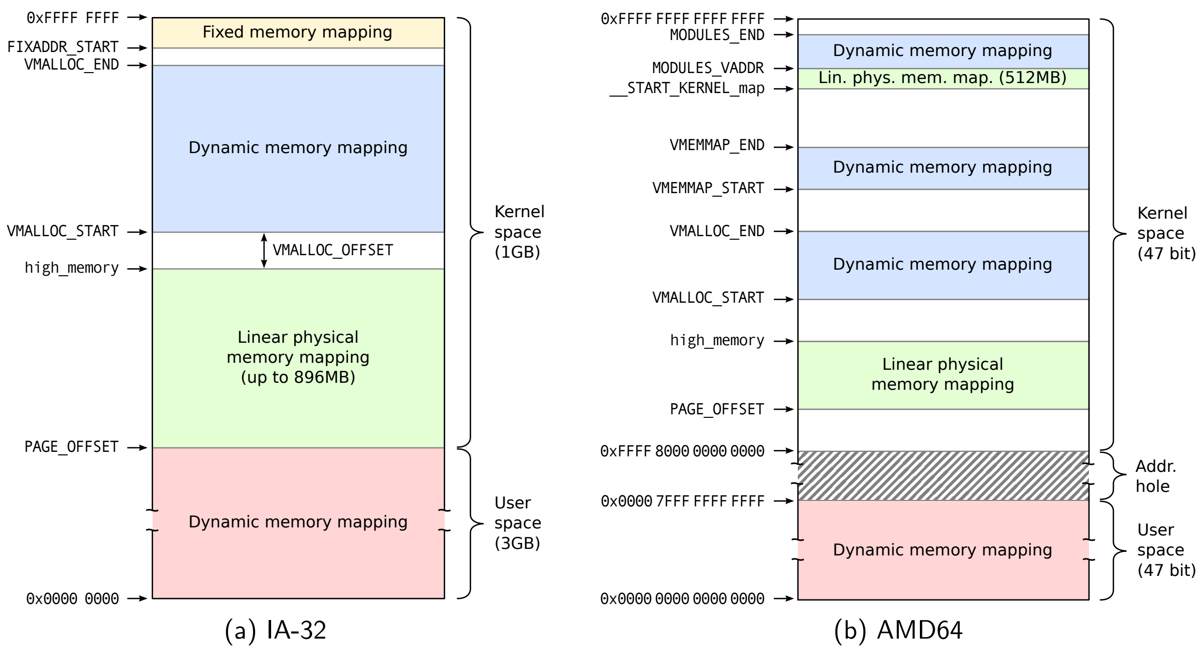


Figure 6.3: Layout of the Linux kernel's virtual address space for IA-32 and AMD64 systems. The red area belongs to user space. Green areas belong to linearly mapped physical memory, blue and yellow areas are mapped on a page granularity, and white regions are unmapped.

of physical memory available to the system. In other words, each virtual address v with 'PAGE_OFFSET' $\leq v <$ 'high_memory' can be translated to a physical address p by calculating $p = v -$ 'PAGE_OFFSET'. This enables us to bootstrap address translation for the dynamically mapped memory areas, as we will see later in this section.

After a guarding hole of size 'VMALLOC_OFFSET', which is intended to catch out-of-bounds memory accesses, the remaining part of the address space is mapped in a non-linear way through the page tables. For the purpose of reading memory, the fixed and dynamic memory mappings shown in Figure 6.3 (a) are treated equally: addresses within these ranges are translated based on the page tables and according to the addressing scheme that the CPU is configured to use (see Section 6.2.2.3).

64-bit Address Space Layout (AMD64). The address space in 64-bit mode is also split into a user space in the lower memory range and a kernel space in the upper range, both of which are 2^{47} bytes in size. The size is constrained by the AMD64 architecture which limits the usable address space to 48 bits: The 48th bit of any virtual address is always extended to bits 49 to 64. That is, the most significant eight bits of an address must all be set to either zero or one. In Figure 6.3 (b), this is indicated as the hatched "addressing hole".

Again, the physical memory of the system is linearly mapped at the address range between '`PAGE_OFFSET`' and '`high_memory`'. Since the address space in 64-bit mode is sufficiently large, this mapping includes all physical memory that the architecture supports, which is up to 64 TB by the time of this writing. In addition, a second linear memory mapping is established starting at the address stored in macro '`__START_KERNEL_map`'. It holds the first 512 MB of physical memory, which includes the entire kernel code segment. For both of these regions, the address translation only requires a simple subtraction of the corresponding virtual offset, as we have discussed previously for the 32-bit case.

As a complement to the linear mappings, the kernel space contains three areas with dynamically mapped memory, each separated from other mappings by large holes of unmapped addresses. The addresses that fall into the dynamically mapped areas can point to any physical memory page and require translation on a per-page basis using the page tables.

6.2.2.2 Obtaining the Address Space Layout

The layout of the kernel's address space varies among different hardware architectures, as described in the previous section, but also between different kernel versions and system configurations. As a consequence, InSight requires the values of the symbolic constants shown in Figure 6.3 for each inspected kernel in order to understand the individual memory layout.

The hardware architecture is read from the kernel configuration, usually found in the file '*.config*' within the kernel's source tree. The architecture tells InSight the length of the virtual addresses and thus the size of pointers (32 bit or 64 bit), how many page table levels are used, and how many entries each table contains.

The '*System.map*' file holds—among other information—the virtual addresses of several kernel symbols which are unknown at compile time. This file is automatically generated after the Linux kernel has been compiled.

The symbolic constants that describe the address space layout are defined in the kernel's source code. Unfortunately, the location of their definition within the source tree has changed several times in the past. In addition, their values are the result of conditional compilation of the source code which depends on Linux's sophisticated build system. To reliably extract this information independent of the kernel version and configuration, we apply the following technique. When initially parsing the debugging symbols for a new kernel version, InSight asks for the path to this kernel's configured source code. It then uses the kernel's build system to compile a "dummy" module for this kernel that stores all the required constants in a local data structure. Note that this module does not need to be loaded into the kernel. Instead, InSight links the intermediate object files that were created during the compilation along with some interstitial code to a small helper program that outputs all the required values. The helper program is then executed and the constants are read from its standard output. InSight stores the obtained values along with the consolidated type information into one file. It performs these steps completely transparently and does not require any user interaction.

Note that the symbol '`high_memory`' actually is a global variable, not a compile-time constant value. It is initialized by the kernel at boot time. InSight reads its value during initialization of a new inspected guest. Since this variable resides within a linearly mapped memory area, no page table based translations are required to read its value.

6.2.2.3 Virtual-to-Physical Address Translation

On x86-based architectures, Linux currently supports the following three addressing schemes:

1. 32-bit virtual address space without PAE
2. 32-bit virtual address space with PAE
3. 64-bit virtual address space

InSight implements translation from virtual to physical addresses for all three schemes. We do not detail the working of the page tables here, but instead refer to the Intel 64 and IA-32 Architectures Software Developer's Manual [Int09]. Since the address space layout varies between the 32-bit and 64-bit mode, as shown in Figure 6.3, different information is required to perform the address translation for each mode.

Table 6.2 shows a complete list of information and symbols that InSight uses for this task, together with the way they are obtained. Strictly speaking, only the symbols without parentheses are essential, since they define the linearly mapped memory areas. All addresses outside of these areas can be translated using the page tables, if they are mapped. However, we use the other symbols to distinguish between unmapped memory and dedicated guarding holes to provide the user with more meaningful error messages.

6.2.2.4 Obtaining the Kernel's Page Table

The translation for each of the three aforementioned addressing schemes differs in the number of page table levels and in the format of their entries, but it always starts at the physical address of the top-level page table, the so-called *page directory*. This is the information InSight requires to bootstrap the page table-based address translation.

The Linux kernel maintains a set of special page tables, the *master kernel page tables*, that always include the accurate mappings of the pages within the kernel space. During process execution, the page tables of all processes are lazily synchronized with the mappings in the master tables using the kernel's page fault handler. Therefore, we choose the master tables as foundation for address translation.

The virtual address of the master kernel page tables is constant for a particular kernel version. It can be obtained from the '*System.map*' file. However, the translation

Information	*32-bit*	*64-bit*	*Acquisition*
Architecture	✓	✓	Kernel configuration
`PAGE_OFFSET`	✓	✓	Kernel build system
`VMALLOC_START`	(✓)	(✓)	Kernel build system
`VMALLOC_END`	(✓)	(✓)	Kernel build system
`VMALLOC_OFFSET`	(✓)	—	Kernel build system
`VMEMMAP_START`	—	(✓)	Kernel build system
`VMEMMAP_END`	—	(✓)	Kernel build system[a]
`MODULES_VADDR`	—	✓	Kernel build system
`MODULES_END`	—	(✓)	Kernel build system
`START_KERNEL_map`	—	✓	Kernel build system
`swapper_pg_dir`	✓	—	*System.map*
`init_level4_pgt`	—	✓	*System.map*
`vmalloc_earlyreserve`	(✓)	—	Physical memory[b]
`high_memory`	✓	✓	Physical memory

[a]May be manually defined
[b]Only for kernels before version 2.6.23

Table 6.2: Information and kernel symbols used for performing virtual to physical address translation. The symbols with checkmarks in parentheses are actually not required, but they help to rule out invalid addresses faster, especially for the 64-bit systems with four levels of page tables.

requires the corresponding physical address. As it turns out, the master page tables are located in a linearly mapped memory area of the kernel space. Consequently, its physical address can be obtained by subtracting a fixed offset from its virtual address. With this information, InSight is ready to access the page tables and to translate virtual addresses in dynamically mapped memory areas.

6.2.3 Reading Kernel Objects

Each kernel object that is read from memory is internally represented as an object of class *Instance*. An instance basically holds the type as well as the virtual address of the kernel object and provides methods to inspect the type's properties—for example, the declared fields of '`struct`' and '`union`' types. The fields can then be accessed as further instances with data types and addresses corresponding to the fields' declaration and offsets within the embedding object.

Instances that represent a numeric type allow access to the individual number they currently hold: InSight reads the memory occupied by the instance and interprets the value according to the data type, for example, as an unsigned integer value having a width of 32 bits. An instance of a pointer type can be dereferenced, yielding the kernel object that it points to. Arrays allow access to their fields in the same manner.

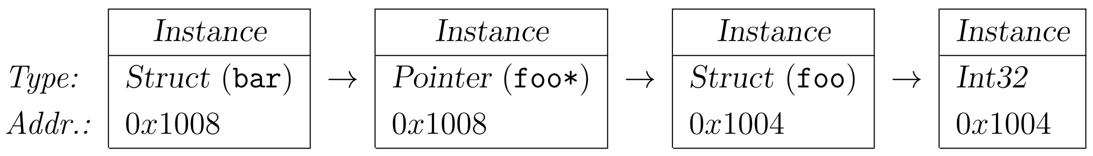
Example 6.4 Consider the following code fragment:

```c
struct foo {
    int value;
};

struct bar {
    struct foo* f;
};

struct foo sf = { 123 };
struct bar sb = { &sf };
```

This code might result in the following memory layout within the guest:

`0x1000`	···	
`0x1004`	`123`	← field ‘`value`’ of variable ‘`sf`’
`0x1008`	`0x1004`	← field ‘`f`’ of variable ‘`sb`’
`0x100C`	···	

Suppose InSight is in possession of the corresponding type information so that it can introspect the guest's memory. We first request an instance of variable ‘`sb`’, then follow its field ‘`f`’, dereference the pointer to retrieve the ‘`struct foo`’ object pointed to by ‘`f`’, and finally access its integer field ‘`value`’. This access pattern will result in the following four *Instance* objects within our framework:



▲

6.2.3.1 Heuristics for Linked Lists

With regard to linked lists and hash tables that are implemented as detailed in Section 5.2.3, we have already argued that the information contained in the debugging symbols is insufficient to understand the involved pointer and type manipulations automatically. Unfortunately, these data structures are ubiquitous in the kernel. In Chapter 7, we extend InSight to reliably detect such dynamic behavior on the source code level. For the time being, we use the following heuristic to provide a semi-automatic way to iterate over lists of kernel objects.

Recall that a data structure can be arranged in a doubly linked list by including a “node” field of type ‘`struct list_head`’ in the structure’s declaration. To organize objects in a hash table, the field must be of type ‘`struct hlist_node`’. When InSight parses the debugging symbols, it detects these two special nodes and marks the particular fields of the embedding structure accordingly. These nodes are assumed to resemble a list of objects of the embedding type and point to the same field within these objects, as shown in Figure 6.4. Whenever the ‘`next`’ or ‘`prev`’ pointer of a marked node structure is dereferenced, InSight changes its type to a pointer of the embedding structure and corrects the address by the field’s offset.

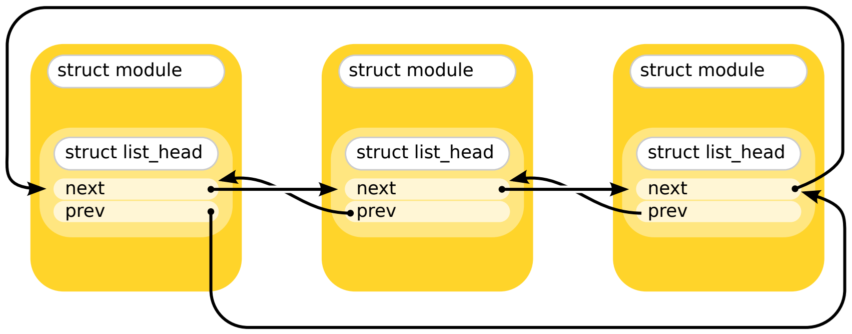


Figure 6.4: The simple heuristic assumes that fields of type ‘`struct list_head`’ are used to arrange objects of the same type into a linked list with the same offset.

Example 6.5 Consider the listing in Example 5.2 on page 56 once again. The declaration of ‘`struct module`’ includes a field ‘`list`’ of type ‘`struct list_head`’. InSight detects the node character of type ‘`list_head`’ embedded within the ‘`module`’ structure and marks field ‘`list`’ accordingly.

Now let ‘`m`’ be a variable of type ‘`struct module`’. If the user accesses ‘`m.list.next`’, he will again receive an object of type ‘`struct module`’ whose address *addr* has been corrected by the offset of ‘`list`’ within ‘`module`’:

$$addr = \text{‘}\texttt{m.list.next}\text{’} - \text{offsetof}(\text{‘}\texttt{struct module}\text{’}, \text{‘}\texttt{list}\text{’})$$

This corresponds to the same operation that is performed by the macros for list iteration.
▲

The described heuristic is correct if the assumptions made about the usage of a ‘`list_head`’ field hold. It is left up to the user to know if this is the case for a particular node field within a data structure. This approach works well for enumerating all processes in the guest starting at the global variable ‘`init_task`’, for example.

In situations as described in Section 5.2.3.3 in which at least one of the assumptions does not hold, the flexible design of InSight allows the user to address this manually by choosing the correct type and/or offset for a node's pointer. Using the type rule engine of InSight, which is introduced in Section 7.3, this expert knowledge can be encapsulated in libraries that are reusable for all applications.

6.2.4 Command Line Interface

InSight provides a command line interface in form of a shell for interactive analysis of a VM's state, allowing inspection of the kernel's data structures and global variables. In addition, it can be used to load and unload images of the guest's physical memory. The image can either be a physical memory snapshot stored in a regular file on disk, a special file (such as a memory-mapped file), or a device node (such as the '*/dev/mem*' device in Linux).

For each loaded memory file, the kernel objects represented by the global variables as well as their fields can be accessed using the common notation '`object.field`'. In addition, the user may choose to read an object from an arbitrary virtual or physical address as any known data type, again allowing the user to further follow any fields of that type. This functionality is similar to that of a debugger, but with the following key differences:

1. A debugger typically inspects a running process at its current point within the control flow — that is, its instruction pointer. It can analyze the stack frames and show the local symbols according to their execution context. In contrast, InSight does not have this information. The purpose of our framework is to analyze the overall system state, not the volatile context of the current execution. For our considerations, the kernel objects that are read from memory are contextless: All state information that permanently influences the kernel's behavior *must be* rooted in some location that is globally accessible, such as a global variable or a hardware register.

   The only exceptions to this rule are objects that lie in user space: The page table belonging to the process that allocated this object represents the object's context. To provide access to user-space objects, InSight allows switching to different page tables that an application may specify.

2. Since InSight relies on the global variables alone, it provides mechanisms to automatically identify as many objects in memory as possible and to construct a graph of kernel objects, as illustrated in Figure 5.1 on page 53. This graph can be traversed and provides the answers to many questions that a debugger cannot tell—for example, how many objects point to a particular object, which object resides at a particular address, and so on.

3. A debugger reads the virtual memory of a process and relies on the operating system and hardware to transparently map accesses to physical memory. As InSight cannot rely on any guest OS support and does not limit itself to a particular hypervisor, it operates directly on physical memory and performs the address translation on its own.

The interactive command line interface is very helpful for understanding the kernel memory and the relationships between objects. Once the user has gained a deeper understanding of the objects that are relevant for a specific application, he may leverage the full power of InSight by writing code for the JavaScript engine to automate repeating VMI tasks.

6.2.5 Scripting Engine

Our VMI framework includes a scripting engine that allows users to write JavaScript code for interacting with kernel objects read from memory and to perform complex analysis tasks. The user can access global variables by requesting an *Instance* object of a variable by using its name or its internal ID. If the resulting object represents a structure, its fields can be accessed as object properties by their names using the dot operator.

Example 6.6 The following code fragment demonstrates the basic interaction with a kernel object using JavaScript code:

```
var swpr = new Instance("init_task");
println(swpr.comm); // output: "swapper"
```

In the first line, an *Instance* of the global variable '`init_task`' is created. This variable is of type '`struct task_struct`' and represents the parent process of all processes in a Linux system, the so-called "swapper" process. The process descriptor '`task_struct`' has a field '`char comm[16]`' which holds the name of the process. In the second line, this field is accessed as a property of the JavaScript object '`swpr`' and its value is printed to the screen. ▲

When a field of a kernel object having a pointer type is accessed, all pointers that point to accessible memory are automatically dereferenced, unless specified otherwise. As a result, the user usually receives a new data structure, a numeric type or a function pointer when a structure field is accessed as an *Instance* property.

In addition to field accesses, an *Instance* object provides function properties to access meta information of the corresponding kernel object, such as the object type, fields, storage size, and address. Part of this information can also be altered. For example, the type of the instance can be changed or its address can be manipulated.

Example 6.7 The following listing shows a more complex example script that prints the list of all loaded modules for a Linux guest:

```
 1  var head = new Instance("modules");
 2  var m = head.next;
 3  m.ChangeType("module");             // change to actual type
 4  var offset = m.MemberOffset("list");
 5  m.AddToAddress(-offset);            // fix the address
 6  head.AddToAddress(-offset);         // for loop termination
 7
 8  do {
 9      println(m.name, m.args);
10      m = m.list.next;                // list_head heuristics
11  } while (m && m.Address() != head.Address());
```

Note that the *Instance* object that we retrieve from the global variable '`modules`' in line 1 is of the type '`list_head`'. This scenario corresponds to the listing in Example 5.2 on page 56 where a global variable of type '`list_head`' represents the head of a list of '`struct module`' objects. The flexibility of InSight allows us to change the object's type to structure '`module`' in line 3 and correct its address in line 5 according to the usage pattern of doubly linked lists. Once this is accomplished, the iteration over all modules in the loop becomes straightforward and does not require any more type or address corrections. In line 10, InSight applies the node heuristics to calculate the correct address and infer the real object type automatically, thus hiding the any further "pointer magic" of the lists from the user. ▲

The great advantage of the scripting engine over any low-level approach is the fact that any sort of runtime error will not lead to a segmentation fault or similar failures. Such errors lead instead to JavaScript exceptions that may be handled in the script being executed. In any case, the errors are contained within the scripting engine and do not propagate to the framework itself. As a script's execution is triggered from the command line interface, the user can modify the script and re-run it directly without having to recompile the source code, restart InSight, or restart the monitored VM. This is a major advantage of InSight over other view-generating approaches and greatly eases the development of new VMI mechanisms.

6.3 Application

The current implementation of InSight already allows one to perform various types of analysis and is deployed in several projects for VMI-based intrusion detection within our research group. We describe the view generation and the application of the collected data in the following paragraphs.

6.3.1 Periodic Analysis

One way of monitoring a system for intrusions is to perform an analysis of the guest in regular intervals and report any suspicious findings. If the interval is chosen to be sufficiently long and the analysis can be performed quickly, this method typically incurs only a small overhead.

An approach for VMI-based intrusion detection that has been described by various authors [GR03, LLCL08, MFPC10] is called *lie detection*. It works by generating two different views simultaneously, one inside of the guest OS and one with the help of VMI techniques. These views are then compared with each other and any discrepancy between them is taken as an indicator for malicious activity. One of our projects implements a lie detector for running processes and loaded kernel modules on a live guest to reveal rootkits that are trying to hide their presence [Kit10].

In a different project, we experimented with new ways of detecting intrusions on the kernel level. For this purpose we created a series of sequential memory dumps of the guest while inducing normal and malicious activities to perform offline analysis of the collected states. Here InSight was used to gather information about running processes, find the location of the code sections of loaded kernel modules to detect kernel code patching, and reveal changes to the system call table, among other things [Vog10]. This experimental work showed that InSight is also very much capable of performing forensic analysis.

6.3.2 Event-driven Analysis

An alternative method for system analysis monitors and reacts to system events. In such cases, a VMI-based approach manipulates the system in such a way that the events of interest cause a trap to the hypervisor [PSE10]. Before the hypervisor returns control to the guest, the VGC collects the required data from the current state of the guest. Depending on the frequency of such events, it is crucial to avoid any unnecessary overhead for the analysis in order to minimize overall performance degradation.

As an example for such an approach, we have combined InSight with Nitro, a VMI-based framework for system call tracing [PSE11]. Nitro uses InSight for view generation to augment the low-level parameters of the monitored system calls. This combination of tools allows us to produce a much richer output of system call information compared to plain numbers and pointer values. For example, InSight is able to determine the concrete type of a process's file handle in calls to the '`read`' and '`write`' system calls and print the corresponding data [Die11]. This data may consist of the IP addresses of a connected socket or the file name of a regular file.

Figure 6.5 shows a sample output for tracing the '*wget*' binary when performing an HTTP request. Each line starting with a time stamp lists one recorded system call with its name followed by the expected parameters. The next lines contain the values of the actual parameters passed to the system call in their context specific semantic

```
Jun 20 17:58:20: sys_write: unsigned int fd, const char __user *buf, size_t count
    fd: unsigned int: 0x2 -> "tty1"
    buf: const char __user *: 0x7FFF702FF080 ->
buffer content hex (of size 46):
43 6f 6e 6e 65 63 74 69 6e 67 20 74 6f 20 67 6f 6f 67 6c 65 2e 64 65
7c 32 30 39 2e 38 35 2e 31 34 38 2e 31 30 33 7c 3a 38 30 2e 2e 2e 20
buffer content string: Connecting to google.de|209.85.148.103|:80...
    count: size_t: 0x2E

Jun 20 17:58:20: sys_socket: int domain, int type, int protocol
    domain: int: 0x2 (PF_INET )
    type: int: 0x1 (SOCK_STREAM )
    protocol: int: 0x0 ()

Jun 20 17:58:20: sys_connect: int sockfd, struct sockaddr __user *addr, int addrlen
    sockfd: int: 0x3 -> (socket) -> [...] type: (SOCK_STREAM ) flags: ()
    addr: struct sockaddr __user *: 0x7FFF702FF2D0 ->
            -> (assuming sockaddr_in)
           0.  0x00  sin_family : sa_family_t   = 2
           1.  0x02  sin_port   : __be16        = 20480
           2.  0x04  sin_addr   : struct in_addr = ...
           3.  0x08  __pad      : unsigned char[8] = (0, 0, 0, 0, 0, 0, 0, 0)
                       dereferencing: sin_addr 1737774545 (209.85.148.103)
    addrlen: int: 0x10

Jun 20 17:58:20: sys_write: unsigned int fd, const char __user *buf, size_t count
    fd: unsigned int: 0x3 -> (socket) -> [...]: (SOCK_STREAM ) flags: ()
    buf: const char __user *: 0x7FFF702FF320 ->
buffer content hex (of size 107):
47 45 54 20 2f 20 48 54 54 50 2f 31 2e 30 0d 0a 55 73 65 72 2d 41 67
65 6e 74 3a 20 57 67 65 74 2f 31 2e 31 32 20 28 6c 69 6e 75 78 2d 67
6e 75 29 0d 0a 41 63 63 65 70 74 3a 20 2a 2f 2a 0d 0a 48 6f 73 74 3a
20 67 6f 6f 67 6c 65 2e 64 65 0d 0a 43 6f 6e 6e 65 63 74 69 6f 6e 3a
20 4b 65 65 70 2d 41 6c 69 76 65 0d 0a 0d 0a
buffer content string: GET / HTTP/1.0
User-Agent: Wget/1.12 (linux-gnu)
Accept: */*
Host: google.de
Connection: Keep-Alive
    count: size_t: 0x6B
```

Figure 6.5: System call information resulting from a HTTP request of the '*wget*' program to google.de, collected with Nitro [PSE11] and augmented using InSight. The system call parameters can be resolved to their semantics within the guest (taken from [Die11]).

interpretation. Consider the lines 1 to 8, for instance, that detail a '`write`' system call. The file descriptor '`fd`' is the one being written to and can be resolved to a file named '`tty1`' in line 2; obviously, the monitored process is writing the first console, which is attached to the device node '*/dev/tty1*' in UNIX systems. The buffer that gets written resides in user-space but can be dereferenced nevertheless. The lines 4 to 7 show its contents in hexadecimal byte values and as an ASCII string. As can be seen in line 7, the '*wget*' binary writes a message to the console in order to inform the user that it is about to connect to host `google.de` at IP address 209.85.148.103 on port 80.

Other system calls are augmented similarly. In order to make data objects in user space accessible, the page table is switched to the current process's table during view generation. This project has also shown that the separation of InSight into a back-end and a front-end helps to minimize the per-analysis overhead. The parameter augmentation contributes only to a small degree to the total overhead of the complete framework.

6.4 Summary

We have identified view generation as a unique and worthwhile challenge for virtual machine introspection. With our research, we strive to provide an external VGC that combines hardware and software knowledge for full state applicability. The result of our efforts is InSight, a VMI framework that focuses on bridging the semantic gap independent of the application at hand. Our tool creates a complete view of the guest kernel, is decoupled from the introspection task itself, and remains flexible enough to be applicable for digital forensics, intrusion detection, malware analysis, and kernel debugging.

Our implementation required addressing several engineering challenges when analyzing a system as complex as the Linux kernel. The solutions have been incorporated in InSight which has resulted in an extremely powerful and flexible VMI framework that offers several interfaces including a scripting engine for further automation of complex tasks.

Finally, we offered evidence of InSight's power and flexibility by outlining several successful applications of InSight within our research group. We share our framework as an open source tool [ins] with other researchers to enable the fast and intuitive development of new VMI and forensic approaches.

Chapter 7

Extracting Semantic Knowledge through Static Code Analysis

In the previous chapter, we introduced or VMI framework, InSight. So far, we have described functionality that is based entirely on knowledge about the types as they are *declared* in the source code. The only exception to that rule is the heuristics for handling doubly-linked lists and hash tables, which we described in Section 6.2.3.

However, we already demonstrated the limitations of an approach that is based only on static type information in Section 5.2.3 and 5.2.4: The manifold instances of dynamic pointer and type manipulations as well as runtime dependencies which can be found in the source code of the Linux kernel—and, without a doubt, in any other OS kernel with comparable complexity—render this knowledge incomplete and sometimes even misleading. It is this incompleteness that demands a significant amount of domain knowledge from users which require access to kernel objects in the deepest corners of the system's memory.

Our framework allows users to incorporate their expert knowledge on usages of certain data types into their VMI program code. With this supplemental information, InSight already supports some interesting applications, such as the examples shown in Section 6.3. Considering InSight in the terminology of our VMI model as illustrated in Figure 6.1 on page 68, one could rightfully argue that such an approach basically applies part of the semantic knowledge μ within the application-specific aggregation or classification components $[\![a]\!]$ and $[\![d]\!]$, rather than the VGC $[\![g_{\lambda,\mu}]\!]$ where it belongs. Since we strive towards full automation of our methods and reducing the required domain knowledge, it is our goal to constraint the semantic knowledge to the VGC as much as possible.

The heuristics for linked lists that we originally integrated into InSight were a first attempt to cope with the *type usage* of pointers that differs from their declaration. The heuristics make certain assumptions about the general usage of '`list_head`' and '`hlist_node`' fields embedded in data structures. This approach works well for many instances of linked lists for which the assumptions hold, but inevitably leads to false kernel objects when the assumptions are wrong. Consequently, it does not solve the problem from the user's perspective, since he now has to know for which data structures and fields the list assumptions hold.

For a generic solution to this problem, the following observation is important: Even though the symbol information only describes the declared types and variables, all dynamic type usages are evident within the source code. In other words, if we can identify type usages within the source code and extract their semantics, we can augment the static type information and effectively re-create this dynamic behavior within our VGC.

In this chapter, we describe a novel approach for identifying dynamic pointer manipulation in full C source code. We call this method the *used-as* analysis. In contrast to a points-to analysis [And94], it focuses on type usages instead of pointer locations. The used-as analysis reveals usages of structure or union fields and global variables in type contexts that differ from their declared types. In addition, it allows one to extract the pointer arithmetic that is performed to transform a pointer value to a target address. Needless to say, our approach requires the source code of the OS in order to analyze it. With full access to the complete code of the kernel and all core components, our analysis technique can be applied to any operating system that is written in C.

We have implemented this analysis technique as an extension to InSight to further improve its accuracy and coverage when reading kernel objects from memory. In applying the used-as analysis to the kernel sources, our VGC establishes extended type relations and captures pointer arithmetic that the kernel would perform at runtime. This information supplements the kernel debugging symbols InSight has been using so far, thus extending the semantic knowledge μ of the software architecture. With this extension, InSight is now able to automate the retrieval of objects referenced by differently typed pointers, pointers stored as integer types, and generic pointers such as '`void*`'. In addition, InSight performs arithmetic operations on the pointer value to transform a source value to a target address, for example, by adding a type specific offset or applying a bit mask with bitwise logical operations.

Chapter overview. This chapter describes our static code analysis and its integration into InSight. First, we propose the used-as analysis for full C code in Section 7.1, which is essentially a type-centric extension of Andersen's points-to analysis. After establishing some symbols and notations, we describe the two steps of our analysis: the points-to analysis and the used-as analysis. In Section 7.2, we describe the implementation of the aforementioned analysis as an extension to our existing tool, InSight. Section 7.3 introduces the type rule engine, which serves as a management interface for the extended type relations and allows the user to extend the type knowledge even further with his own expertise. The application of our extension in Section 7.4 illustrates the elegance and simplicity of the resulting view generation process. We complete this chapter with an overview of related work in the field of pointer analysis in Section 7.5, and summarize our contributions in Section 7.6.

7.1 Static Code Analysis

In static code analysis, the *points-to* analysis, according to Andersen [And94], tries to answer the question, "which set of locations might a particular variable point to?" This method considers assignment statements and function invocations to keep track of the memory locations a pointer variable might hold. This kind of analysis tends to be tricky for the C programming language because of the constructs that this language supports, such as multi-level pointers, the address operator '`&`', and function pointers.

Since pointers are tied to variables which only exist within a certain scope (which may also be the whole program in case of global variables), this type of analysis is rather data-flow driven and a points-to set is only valid within a pointer's scope. A variant of this method [HT01] has already been applied to identify kernel objects of the Windows kernel using an inter-procedural points-to analysis [CCL+09].

Recalling the motivation for our work from Section 5.2.3, the goal of our analysis is to identify dynamic pointer manipulations that are performed either for fields of data structures or for global variables. Suppose we construct the graph of kernel objects starting from all global variables and find an object of a certain type by following a pointer field from another object. This newly discovered object is not bound to any control flow context or scope, such as a function or lexical block. For the purpose of our analysis, it resides in the global scope whether it was originally allocated on a heap, on a stack, or originated from a static data segment. So when we access a field of an object of a certain type, two questions arise:

1. Is this type's field *used as* a pointer to an object of a type that differs from its declaration?
2. How do we need to transform the value of this field in order to retrieve the object's address?

It is exactly these two questions that our used-as analysis will answer.

7.1.1 Used-As vs. Points-To Analysis

The approach of Carbone et al. [CCL+09] is based on the assumption that the actual data type stored in a pointer location is used at least once in some assignment statement within the inter-procedural control flow of the pointer variable. As we have explained before, the objects read from memory do not have a particular context or scope. As a consequence, all possible program points using this pointer type must be considered when such a pointer is encountered. We exploit this observation as follows.

We use a method similar to a field-sensitive flow-insensitive points-to analysis. The key difference is that our analysis establishes used-as relations between structure or union fields as well as global variables on the one hand, and data types on the other. That is, it identifies all types that a pointer or an integer type is used as. In addition, it records all static pointer arithmetic that is applied in such situations to retrieve the target address from a source pointer. These related types together with their pointer arithmetic represent the candidate types that have to be considered when following the field of an object or reading a global variable from memory. Such an approach contrasts with the conservative pointer treatment of a static type system, for example, as performed by a compiler. However, completeness of the generated view is crucial during introspection. Consequently, we find it justified to over-approximate the view and consider all possible interpretations of a pointer at the cost of possible ambiguities.

Our approach has several advantages over traditional points-to analysis approaches. First, it only requires an intra- instead of an inter-procedural analysis, leading to reduced complexity. This is due to the fact that our analysis needs to find a type usage only once per type, not once per pointer variable.

Second, our analysis works on the abstract syntax tree (AST) of full (preprocessed) C code—including many GCC extensions—and does not perform any preliminary transformation steps. In contrast, many points-to analysis algorithms require the source code to be transformed into simplified statements [ALSU07], such as three-address code or single-static assignment [And94, Ste96, Das00, HT01, PKH07], or they work on some compiler-specific medium level representation of the code [CCL+09].

Third, the used-as analysis detects type usages not only in assignment statements, but in all other expressions that might change a pointer's type, such as type casts or return statements. The following example illustrates the importance of this difference for our goal.

Example 7.1 Consider the following code fragment:

```
1  struct A { /* ... */ };
2  struct B { void *data; };
3
4  struct A *pa = malloc(sizeof(struct A));
5  void *p = malloc(sizeof(struct B));
6  ((struct B*)p)->data = pa;
```

When this code is transformed into simplified statements, line 6 would introduce one or more temporary variables. The result of the transformation could look as follows:

```
6  /* was: ((struct B*)p)->data = pa; */
7  struct B *tmp1 = (struct B*)p;
8  tmp1->data = pa;
```

Since traditional points-to analyses are variable-specific rather than type-specific, there are basically two possible outcomes for a points-to analysis of line 6 or of the simplified statements in lines 7 and 8:

- If the points-to analysis is field-insensitive, the analysis considers all assignments to fields as assignments to the variable itself. Thus, it would conclude that both '`tmp1`' and '`p`' point to the '`struct A`' object '`pa`', which is clearly not the case.

- If the points-to analysis is field-sensitive, the analysis would catch the type usage in the assignment for the introduced temporary variable ‘`tmp1`’ of type ‘`struct B*`’ but would not propagate it to variable ‘`p`’, since ‘`p`’ is of a different type and not even a ‘`struct`’ or a ‘`union`’.

In contrast, the used-as analysis correctly handles such situations and recognizes both runtime type usages in the original as well as in the simplified code: first, that variable ‘`p`’ is cast to type ‘`struct B*`’, and second, that field ‘`data`’ of type ‘`struct A`’ is assigned a ‘`struct A*`’ object. Note that while the first usage is bound to variable ‘`p`’, the second usage is bound to type ‘`struct B`’ and will be considered any time this field of an object of this particular type is accessed. ▲

Finally, most points-to analysis algorithms ignore pointer arithmetic — that is, the calculation of a variable’s address using an arithmetic expression which might involve the address, value, or pointed-to address of another variable. The only exception here is the KOP system [CCL+09], which supports adding a type-specific offset to a pointer location, but nothing more complex. Our approach overcomes this limitation and supports full C arithmetic expressions to transform a source value into a target address. In fact, our analysis has revealed usages of all sorts of pointer arithmetic in the Linux source code, including modulo division, bitwise logical operations (i. e., application of bit masks), and bit shifts.

7.1.2 Notation and Symbols

As already mentioned in the previous section, our analysis works on the AST of full C source code. This leads to a more complicated formal notation of the deduction rules of our algorithm. In this section, we introduce the notation and symbols we are using for the description.

7.1.2.1 Symbol Transformations

Our algorithm starts from symbols representing variables and analyzes the expression and the type context in which they are used by traversing the AST. In its context, a variable may undergo several *transformations* within the expression, such as dereferencing or a field access. We denote a list of transformations for a symbol with $T = \tau_1, \tau_2, \ldots, \tau_k$ and use ε to represent an empty list. To express that variable x is transformed by list T, we write $\langle x, T \rangle$, where the transformations τ_i in T are applied from 1 to k. A complete overview of all symbol transformations is given in Table 7.1.

Example 7.2 Consider the following code fragment:

```
(*f());
return &x[2]->y.z;
```

Description	*Example*	*Notation*
Field access	`x.y`	$\langle x, .y \rangle$
Array access	`x[3]`	$\langle x, [3] \rangle$
Address operator	`&x`	$\langle x, \& \rangle$
Dereferencing	`*x`	$\langle x, * \rangle$
Arrow operator	`x->y`	$\langle x, *, .y \rangle$
Function invocation	`f()`	$\langle f, ↳ \rangle$
Function return	`return x`	$\langle x, ↱ \rangle$

Table 7.1: Symbol transformations and their corresponding formal notation.

The first line results in the transformation $\langle f, ↳, * \rangle$ while the second line is expressed as $\langle x, [2], *, .y, .z, \&, ↱ \rangle$. ▲

7.1.2.2 Expressions

In order to handle runtime pointer arithmetic correctly, it is important to consider full C expressions. We abbreviate an arbitrary legal C expression involving only variable '`x`' and compile-time constant values as $e(x)$. Note that this expression might contain symbol transformations to x. This is fine as long as the transformations are irrelevant and helps to abbreviate the notation. In particular, if $T \neq \varepsilon$, we can construct e' from e such that $e(x) = e'(\langle x, T \rangle)$. In that case, $e(x)$ is just a more concise representation of $e'(\langle x, T \rangle)$. Thus, we prefer the former notation unless the actual transformations T are important in a particular context.

The following example illustrates the relation between expressions and transformations.

Example 7.3 Given the C expression '`(x->y + 3) * 4`', we can see this expression as $e_1(v) = (v+3) \cdot 4$. Thus, we can denote it with $e_1(\langle x, *, .y \rangle)$. However, the same code can also be written as expression $e_2(v) = (\langle v, .y \rangle + 3) \cdot 4$, resulting in the notation $e_2(\langle x, * \rangle)$. Finally, the expression can also be transformed to $e_3(v) = (\langle v, *, .y \rangle + 3) \cdot 4$, so we can write $e_3(x)$. ▲

7.1.3 Step 1: Points-To Analysis

The used-as analysis is type-centric rather than variable-centric. As a consequence, the used-as relations are only relevant for two types of symbols: global variables of types that allow them to hold a pointer value and structure fields of these types. However, such symbols are often assigned to local variables before the actually interesting type usage occurs. Thus, we first perform a field-sensitive, intra-procedural, and control flow insensitive points-to analysis of the source code. This allows us to trace the type usage to the variables and types of interest across multiple statements. We use the inference rules listed in Figure 7.1 to derive the complete points-to map as explained in the remainder of this section.

7.1.3.1 Initialization

We examine all assignment and return statements in the source code and initialize the points-to map according to the ASSIGN and RETURN rules. Note that for the left-hand side of an assignment (i. e., an *lvalue*[1]), the ASSIGN rule records all transformations that are applied to this symbol in a given statement. Keeping track of the transformations is important for three reasons:

1. By considering the address and star operators, we can distinguish between the values of '`&x`', '`x`', and '`*x`'. They also enable us to resolve indirect assignments through pointer variables, such as in '`x = &y; *x = z;`'.

2. Through the field and array access transformations, we achieve field sensitivity for structures and arrays.

3. The pointer analysis is intra-procedural in nature; that is, it does not consider the pointer assignments as they flow into function parameters and back through return statements. However, we keep track of global variables that are used within functions and are returned to the caller by means of the function call and function return transformations.

Each $\langle x, T\rangle$ actually maps to a *set* of possible locations. For the sake of readability of the notation, we only write $\langle x, T\rangle \mapsto e(y)$ to express that $e(y)$ belongs to the points-to set of $\langle x, T\rangle$ (instead of using the less concise notation $\langle x, T\rangle \supseteq \{e(y)\}$).

The following example demonstrates the application of the ASSIGN and RETURN rules during the initialization:

Example 7.4 Given the code fragment below:

```
1  int *x, y[4], *z;
2  int f() { return *x; }
3  y[1] = z + 2;
4  x = y[1] * 3;
```

The initialization for this code results in the following points-to set:

(1)	RETURN, line 2:	$\langle f, \varepsilon\rangle \mapsto \langle x, *, \Rsh\rangle$
(2)	ASSIGN, line 3:	$\langle y, [1]\rangle \mapsto \langle z, \varepsilon\rangle + 2$
(3)	ASSIGN, line 4:	$\langle x, \varepsilon\rangle \mapsto \langle y, [1]\rangle \cdot 3$

▲

[1]An "lvalue" is a value that has an address and can be assigned to.

$$(\textsc{assign}) \qquad \langle \texttt{x}, T\rangle \ \texttt{=}\ e(\texttt{y}) \models \langle x, T\rangle \mapsto e(y)$$

$$(\textsc{return}) \qquad \mathit{type}\ \texttt{f() \{ return } e(\texttt{x})\texttt{; \}} \models \langle f, \varepsilon\rangle \mapsto \langle e(x), \Rsh\rangle$$

$$(\textsc{trans}) \qquad \frac{\langle x, T_x\rangle \mapsto e_y(\langle y, T_y\rangle) \quad \langle y, T_y\rangle \mapsto e_z(z)}{\langle x, T_x\rangle \mapsto e_y(e_z(z))}$$

$$(\textsc{star-1}) \qquad \frac{\langle x, T_x\rangle \mapsto e_y(\langle y, T_y, \&\rangle) \quad \langle x, T_x, *\rangle \mapsto e_z(z)}{\langle y, T_y\rangle \mapsto e_z(z)}$$

$$(\textsc{star-2}) \qquad \frac{\langle x, T_x\rangle \mapsto e_z(\langle z, T_z, \&\rangle) \quad \langle y, T_y\rangle \mapsto e_x(\langle x, T_x, *\rangle)}{\langle y, T_y\rangle \mapsto e_z(\langle z, T_z\rangle)}$$

$$(\textsc{call-func}) \qquad \frac{\langle x, T_x\rangle \mapsto e_f(\langle f, \text{↳}\rangle) \quad \langle f, \varepsilon\rangle \mapsto \langle e_y(y), \Rsh\rangle}{\langle x, T_x\rangle \mapsto e_f(e_y(y))}$$

$$(\textsc{call-ptr}) \qquad \frac{\langle x, T_x\rangle \mapsto \langle y, T_y, \text{↳}\rangle \quad \langle y, T_y\rangle \mapsto \langle f, \&\rangle}{\langle x, T_x\rangle \mapsto \langle f, \text{↳}\rangle}$$

Figure 7.1: Inference rules for the pointer analysis that is performed as part of the used-as analysis. The operator $\models$ describes relations that are derived directly from source code statements.

7.1.3.2 Transitivity

After the initialization, the TRANS rule allows us to derive transitive points-to relations. In order to retain field sensitivity, the rule requires that the same transformations T_y are applied to y on the left and right-hand side.

Example 7.5 We continue Example 7.4 and apply the transitivity rule to the previous points-to set:

(4) TRANS, (2), (3): $\langle x, \varepsilon \rangle \mapsto (\langle z, \varepsilon \rangle + 2) \cdot 3$ ▲

Note that for inferring transitive relations, we required that $x \neq y \vee T_x \neq T_y$. That is, we disallow recursive expressions. For this reason, we only consider regular assignments during initialization and ignore all arithmetic assignments, such as '`x += y`', as they are inherently recursive. For the used-as analysis step, we only need to encounter each type usage once per type. Consequently, the non-recursive nature of the points-to analysis step has no effect on the resulting used-as relations.

7.1.3.3 Indirect Assignments

The rules STAR-1 and STAR-2 handle indirect assignments to and from pointer variables, respectively. The next example illustrates their usage:

Example 7.6 Consider the following three assignments:

```
1  p = &x;
2  *p = y;
3  z = *p;
```

The rules STAR-1 and STAR-2 handle the indirect assignments in line 2 and 3. To derive that z points to y, we simply apply the TRANS rule again:

(1)	ASSIGN, line 1:	$\langle p, \varepsilon \rangle \mapsto \langle x, \& \rangle$
(2)	ASSIGN, line 2:	$\langle p, * \rangle \mapsto \langle y, \varepsilon \rangle$
(3)	ASSIGN, line 3:	$\langle z, \varepsilon \rangle \mapsto \langle p, * \rangle$
(4)	STAR-1, (1), (2):	$\langle x, \varepsilon \rangle \mapsto \langle y, \varepsilon \rangle$
(5)	STAR-2, (1), (3):	$\langle z, \varepsilon \rangle \mapsto \langle x, \varepsilon \rangle$
(6)	TRANS, (2), (3):	$\langle z, \varepsilon \rangle \mapsto \langle y, \varepsilon \rangle$ ▲

7.1.3.4 Function Invocations

Functions can be invoked in two ways: through the function name, or through a function pointer. For our analysis, function calls are only of interest if they return a reference to a global variable that undergoes a type usage later on. We handle both cases as follows.

If a function that returns a variable in its body is called in some place, the CALL-FUNC rule derives the points-to relation between the expression being returned and the variable that receives the function's return value. For function pointers, the rule CALL-PTR transforms the call of the function pointer into a call of the function that it points to. The following example shows their application:

Example 7.7 The code lines below define a function as well as a function pointer, both of which are invoked afterwards:

```
1  int *x, *y, *z;
2  int* f() { return x; }
3  int* (*pf)() = &f;
4  y = f();
5  z = pf();
```

The function pointer '`pf`' points to function '`f`'. The inference rules derive the following points-to set from this code:

(1)	RETURN, line 2:	$\langle f, \varepsilon\rangle \mapsto \langle x, \Rsh\rangle$
(2)	ASSIGN, line 3:	$\langle pf, \varepsilon\rangle \mapsto \langle f, \&\rangle$
(3)	ASSIGN, line 4:	$\langle y, \varepsilon\rangle \mapsto \langle f, \text{↳}\rangle$
(4)	ASSIGN, line 5:	$\langle z, \varepsilon\rangle \mapsto \langle pf, \text{↳}\rangle$
(5)	CALL-FUNC, (1), (3):	$\langle y, \varepsilon\rangle \mapsto \langle x, \varepsilon\rangle$
(6)	CALL-PTR, (2), (4):	$\langle z, \varepsilon\rangle \mapsto \langle f, \text{↳}\rangle$
(7)	CALL-FUNC, (1), (6):	$\langle z, \varepsilon\rangle \mapsto \langle x, \varepsilon\rangle$

▲

7.1.3.5 Deriving the Transitive Closure

After initializing the points-to map using the ASSIGN and RETURN rules, we apply the other rules listed in Figure 7.1 repeatedly. When no more new relations can be derived, the transitive closure of the points-to map has been found.

Since our rules allow arbitrary C expressions on the right-hand side to be mixed with symbol transformations which are chained to nested expressions, situations can arise where the combined expression has no equivalent in C. For example, all relations resulting from the RETURN rule are actually only useful as intermediate results and require further combination with other rules to produce new points-to relations. Consequently, we remove all relations without valid C equivalences from the points-to set once the transitive closure is found.

7.1.4 Step 2: Establishing Used-As Relations

In order to establish the used-as relations, we compare the type each global variable or structure field is used as to its declared type. For this comparison, we also take the points-to map into account that has been generated in the first step to detect indirect type usages by means of local variables. A type usage may occur in the context of:

Listing 7.1: Various usage patterns that establish a used-as relation between field 'data' of 'struct B' and 'struct A*'.

```c
struct A { int value; struct A *next; };
struct B { void *data; }

struct A* func1(struct A *a) { return a; }

struct A* func2() {
    struct B b;
    struct A a = { 42, b.data };    // struct initializer
    struct A *pa = b.data;          // pointer initializer
    pa = b.data;                    // assignment
    a = *((struct A*)b.data);       // dereference (*)
    ((struct A*)b.data)->value++;   // dereference (->)
    pa = func1(b.data);             // function parameter
    return b.data;                  // return statement
}
```

- assignment statements,
- initializers,
- pointer dereferencing after type casts,
- function parameters, and
- return statements.

An example for each of these usages is given in Listing 7.1. For all of the cases shown in the code fragment, our used-as analysis comes to the same conclusion: Field 'data' of 'struct B' having type 'void*' is in fact used as a pointer of type 'struct A' with no additional pointer arithmetic. This relation is stored in the type information for 'struct B' and will be considered whenever the field 'data' of such an object is accessed.

Special care needs to be taken for the types of global variables and for structures that are embedded in other structures. The type usages of such objects are specific to their context, which is the variable name for global variables or the embedding data structure for nested structures. The next example illustrates the underlying problem.

Example 7.8 Consider the listing in Example 5.2 on page 56 once again. The field 'list' in line 4 is defined as an embedded 'struct list_head' within structure 'module'. Listing 5.2 on page 59 shows how the Linux kernel uses this field to iterate over the list of loaded kernel modules. In a nutshell, the relevant code lines that constitute the interesting type usage can be summarized as follows:

```c
1 struct list_head modules, *__mptr;
2 struct module* mod;
3 __mptr = modules.next;
4 mod = (struct module*)((char*)__mptr -
5                         offsetof(struct module, list));
```

The field ‘`next`’ of global variable ‘`modules`’ is first assigned to a temporary variable in line 3. Then, ‘`list.next`’ is cast to ‘`char*`’ in line 4 before a specific offset is subtracted and the resulting address is cast to ‘`struct module*`’ again.

This example includes two context specific pointer usages: one for field ‘`next`’ of variable ‘`modules`’ and one for field ‘`list.next`’ within ‘`struct module`’. It is obvious that the respective pointer manipulations are valid only within the same usage context. If we were to apply the used-as relation as seen for ‘`list.next`’ to all other fields or variables of type ‘`struct list_head`’, we would essentially mix up all types that are arranged in doubly linked lists, leading to impractically high numbers of candidate types to consider. ▲

Our implementation correctly handles such situations and considers the usage context when evaluating used-as relations (see Section 7.2.2). This results in variable and type context-sensitive used-as relations, which reduces ambiguities and vastly improves the type accuracy for locating kernel objects. Note that we intentionally ignore type usages that occur outside of any context. For example, consider a pointer ‘`void* p`’ that is cast to some type ‘`foo*`’. Without anchoring this used-as relation to the context of a global variable or a data structure or union, we would come to the conclusion that *every* pointer of type ‘`void*`’ might be used as a pointer of type ‘`foo*`’, which is clearly not what we want.

7.2 Implementation

We have implemented the proposed used-as analysis as an extension to our VMI framework InSight (see Chapter 6). This tool uses the debugging symbols of the inspected kernel to locate global variables in the kernel's address space and derive the layout of data structures. The extended type information gathered from the used-as analysis now augments the static type information and thus the semantic knowledge μ of the software architecture. Since the external view-generating component $[\![g_{\lambda,\mu}]\!]$ depends on this knowledge as illustrated in Figure 6.1, this extension allows InSight to consider all possible pointer interpretations of any field or variable that occurs anywhere in the kernel's source code.

7.2.1 Source Code Level Analysis

The consolidated type information extracted from the debugging symbols builds the initial knowledge base for InSight. To capture the dynamic pointer manipulations of the kernel, InSight parses the kernel's source code in a second step, performs the used-as analysis, and builds an extended type graph from the collected information to reflect such dynamic manipulations. Through this combination of debugging symbols and static code analysis, InSight achieves a high object coverage and type accuracy.

The Linux kernel comes with a highly sophisticated build system based on the GNU '`make`' utility. Depending on the kernel configuration and system environment, the build system selects proper compiler parameters, macro definitions, and include paths to compile the source files. In order to avoid having to mimic this complex system, we instead use a small wrapper script for the GNU C compiler that stores the preprocessed source files during compilation in a separate directory. Thus, we recompile the guest kernel with debugging symbol generation enabled and set our wrapper script as the compiler to be used. This assures that the code being compiled exactly corresponds to the code that InSight analyzes later on.

In our experience, the used-as analysis of almost 230 million lines (6.4 GB) of preprocessed C code of a Linux 2.6.32 kernel takes less than three hours on a standard PC equipped with a CPU of type Intel Core 2 Quad Q9550 (2.83 GHz) and 12 GB of RAM.

7.2.2 Applying Used-As Relations

The used-as relations found by our analysis are integrated into the type database of InSight. As a result, all interfaces that InSight provides benefit from the additional semantic knowledge. Each used-as relation consists of:

- a source global variable or field within a data structure (the context type),
- a target type (the used-as type), and
- a C expression describing the transformation from the source value to the target address.

The expression typically includes a reference to a variable x, as we have previously denoted with $e(x)$. Recall that internally, kernel objects are represented as *Instance* objects. Instances basically consist of a virtual address (where the object resides in memory) and an associated type. If that type is a '`struct`', a '`union`', or an array, its fields or elements can be accessed to derive further instances. Whenever a global variable v or a field m of a data structure is accessed, the following steps are performed:

1. The type database is queried for used-as relations of the accessed variable or field. If no such relations exist, an instance of the defined type of v or m, respectively, is returned.

2. For all used-as relations, the context of the current instance is tested for compatibility with the context of variable x within expression $e(x)$.[2] The context includes the variable name of v (for global variables) and the data type of v or m, respectively. If more than one compatible relation is found, this ambiguity is handled as follows:

   2.1 If exactly one of the used-as relations is context specific with respect to variable v, that relation is selected.

   2.2 If exactly one used-as relation has a larger context than all other relations, that relation is selected. The context size is determined by the number of fields which are accessed in $e(x)$, starting from the context type — that is, the type of x.

   2.3 If none of the previous steps selected a single relation to use, the pointer usage of the variable or field is ambiguous and cannot be resolved automatically. In that case, an instance of its defined type is returned.

3. If exactly one used-as relation was selected, the expression $e(x)$ is evaluated to derive the target address, where x is substituted with the current instance. If no errors occur during the evaluation (e. g., null pointer dereference, page not present, division by zero, etc.), then a new instance of the target type at the evaluated address is created; otherwise, an error is returned.

4. The memory area claimed by the new instance is compared to the area occupied by the instance of variable x. If the areas overlap, the new instance is discarded and an instance of the field's defined type is returned; otherwise, the new instance derived from the evaluation is returned.

The strategy for resolving ambiguities described in step 2 guarantees that the context of specific type usages is preserved. This approach counters potential problems as outlined in Example 7.8.

The test for memory overlaps in step 4 is especially important for linked lists: The '`next`' and '`prev`' fields of an empty list structure point to the structure itself, as shown in Figure 5.2 on page 57. If a global variable of type '`struct list_head`' is empty, the instance derived from applying a used-as relation might result in a non-existing object that seems to embed the global variable. The overlap test detects such situations and returns the originally defined type instead. Given that the instance for variable x was valid, this fall-back solution always produces the correct result.

In our experiments, typically less than 0.5% of all types had fields with ambiguous used-as relations. In other words, 99.5% of all data structures can be handled in a completely automated way. While this high number sounds promising, it does not allow

[2]In fact, the compatibility tests are performed only once when the symbols are loaded and the result is stored in a hash table to minimize the overhead at runtime.

the conclusion that our semantic view of the kernel's memory is correct to the same degree. Unfortunately, there are still a number of open problems that lead to a higher uncertainty than the previously stated numbers indicate. We describe some of these problems in the remainder of this section and approaches for addressing them in the section that follows.

Legal usage of declared types. If a field has exactly one used-as relation, the algorithm described above chooses this type instead of the originally defined type. However, this does not always correspond to the way the kernel uses this type. Consider the definition of '`struct page`' in Listing 5.4 on page 63 once more. The field '`mapping`' in line 17 is defined as '`struct address_space*`', but in some situations this pointer points to a '`struct anon_vma`' object. The kernel distinguishes these two types by testing the least significant bit of the pointer address. If it is set, the field points to an '`anon_vma`' object, otherwise it points to an object of the declared type '`address_space`'. Before the pointer is dereferenced, the two least significant bits of the address are always masked. Due to the control flow insensitive nature of the used-as analysis, this technique does not support the identification of the condition in which the type usage occurs.

Code obfuscation. The Linux kernel provides a special macro '`RELOC_HIDE`' which uses inline assembler instructions to assign a source pointer to a target variable. The sole purpose of this macro is to obfuscate the relation between source and target for the compiler in order to prevent certain optimizations that the compiler would apply otherwise. This obfuscation affects InSight's source code analysis in the exact same way as a compiler, leading to an incomplete view.

Unions. In Section 5.2.4, we already explained the problematic '`union`' data type. Upon closer inspection, a union is not much different from ambiguous used-as relations: each union field represents a viable choice; however, only one is correct, and we cannot decide which field is valid without additional context information. As a matter of fact, most unions are embedded within '`struct`' types that carry the required context information, which is used by the kernel to determine the correct type at runtime. With a control flow sensitive analysis, we probably could identify the relevant context information automatically.

Per-CPU variables. On the x86 architecture, Linux supports multiple CPUs with symmetric multiprocessing (SMP). This feature also introduces so-called per-CPU variables into the kernel. As the name suggests, these variables provide one unique instance for each CPU in the system [CRKH05]. Any variable or field of a structure with a pointer type can be used as a per-CPU variable; they do not require any special type or otherwise indicative definition. What makes accessing these variables difficult is the fact that the per-CPU variable does not actually point to a valid object. Instead, an additional

CPU specific offset has to be added to the address. The offset is stored in the global array '`__per_cpu_offset[`i`]`', where i is the index of the respective CPU. This offset is non-null for all SMP kernels, even if only one CPU is available to the system.

Dynamic arrays. Some pointers are used as arrays of dynamic size, as discussed in Section 5.2.4. The number of elements for the array and thus the amount of allocated memory are determined at runtime. Sometimes the length of an array is stored in another variable; other times the end of an array is defined by a special terminating element (for example, a string is represented as an array of '`char`' values, where the end of the string is marked by a the value 0).

Ambiguous used-as relations. Finally, there is still the approximately 0.5% of data structures having fields with ambiguous used-as relations. Even though this number is quite low, it does not imply that only 0.5% of all kernel objects are affected by this problem. In our experience, only one field in a critical data structure that we cannot resolve automatically can be enough to hide a large number of further objects that are accessible only though this particular field.

7.3 Type Rule Engine

The approach for applying used-as relations as described in the previous section works well in many cases, but also has its limitations. First and foremost, the control flow insensitive nature of our analysis leads to the application of type usages without considering the conditions under which they were originally discovered, as discussed above. In addition, the generic approach for deriving new instances has proven to be too inflexible in order to cope with the wide variety of dynamic pointer manipulations that can be found in the code of the Linux kernel. If there is one conclusion that we can draw from our experience with Linux, it is that this highly complex piece of software is simply not well-suited for "one-fits-all" solutions.

In order to address the remaining challenges and support more flexible application of used-as relations, we added a *type rule engine* to InSight. The engine serves as the central database and management interface for the additional knowledge about type usages. The type rules themselves are specified in the extensible markup language (XML) [BPSM+97] and are automatically generated for all identified used-as relations; however, they can also be supplied by the user. In summary, the engine comes with the following advantages for our VMI framework:

1. The type rules allow the specification of used-as relations of variables and fields as before, with an arbitrary C expression to calculate the target address from a source pointer. However, through the usage of the human-readable XML format for their definition, this knowledge is much more transparent and easier to understand from

a user's perspective. He can easily inspect all the rules that are generated during the source code analysis. Most importantly, the user can now add, delete, and modify type rules as he sees fit to cover special cases.

2. A user can assign priorities to type rules. Rules with higher priority take precedence over rules with lower priority of the same context size. This process allows one to manually resolve ambiguous used-as relations.

3. In addition to the used-as relations as described in Section 7.2.2, the type rule engine also supports rules that execute JavaScript code to retrieve the instance for a field or a variable. With this powerful extension, all the exceptional cases that have been listed in the previous section can be handled.

In other words, the type rule engine allows the user to modify and extend the semantic knowledge μ of the inspected operating system using a standardized interface. This knowledge is then transparently applied by the VGC to the benefit of all VMI applications built upon our framework. As a consequence, the rule engine allows InSight to remain generic, in that it does not include OS- or architecture-dependent code to cope with sophisticated pointer manipulations, but at the same time, it supports handling of such cases through specific type rules. All manually contributed expert knowledge is centralized in reusable type rules. For better maintainability, each rule can specify the exact hardware architecture, OS family, and kernel version it supports. In the remainder of this section, we describe the usage of type rules in InSight and how they can help to address the open issues.

7.3.1 Expression Rules

Every type rule consists of a filter part and an action part. The filter defines the context in which the rule applies. Supported filter criteria are the data type, the type name, the variable name, and so on. A complete list of possible filters is given in Table 7.2.

The action defines what happens when an instance matches the filter. The "expression" action corresponds exactly to the used-as relations introduced in the previous section: It specifies a source type, a target type, and an expression in C syntax to calculate the target address from a source pointer or an integer type of pointer size.

Example 7.9 The complete specification of an expression type rule can is given in Listing 7.2. It shows a rule that allows one to follow the '`list.next`' field of an instance of type '`struct module`' to access the next element within the list of '`module`' objects. This rule was automatically generated during source code analysis of a 32-bit Linux guest. ▲

Note that a rule's name and description only have an informative character for the user and are irrelevant for our further discussion.

Listing 7.2: Example of an expression type rule that specifies how to access the next element within the list of '`struct module`' objects. For better readability, we omitted the mandatory encoding of the character ">" as an XML entity.

```xml
<typeknowledge version="1">
    <rules>
        <rule priority="2">
            <name>module.list.next</name>
            <description>
                ((struct module *)->list.next - 4) => (struct module *)
            </description>
            <filter>
                <datatype>struct</datatype>
                <typename>module</typename>
                <members>
                    <member>list</member>
                    <member>next</member>
                </members>
            </filter>
            <action type="expression">
                <sourcetype>struct module *src</sourcetype>
                <targettype>struct module *</targettype>
                <expression>src->list.next - 4</expression>
            </action>
        </rule>
    </rules>
</typeknowledge>
```

Filter name	*Description*
`datatype`	Matches the actual data type, e. g., ‘`Union`’, ‘`UInt32`’, etc.
`filename`	Matches binary file a global variable belongs to, e. g., ‘`vmlinux`’ for the kernel itself or ‘`snd.ko`’ for module “snd”.
`members`	Matches one or more levels of (nested) fields, specified as a list of ‘`member`’ elements. Each field may be further refined with ‘`datatype`’, ‘`typename`’ and ‘`size`’ attributes as described in this table.
`size`	Matches the type size.
`typename`	Matches the type name, either by a literal match, by a wildcard expression, or by a regular expression.
`typeid`	Matches type internal type ID, which might be required for anonymous or ambiguous structures.
`variablename`	Matches the variable name, either by a literal match, by a wildcard expression, or by a regular expression.

Table 7.2: Supported type filter criteria for specifying type rules. One or more of these filters have to be specified within the ‘`filter`’ element.

7.3.2 Script Rules

The expression rules are human readable and the specification of a priority helps to resolve several ambiguities. However, the expression action is static and cannot dynamically adapt to the context, as it would be required to resolve unions, for instance. To fill this gap, we introduce script type rules.

Script rules differ from expression rules only in their ‘`action`’ element. This action can either specify a file name containing JavaScript code and the name of a function within that file that should be called, or it may contain inline script code. The code is executed when the rule’s filter matches a variable or a field and receives the triggering instance and the accessed fields as arguments. The following return values are allowed:

- If the rule returns the boolean value ‘`false`’, then the script action is ignored and an object of the variable’s or field’s defined type is returned.
- If the rule returns a valid instance object, that instance is used as the object represented by the field or variable being accessed.
- If the rule returns an array of instance objects, that field or variable is considered to be an array of the given length with the given objects as elements.
- If the rule returns an invalid (null) instance, the field or variable is considered to be inaccessible.

The following example illustrates how script rules can be employed to disambiguate a ‘`union`’ field embedded within a structure.

Example 7.10 Consider Listing 7.3 which shows a script rule with inline code. In order for an instance of type ‘`struct sysfs_dirent`’ to match the rule, two consecutive field accesses are required. The first is an anonymous field (line 8) which happens to be the union in question, the second field uses the special wildcard ‘`match="any"`’ (line 9) which matches any field within the anonymous structure.

Two arguments are passed to the code in the ‘`arguments`’ array when it is executed: the first is the instance that matched the filter (line 13), the second is an array of (nested) fields that were accessed to match the filter (line 14). The fields are given as numeric identifiers, making them definite even for anonymous fields.

In order to decide which of the union fields is valid, the rule inspects the field ‘`s_member`’ of the embedding ‘`sysfs_dirent`’ structure. The value of this field is retrieved in line 15 and tested in line 27 to see if it matches the flag required for the field being accessed. If the correct flag is set, the rule returns ‘`false`’ to indicate that the defined type should be used as-is; otherwise, an invalid instance is returned to effectively forbid access to the wrong union field. ▲

It is not necessary to resolve all unions with script rules, even though it would be possible.[3] However, for some key data structures, this step is required in order to make further objects available that are only accessible over this path. The ‘`sysfs_dirent`’ structure shown in the previous example is such a key data structure as it organizes the entire *sysfs* file system hierarchy. With only this one handcrafted rule, we can correctly identify thousands of objects within this hierarchy and make them available to all applications using our VMI framework.

We use further script type rules to handle many of the special cases we described in Section 7.2.2, such as per-CPU variables, the ‘`page`’ structure, radix tree nodes, and the closely related ‘`idr`’ and ‘`idr_layer`’ structures.

Another useful application for script rules is global variables belonging to kernel modules. A kernel module can be loaded to an arbitrary memory location at runtime; thus, its global variables are not located at fixed addresses. The debugging symbols contain only the offset of each variable relative to the base address of its segment. In order to give access the global variables of loaded kernel modules, we have created a special type rule that locates the base address of the corresponding segment for any loaded module and corrects the variable’s address automatically. If the module is not loaded, the rule returns an invalid instance to indicate that this variable is currently not accessible. Due to the expressiveness of the type rule filters and the script code, one rule is sufficient to handle the variables of all kernel modules in a uniform way.

[3] The Linux kernel 2.6.32 of the Debian distribution used for most of our experiments contains a total of only 147 unions (not considering the data types of all kernel modules), so we consider creating manual rules for each of these data types a feasible task.

Listing 7.3: Example of a script type rule that resolves the anonymous union within '`struct sysfs_dirent`' based on the flags stored in the '`s_flags`' field. For better readability, we omitted the mandatory encoding of special characters as XML entities.

```
<rule priority="101">
    <name>sysfs_dirent.{s_dir,s_symlink,s_attr,s_bin_attr}</name>
    <description>Resolve sysfs entry type in union</description>
    <filter>
        <datatype>struct</datatype>
        <typename>sysfs_dirent</typename>
        <members>
            <member></member>
            <member match="any" />
        </members>
    </filter>
    <action type="inline">
        var dirent = arguments[0];
        var members = arguments[1];
        var flags = dirent.s_flags.toUInt32();
        var elem = dirent.Member(members[0]).Member(members[1]);

        // Types as defined in <fs/sysfs/sysfs.h>
        var sysfs_types = new Object({
            s_dir:      0x1,
            s_attr:     0x2,
            s_bin_attr: 0x4,
            s_symlink:  0x8
        });

        // If flag is set corresponding to name, the instance is valid
        if (flags & sysfs_types[elem.Name()])
            return false;
        else
            return new Instance();
    </action>
</rule>
```

7.3.3 Operating System Filters

Hand-written type rules are usually created to address a specific issue in the view generation process for a distinct OS. For example, all the previously shown type rules are designed for Linux guests. But even within the same OS family, things tend to change between hardware architectures or kernel versions. Thus, it is important to be able to specify which hardware architecture, OS family, and kernel version is targeted by a particular rule.

For the type rule engine, this can be achieved by specifying OS filter criteria for one or several rules. Table 7.3 lists all possible OS filters that can be used for type rules. They are added as attributes to the '`typeknowledge`', '`rules`' or '`rule`' element of the XML specification. Any attribute appearing on a deeper nesting level overrides the same attribute on a higher level, as the next example illustrates.

Example 7.11 Consider the following specification for type rules:

```
1  <typeknowledge version="1.0">
2      <rules os="linux" minosversion="2.6">
3          <rule>...</rule>
4          <rule minosversion="2.6.16">...</rule>
5          <rule>...</rule>
6      </rules>
7  </typeknowledge>
```

The '`rules`' element in line 2 specifies the target OS for the enclosed rules to be a Linux kernel of version 2.6 or greater. This is the effective OS filter for the rules in lines 3 and 5. The '`rule`' element in line 4 further refines the kernel version to be at least 2.6.16. The requirement for the Linux OS family for this rule, however, is inherited from the enclosing '`rules`' element. ▲

With the help of OS filters, the user can write type rules as generic as possible and

Filter name	*Description*
`architecture`	Specifies the system architecture of the targeted guest OS, for example, '`x86`', '`x86_pae`', or '`amd64`'.
`os`	Specifies the targeted operating system family, such as '`linux`' or '`windows`'.
`minosversion`	Specifies the minimum OS version that is compatible with a rule.
`maxosversion`	Specifies the maximum OS version that is compatible with a rule.

Table 7.3: Supported OS filter criteria for type rules. The filters are specified as attributes to the '`typeknowledge`', '`rules`' or any '`rule`' element.

at the same time be architecture and version specific where this is required without affecting view generation for all other guests.

7.4 Application

The most versatile interface of our framework is the scripting engine. We introduced its basic functionality in Section 6.2.5. With the additional coming knowledge in the form of used-as relations and type rules, accessing and using kernel objects becomes even more easy and intuitive.

Example 7.12 The following listing shows how to print a list of loaded kernel modules for a Linux guest, similar to the way that the standard tool '*lsmod*' does:

```
var head = new Instance("modules");
var m = head.next;   // modules.next used as struct module
while (m.MemberAddress("list") != head.Address()) {
    println(m.name, m.args);
    m = m.list.next; // module.list.next used as struct module
}
```

▲

In direct comparison with the JavaScript code given in Example 6.7 on page 84 which performs the same task, this version is much more compact and easier to read. The most important improvement, however, is the fact that this code does not require any more expert knowledge from the user about the expected types and how the lists work internally. All pointer arithmetic and type casts are done transparently behind the scenes. The user can focus on the actual introspection task without having to deal with manual address calculations. In addition, the less manual pointer arithmetic the code contains, the more portable it is across different kernel versions and system architectures.

7.5 Related Work

The used-as analysis introduced in this chapter is similar to an intra-procedural, field-sensitive, control flow insensitive points-to analysis. In fact, the first step of our analysis involves a traditional points-to analysis on full C source code. In the following, we describe some well-known algorithms for points-to analysis followed by a discussion of how our used-as analysis is different.

In his PhD thesis, Andersen [And94] moves away from the established alias analysis for pointers and introduces a constraint-based pointer analysis algorithm. His algorithm approximates the locations a pointer variable may point to by a set of abstract pointer locations which are also intra-procedural, field-sensitive, and control flow insensitive. To derive this points-to set, he defines a set of type inference rules based on the standard semantics of the C programming language. The type inference rules are then applied to the initial set repeatedly until the transitive closure of all locations for each pointer variable is found. This algorithm produces fairly precise points-to sets but does not scale well for large programs.

Steensgaard [Ste96] proposes a unification-based, inter-procedural pointer analysis algorithm. The focus of this algorithm is a low algorithmic complexity which is almost linear in the size of the program, as opposed to exponential worst-case run-time of Andersen's algorithm[4]. This low run-time complexity comes at the cost of less precise points-to sets. In contrast to Andersen's approach, this algorithm not only infers locations for pointer variables, but also types of values in these locations. The algorithm treats all assignments as bi-directional. That is, for every assignment '`x = y`', the algorithm unifies the points-to sets of '`x`' and '`y`' and concludes that both variables might point to the same set of locations. When a variable might point to several locations, all locations are restricted to hold values of the same type. In other words, the algorithm does not allow that the same variable might point to objects of different types.

In order to achieve better accuracy with some added algorithmic complexity, Das extends Steensgaard's unification-based approach to a "one level flow" algorithm [Das00]. The idea is to avoid unification of pointer locations at the first level within the points-to graph while unifying the pointer locations in derived pointer locations. Instead, he introduces directional flow edges between points-to sets at the first level. For example, the assignment '`x = y`' introduces a flow edge from the pointer target node of '`y`' to the pointer target node of '`x`'. This solution improves the precision compared to Steensgaard; however, the pointer locations related by flow edges are still restricted to point to values of the same type.

Heintze and Tardieu [HT01] describe a context-insensitive variation of Andersen's algorithm. Compared to Andersen, their algorithm reduces the runtime complexity by implementing dynamic transitive closure. For their analysis, the authors maintain the points-to graph in a pre-transitive state. If information about a particular node in the graph is required, they compute the required data on the fly. Through a sophisticated combination of graph reachability information, cycle elimination, and information caching, their algorithm scales well to "millions of lines of C code in a second", according to the authors. This constrained-based approach uses only four deduction rules for computing the points-to set, which are essentially simplified versions of our ASSIGN, TRANS, STAR-1, and STAR-2 rules defined in Figure 7.1 on page 96. Since the application of their algorithm is different from ours, they do not require deduction rules for handling return statements and function (pointer) invocations.

Pearce et al. [PKH07] propose a field-sensitive, flow-insensitive pointer analysis technique. Their approach is comparable to the work of Heintze and Tardieu, however it

[4]However, Steensgaard claims that the complexity of Andersen's algorithm for real-world programs typical is approximately $\mathcal{O}(n^4)$ rather than $\mathcal{O}(exp(n))$, where n is the program size.

is able to handle function invocations and function pointers. Similar to Andersen, they achieve field sensitivity by treating each field of a data structure as a unique variable, leading to an improved accuracy compared to Heintze and Tardieu.

Much more work has been published in the field of pointer analysis (e. g., [CWZ90, WL95, SH97, FFSA98, YHR99, HL07]). In general, they all follow the basic method of either Andersen or Steensgaard and improve on different features such as performance, scalability, context sensitivity, control flow sensitivity, or precision. However, all of the approaches mentioned so far differ from our used-as analysis in one important aspect: they were not developed with pointer arithmetic in mind as it is used in highly optimized kernel code. For example, none of the constrained-based approaches based on Andersen's analysis provides deduction rules for arithmetic expressions with pointers. The unification-based algorithms following Steensgaard's idea even constrain pointer variables to only point to objects of the same type. However, the examples given in Chapter 5 illustrate that algorithms with this limitation are not applicable for the code we analyze: a field of type '`void*`' is often used as a pointer to different types of objects depending on the context. Another major difference is the nature of our analysis: while the other approaches try to approximate the *locations* a pointer might point to, we are instead interested in the *type* of that pointer location. In addition, we allow more complex transformations from a source pointer to a target location; that is, we not only consider direct assignments, indirect assignments and transitive relations, but also support complete C expressions to compute the target address.

The KOP system introduced by Carbone et al. [CCL$^+$09] is the most closely related work to our used-as analysis. In fact, this work was the inspiration for our used-as analysis. Their algorithm is an extension of the work done by Heintze and Tardieu [HT01] and uses the same four deduction rules. However, they allow some limited pointer arithmetic — their deduction rules support adding an integer offset value to a pointer address in the assignment statement. The other three rules consider this offset when deriving new pointer locations from the intermediate set. While this approach represents a step into the right direction, it is insufficient in our experience; our used-as analysis of the entire Linux kernel reveals that complex arithmetic expressions involving bit shifts and bitwise logical operations to calculate the target address are not uncommon in kernel code.

7.6 Summary

We have presented an analysis technique to capture dynamic pointer and type manipulations in full C source code. In contrast to the rather data-flow-oriented points-to analysis, our analysis is type-centric and establishes used-as relations between global variables, structure fields and pointer types. In order to support the complete emulation of dynamic pointer manipulations, our approach also extracts any pointer arithmetic that is performed to yield the target address from a pointer value and supports arbitrary C expressions. The implementation of this kind of analysis in our VMI framework, InSight, augments the type information it was already using.

In order to support type usages that require even more context information, we added a type rule engine to our framework that enables the user to specify additional semantic knowledge in the form of XML type rules. The rule engine allows us to address cases which cannot be resolved automatically based on the source code analysis, such as union fields or per-CPU variables. The type knowledge can be expressed in the form of small JavaScript programs that are executed for triggering objects. This code is able to dynamically resolve fields and variables as they are accessed using more context information than the source code analysis provides. These rules represent a centralized database of expert knowledge that is transparently available to all applications based on our framework.

With these extensions, InSight is now able to consider all feasible interpretations of a field or variable, leading to vastly improved coverage and accuracy when performing introspection tasks.

Chapter 8

Evaluation

The goal of this work is to generate a view of a guest OS's state running inside a VM that is as close to the guest internal view as possible. A VGC that is capable of generating such a view is said to be full state applicable (see Section 3.3.1). In Chapter 6 and 7, we describe our approach for generating views with full state applicability from the hypervisor level. For all of our methods, we strive for a high degree of automation and accuracy. In this chapter, we evaluate the performance of our VMI framework, InSight, with regard to several aspects. The results of our evaluation provide evidence for the effectiveness and efficiency of our approach.

Chapter overview. One key performance indicator for InSight is how much kernel memory it can identify correctly and how many objects it misses. The details and results of the object coverage test can be found in Section 8.1. To illustrate the impact of the different knowledge sources that can be used, we repeat the evaluation with varying configurations. We demonstrate the effectiveness of VMI-based malware analysis and detection in Section 8.2. Here, InSight is used to analyze the changes applied by several rootkits and to detect hidden processes and network connections. We summarize our achievements in Section 8.3 along with our concluding remarks.

8.1 Kernel Object Coverage and Accuracy

The probably most important aspect for a VMI framework that is supposed to bridge the semantic gap is the "completeness" of the view that it generates. With the complete semantic knowledge at hand, a perfect understanding of the kernel memory, and no side effects from inspecting a running system caused by interrupting a "guest-atomic" operation[1], in theory the generated view can be as complete as the view within the guest

[1]That is, an operation that is executed without interruption within the guest, but still can be interrupted by the hypervisor.

itself.

In this section, we evaluate how much memory occupied by kernel objects InSight correctly identifies when building the kernel object graph for a Linux operating system. The graph is built by starting at the global variables of the kernel and following all array elements, structure fields and pointers to further objects within the kernel address space. While this approach works well for constructing the graph, we have to solve a different problem first: How do we verify an object once we found it? We describe our method for validation in the following.

8.1.1 Validating Kernel Objects

We use three different ways to validate the objects that we find: the slab caches of the Linux kernel, type invariants, and validation heuristics. Of these three mechanisms, the slab cache validation is the most reliable one. Unfortunately, this mechanism is not available to InSight in a productive environment due to its high run-time overhead. Thus, we only use it to assess the end result of each experiment; however, this information is not considered during construction of the object graph itself (i. e., the graph itself is in no way affected by the slab data). In contrast, the other two mechanisms only rely on architectural knowledge of the guest OS as well as on information collected as part of the used-as analysis. As a consequence, the latter two mechanisms are always available to InSight, in particular when generating views of a productive VMs.

8.1.1.1 Validation using Slab Caches

The Linux kernel provides several functions for allocating continuous memory areas internally. Most allocations are done through the *slab allocator* [BC05].[2] It is designed for frequent allocations and deallocations of kernel objects of the same type. Each of these types is managed in its dedicated *slab cache* to avoid memory fragmentation. Less frequently used types are allocated through a call to the kernel function '`kmalloc()`'. This function serves memory requests only in geometrical sizes, that is, sizes that are powers of 2. The '`kmalloc()`' function itself uses several slab caches to satisfy the memory requests, one for each power 2^i, $3 \leq i \leq k$, where k depends on the total amount of physical memory (typically $13 \leq k \leq 17$).

In order to use the slab caches to validate objects, we can keep book of all slab allocations and deallocations at runtime. Even though this approach incurs a significant performance overhead, it provides us with precise lists of allocated kernel memory at a given point of execution. For object specific caches, we expect to find the exact same objects at the same memory locations in the kernel graph as well as in the slab

[2]In the meanwhile, Linux provides three different implementations for the slab allocator on the Intel architecture and allows to select one at compile time. We already briefly touched upon one implementation, namely the SLUB allocator, in Example 5.7 on page 64. They all provide the same functionality and interface, so our discussion is not affected by the selected implementation.

caches. Other heap allocated objects without a type-specific cache cannot be identified directly. However, if we find such an object in generic slab space for serving '`kmalloc()`' allocations of the appropriate size, we take this as a strong indicator that the object is valid.

8.1.1.2 Validation with Type Invariants

Another possible way to validate objects is based on invariants for data structures. This approach is based on the observation that some fields with numeric types are only ever assigned a fixed set of constant values during regular operation. If these fields and the corresponding sets are known, we can test each such field against its value set and invalidate objects for which the invariants do not hold. This approach has been used in work related to detecting kernel-level attacks [BGI08, DGSTG09, WZS11].

For our evaluation, we use object invariants to validate the objects that we find, but not for the purpose of malware detection.[3] When InSight parses the kernel sources to extract the used-as relations, it also records the values assigned to each individual '`struct`' or '`union`' field. If a field is always set to compile-time constant values, we consider the set of all assigned values as invariant for that field. During construction of the kernel object graph, we ignore all objects that violate their type's invariants.

8.1.1.3 Heuristics for Object Validation

In addition to the slab caches and the type invariants, we use several heuristics to determine the validity of an object. The following tests are performed per object:

1. The object must be located in one of the well-known memory areas used by the Linux kernel (cf. Section 6.2.2.1).
2. The object must be accessible, that is, the page table entry for its address must be valid (cf. Section 6.2.2.3).
3. If the object represents a function pointer, it has to point into an executable memory area.
4. If the object represents a pointer, it must point into one of the well-known memory areas used by the Linux kernel.
5. If the object is of type '`struct list_head`', its '`next`' field must point to another '`struct list_head`' object whose '`prev`' field must point back to the current object. The same must be true when first following the '`prev`' field and then the '`next`' pointer back to the current object.

[3] We consider adding API functions to make the invariant checking mechanism available as part of the VMI framework for future releases of InSight.

6. If the object represents a data structure:
    a) If the object is not embedded within another object, its address must be aligned to a four byte boundary.
    b) If the object has a ‘`struct`’ type, all fields have to pass these tests recursively.
    c) If the object has a ‘`union`’ type, one field has to pass these tests recursively.

Each test has an associated penalty that is added when the test fails. The resulting penalties of all individual tests are combined along with the rating of the “parent” object into one rating for the current object.[4] The end result is a score in the real interval $[0, 1]$. It is a measure for the plausibility that an object of the given type at the given location is valid. When the object graph is constructed, we disregard all objects with a score below a certain threshold and do not follow any of its pointers anymore.

8.1.2 Test Environment

For our experiments, we use two different virtual machines. Both VMs run an installed Debian Linux distribution [deb], one is using the Intel IA-32 architecture (“i686”, PAE disabled) and the other the AMD64 (Intel 64) architecture. Both machines are running a kernel of version 2.6.32-5, where the suffix “5” is a Debian-specific version number. Each machine is assigned a single CPU, 512 MB of RAM, and 20 GB of virtual hard disk space. The host machine is equipped with an Intel Core2 Quad (Q9550) CPU running at 2.83 GHz and 12 GB RAM.

We simulate a web server scenario and install the mail transfer agent Exim, the web server Apache with PHP5 support, and the relational database system MySQL within both guests.

8.1.3 Experiments

All experiments are performed on guest physical memory snapshots of the respective VM stored on the local disk. First, we take a snapshot of each machine in idle state after it has completed its boot sequence and started all services (called “i686” and “AMD64” in the results). In addition, we create a high load on the 32-bit VM and take another snapshot (called “i686 HL”). The load consists of intense local file operations as well as parallel HTTP requests to the web server that delivers web pages from a Wiki system written in PHP.

For each memory snapshot, we build the kernel object graph several times while applying different knowledge sources. The graph is constructed using a breadth-first-search. A node in the graph describes a continuous memory area within the virtual

[4]The rating of the parent is multiplied with the intermediate rating of the current object. As a consequence, the rating of a new object can never be better than the rating of the parent. Global variables use a parent rating of 1.

address space claimed by an object of known type. For '`struct`', '`union`', or array types, only one node is created for the occupied memory region, even if they contain nested structures or arrays. However, we inspect each field or array element recursively and follow all pointers to further objects that are contained in nested types.

With regard to unions, we always consider all fields (unless a script rule is active that explicitly disambiguates the union, as shown in Example 7.10 on page 108). If several union fields have a pointer type, *all* pointers are followed and the resulting objects are added as children of the union's node. This results in potentially overlapping, contradicting, or invalid objects within the virtual address space. In other words, the generated graph represents an over-approximation of the true memory layout. In such cases, the plausibility score that is calculated per object as described in Section 8.1.1.3 often reveals the valid union field.

In the first experiment, we use only the types as defined in the debugging symbols of the kernel without considering any used-as relations. This represents the basic feature set of InSight is described in Chapter 6, but without the list-specific heuristics discussed in Section 6.2.3.1. The corresponding experiment is entitled "no rules" in the results.

In the next experiment, called "used-as", we add the used-as relations to the type rule engine as they are parsed from the kernel sources as described in Section 7.1. The used-as analysis results in 9 530 type rules for the 32-bit guest and 9 512 type rules for the 64-bit guest. We do not modify any type relations or adjust the corresponding rules' priorities; rather, we use the automatically generated rules as-is.

Finally, we add a total of 64 hand-written rules to the set of generated rules. These rules incorporate our knowledge of the kernel and are necessary to handle several of the dynamic pointer manipulations that cannot be resolved completely with our used-as analysis, as discussed at the end of Section 7.2.2. All of these manually created rules have high priorities to override any conflicting rule from the set of auto-generated rules. This experiment is entitled "used-as + manual" and leverages all features of our framework to increase object coverage and accuracy.

In order to better quantify the effect of the hand-written rules, we perform an additional control experiment where we abstain from applying the used-as relations and only use the manually created rules as knowledge source. This configuration is simply called "manual".

To measure the quality of the generated graph, we use the slab cache validation schemes described in Section 8.1.1. All objects found are compared to the objects allocated in the slab caches.

8.1.4 Results

Figure 8.1 shows the time required to construct the complete kernel object graph for each of the aforementioned experiments. The build time shows a strong variation among all experiments. The graph is built in almost 20 minutes for the 32-bit images and no

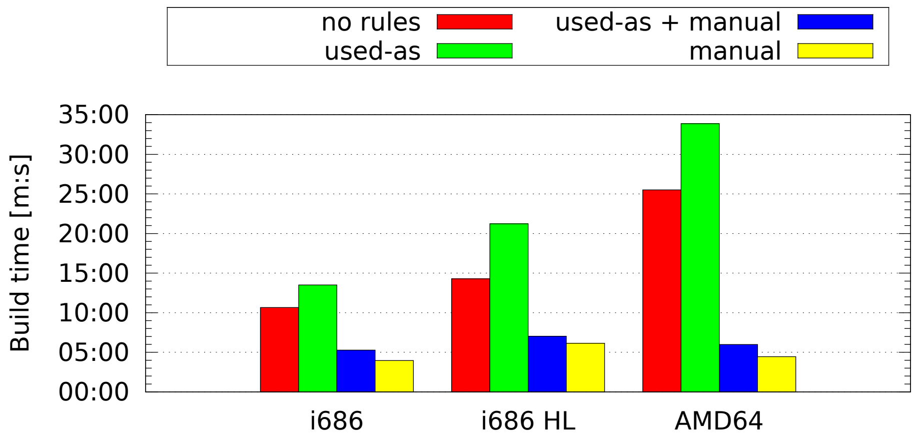


Figure 8.1: Time required to build the kernel object graph for each experiment.

more than 35 minutes for the 64-bit image. When the manually created type rules are used, the build time decreases to approximately 5 minutes for all snapshots.

The coverage and accuracy results for all three memory snapshots are listed in Table 8.1. We discuss these numbers with regard to the coverage of objects and correctness of the object graph in the following. These numbers also help to explain the variation in the build time.

8.1.4.1 Slab Object Coverage

The overall object coverage of the resulting kernel map is reflected in the amount of slab cache memory we can identify correctly. For our experiments, these objects represent the only "ground truth" for allocated memory we can compare our results to. As previously mentioned, some object types are allocated in generic memory areas and do not have a distinct slab cache. For those objects, we can only verify that the memory is currently in use, but we cannot say precisely which type of object is stored there. However, in our snapshots 80 to 87 percent of all slab memory is allocated by objects with known types, that is, the generic slab space accounts for no more than 20 percent in all experiments. Thus, the given slab object coverage for known types serves as a good metric for the overall coverage of objects in the kernel address space.

An object in the slab caches counts as found if the kernel graph contains an object of the exact same type at the exact memory address as an object in a type-specific cache. In the memory snapshots we use, the total amount of slab allocated memory varies between 4 529 kB and 6 990 kB depending on the system load and architecture. Of this memory, only between 10.5 and 20.9 percent is identified correctly if only the defined types are considered. These numbers are visualized as the red bars in Figure 8.2. When using the semantic knowledge derived by the used-as analysis, up to 46.9 percent of the memory can be properly identified (the green bars). This corresponds to improvements by factors

Experiment		*Objects in graph [kB]*									*Slab objects [kB]*		
snapsh.	*know. src.*	*total*	*globals*		*valid*		*undecided*		*invalid*		*total*	*found*	
i686	no rules	21 470	3 984	18.6%	1 078	5.0%	1 835	8.5%	14 573	67.9%	4 529	705	15.6%
i686	used-as	27 243	3 984	14.6%	2 762	10.1%	5 621	20.6%	14 876	54.6%	4 529	1 817	40.1%
i686	u.-a. + m.	9 984	4 060	40.7%	4 494	45.0%	1 398	14.0%	32	0.3%	4 529	4 468	98.7%
i686	manual	9 806	4 060	41.4%	4 471	45.6%	1 246	12.7%	29	0.3%	4 529	4 422	97.6%
i686 HL	no rules	14 395	3 984	27.7%	1 160	8.1%	1 246	8.7%	8 005	55.6%	6 990	733	10.5%
i686 HL	used-as	38 137	3 984	10.4%	3 850	10.1%	8 702	22.8%	21 601	56.6%	6 990	2 159	30.9%
i686 HL	u.-a. + m.	12 797	4 060	31.7%	6 961	54.4%	1 768	13.8%	8	0.1%	6 990	6 928	99.1%
i686 HL	manual	12 747	4 060	31.9%	6 958	54.6%	1 722	13.5%	7	0.1%	6 990	6 890	98.6%
AMD64	no rules	42 461	5 749	13.5%	1 988	4.7%	4 427	10.4%	30 297	71.4%	6 503	1 362	20.9%
AMD64	used-as	45 792	5 749	12.6%	4 352	9.5%	4 776	10.4%	30 915	67.5%	6 503	3 047	46.9%
AMD64	u.-a. + m.	15 005	5 917	39.4%	6 330	42.2%	2 731	18.2%	27	0.2%	6 503	6 279	96.6%
AMD64	manual	14 776	5 917	40.0%	6 276	42.5%	2 564	17.4%	19	0.1%	6 503	6 164	94.8%

Table 8.1: Comparison of the coverage and correctness of the kernel object graph for i686 and AMD64 guests when idle or under high load ("HL") using different knowledge sources. All numbers are given in kilobytes of allocated memory.

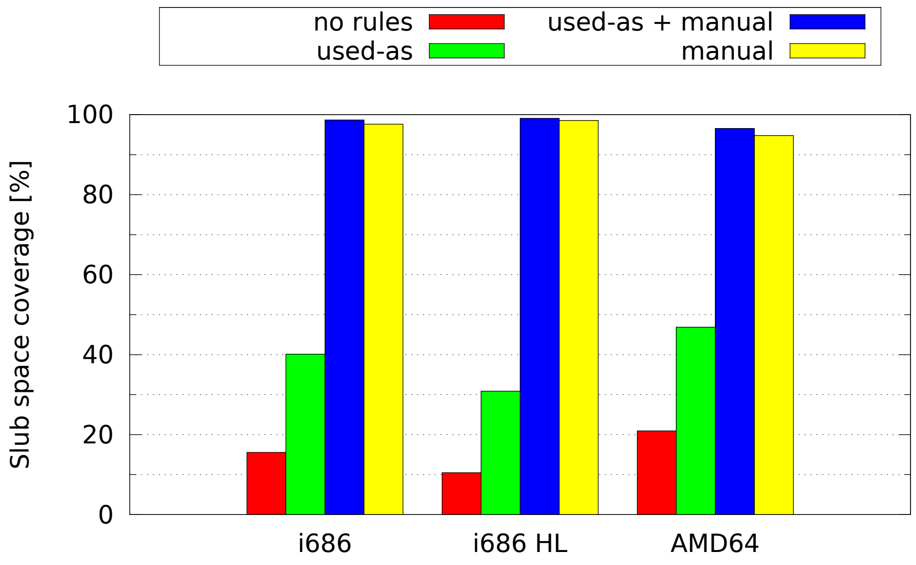


Figure 8.2: Coverage of slab cache memory for all experiments. This represents the values in the right-most column of Table 8.1.

between two and three, subject to which snapshot is used. With the additional semantic knowledge expressed in the hand-written type rules, the memory coverage rises to values between 96.6 and 99.1 percent (blue). In other words, almost the entire slab allocated memory can be identified correctly when all knowledge is taken into account. If only the hand-written rules are applied without the used-as rules, the results (yellow) are only slightly worse compared to the previous experiment. In Section 8.1.5, we discuss these numbers in greater detail.

8.1.4.2 Overall Object Validity

Another important performance indicator for our VGC is the correctness of the generated object graph. Table 8.1 shows the total amount of memory claimed by all objects found, by global variables, by valid objects, by invalid objects, as well as the amount of memory held by objects whose validity cannot be decided. These categories require some explanation.

We consider an object to be *valid* if it either is contained in a type-specific slab cache, or if it is embedded within a slab object. Conversely, we consider an object to be *invalid* if it overlaps with a slab object or a global variable and is not embedded within that type. For example, suppose we find an object of type '`struct list_head`' which falls into the memory area of a slab object. Let us assume this object is a '`task_struct`'. Now, if the newly found object is located at the offset of any of the fields with type '`list_head`' within '`task_struct`', such as '`tasks`', '`sibling`', or '`children`', the object is counted as valid, otherwise it counts towards the invalid objects. Also, for types that do have

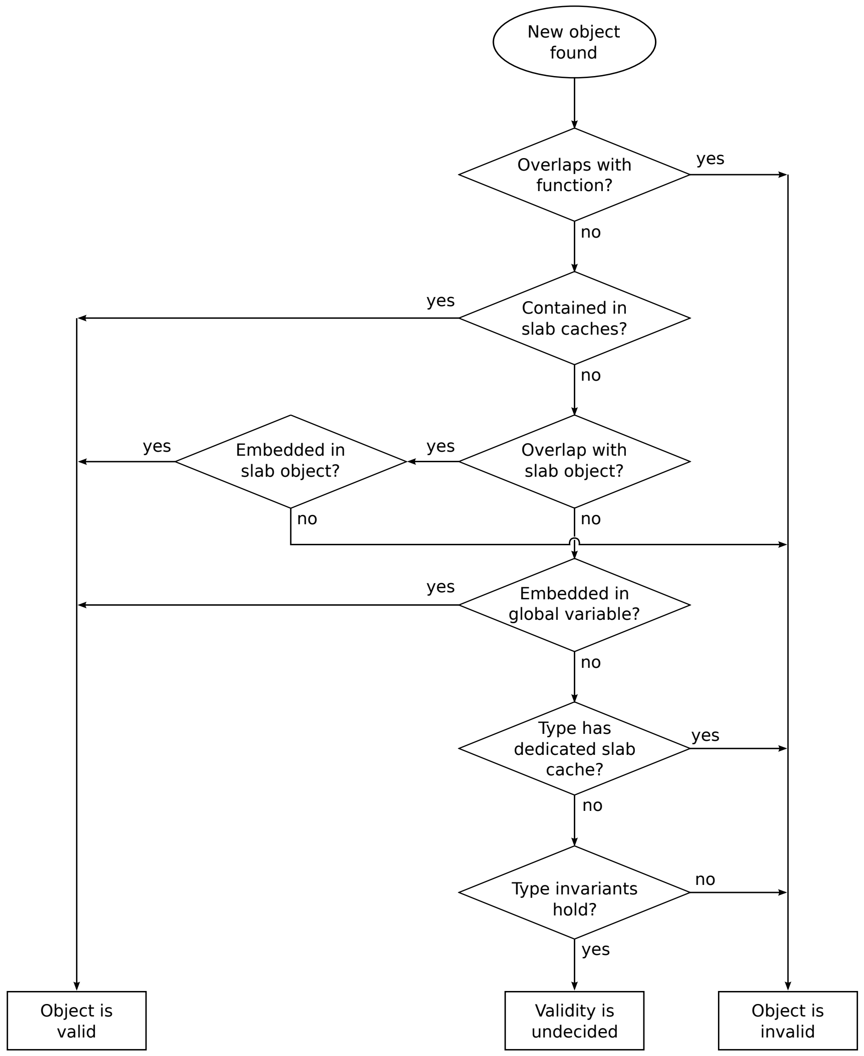


Figure 8.3: Decision function for the validity of an object found in kernel space.

a dedicated slab cache, we require that objects of that type are contained in the cache and count all such objects not found in the cache as invalid. In addition, any object is considered invalid that overlaps with the code section of a function.

Furthermore, we use the type invariants for '`struct`' objects (cf. Section 8.1.1.2) to validate data structures. In our experience, they are very useful to identify invalid or uninitialized objects. However, an objects whose type invariants hold is not necessarily valid. Thus, we still consider the validity of such an object to be *undecided* even in the presence of valid invariants. The whole process of deciding on the validity of an object is depicted in Figure 8.3.

The results for the overall object validity from Table 8.1 are visualized in Figure 8.4. The amount of memory for global variables is constant for each architecture, however

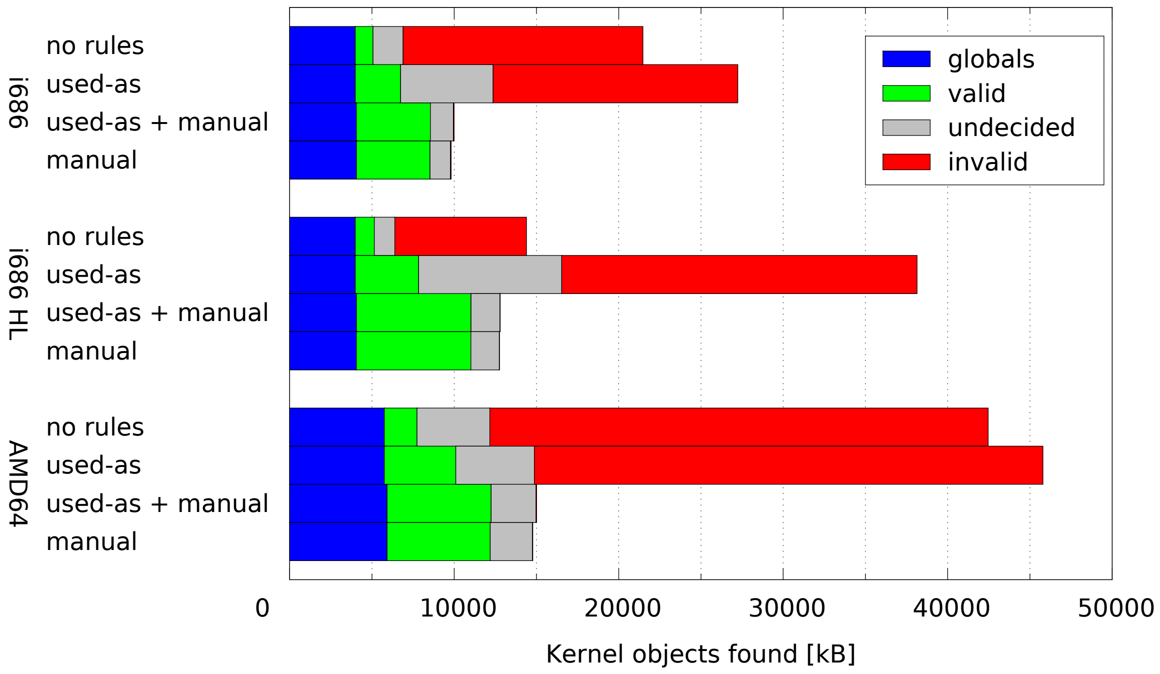
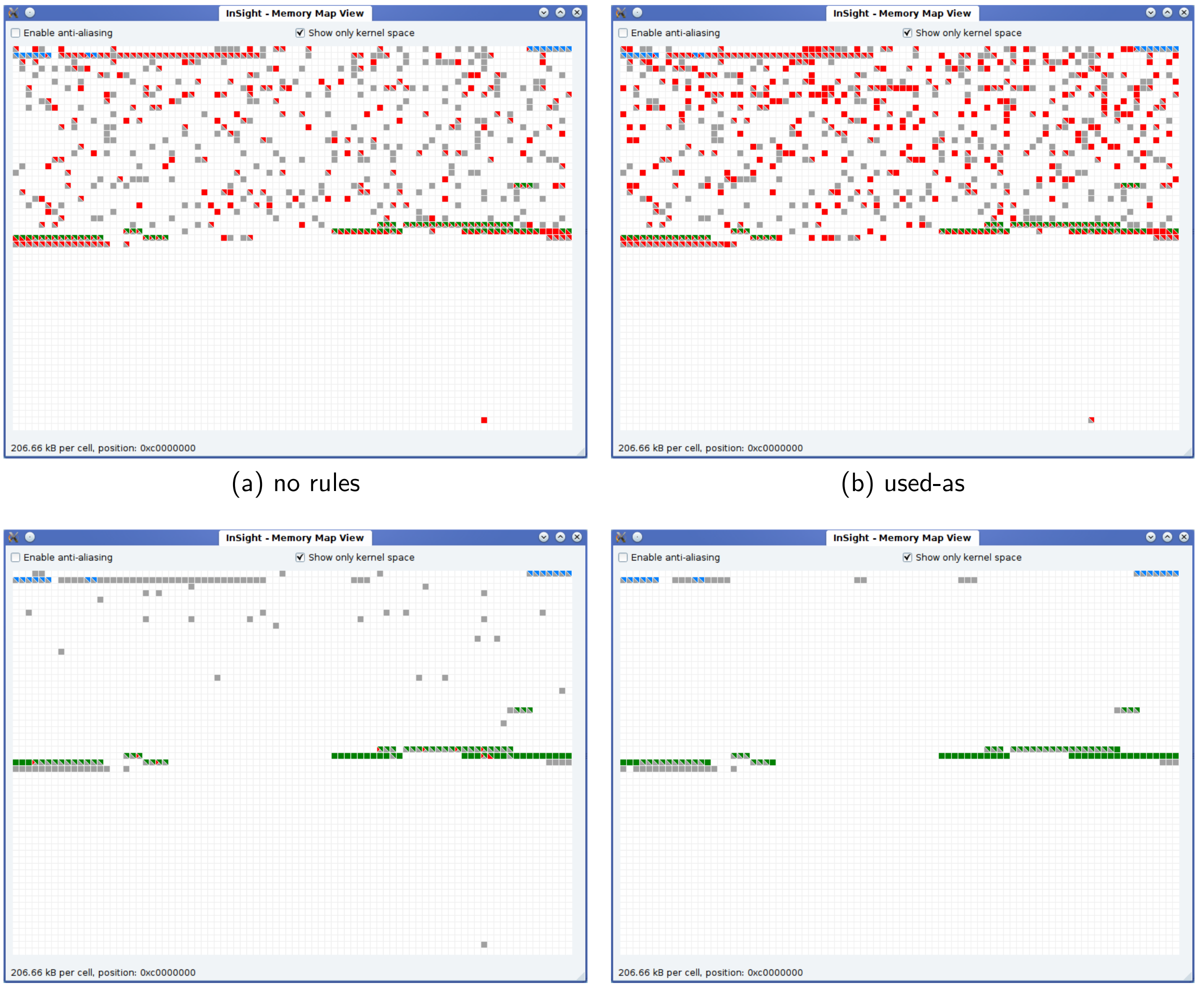


Figure 8.4: Visualization of the objects' validity within the generated kernel object graph for each experiment (cf. Table 8.1). The global variables ("globals") always count towards the valid objects.

it slightly increases when the manual type rules are considered. This is due to a hand-written script rule which allows us to access the global variables of loaded kernel modules, as described in Section 7.3. When comparing the experiments using no type rules and using the rules resulting from the used-as analysis, the amount of memory for valid objects increases by a factor between two and three. This is in accordance with the previous observations about the slab object coverage. In addition, more memory is classified as undecided or invalid. While this increase is only small for the "i686" and "AMD64" snapshots, it is significant for the snapshot taken under high system load.

When adding the 64 manual type rules to the knowledge pool while building the object graph, we not only identify even more memory correctly. At the same time, the amount of undecided memory is vastly reduced and the number of invalid objects is dropping close to zero. Similar to the slab coverage test, the experiments that rely on the hand-written rules alone do not perform substantially worse compared to the all-knowledge configurations. As already mentioned, we discuss these observations in Section 8.1.5 and also explain why at first glance the results seem to be more influenced by the hand-written rules then by the type knowledge derived from the used-as analysis.

Figure 8.5 visualizes the reconstructed layout of the kernel address space for the dif-



InSight - Memory Map View

[ ] Enable anti-aliasing [✓ Show only kernel space]

206.66 kB per cell, position: 0xc0000000

(a) no rules

InSight - Memory Map View

[ ] Enable anti-aliasing [✓ Show only kernel space]

206.66 kB per cell, position: 0xc0000000

(b) used-as

InSight - Memory Map View

[ ] Enable anti-aliasing [✓ Show only kernel space]

206.66 kB per cell, position: 0xc0000000

(c) used-as + manual

InSight - Memory Map View

[ ] Enable anti-aliasing [✓ Show only kernel space]

206.66 kB per cell, position: 0xc0000000

(d) objects in slab caches

Figure 8.5: Visualization of the kernel address space for the different sets of type knowledge applied to the "i686 HL" snapshot. The top left corner represents the `PAGE_OFFSET` ($0x\mathrm{C000\,0000}$), the lower right corner marks the end of the address space ($0x\mathrm{FFFF\,FFFF}$). Memory cells containing code are drawn in blue, cells with valid, invalid, and undecided kernel objects drawn in green, red, and gray, respectively. Global variables also appear as gray squares. As a point of reference, (d) only shows global variables, code sections, and the objects stored in the slab caches. The similarity between (c) and the "ground truth" in (d) are obvious.

ferent sets of type rules applied to the “i686 HL” snapshot. As a point of reference, the memory that is used by global variables and given out by slab caches is shown in (d) as gray and green squares, code sections are drawn in blue.[5] This represents the “ground truth” that we can rely on to verify our results. The memory layout according to the kernel object graph built by InSight with different sources of knowledge are depicted in (a), (b), and (c). The squares in green, red, and gray represent the memory occupied by valid, invalid, and undecided objects, respectively. In (a) and (b), the high number of invalid objects lets the number of valid objects appear small in comparison and distorts the overall picture. In contrast, the similarities between (c) and (d) are obvious and do not require further explanation.

8.1.5 Discussion

The coverage test in Figure 8.2 shows that the type knowledge derived from the used-as analysis significantly improves the results without depending on human experts. Compared to the baseline of using only the declared types, the automatically generated type rules increase the slab coverage by a factor of almost three for the “i686 HL” snapshot. Since only a small fraction of the slab allocated memory can be identified without additional semantic knowledge, these numbers give evidence of the prevalence of dynamic pointer and type manipulations throughout the Linux kernel. This observation emphasizes the necessity for view-generating methods that go beyond using the debugging symbols and some hard-coded offsets.

Taking a closer look at the overall object validity in Figure 8.4, however, it becomes apparent that the used-as knowledge does not help to reduce the amount of invalid objects at all. This is not surprising as our approach considers all possible interpretations of a pointer within its type or variable context, leading to an over-approximation of candidate types. The number of invalid objects also correlates to the variation in the build time, shown in Figure 8.1. While the validation heuristics introduced in Section 8.1.1 helps to disambiguate many cases with multiple interpretations for one pointer, it is far from perfect and fails to classify lots of invalid objects correctly.

In summary, our results illustrate the potential of a completely automated approach such as our used-as analysis on one hand, but also its limitation on the other hand: with a total of 46.9 percent of identified slab memory in the best case, the used-as knowledge has substantially improved the coverage, but we are still missing large areas of slab allocated memory. What is worse, the additional knowledge does not help us to reduce the extent of invalid objects. This limitation is due to the fact that the kernel contains even more complex code constructs that hide objects from a VGC other than straightforward pointer arithmetic and type casts. We already examined several of such cases in Section 7.2.2.

[5]Note that in this visualization, objects of global variables are drawn as gray squares as opposed to Figure 8.4 where memory occupied by global variables is represented in blue.

An additional limitation of our analysis stems from the fact that it requires a type context to establish a used-as relation. Recall that a pointer usage is anchored to the context of either a global variable or a (nested) field of a data structure, but not to a control flow point, scope, or function. The reason for this is that we access the objects by starting at global variables and following their pointer fields; we do not have enough information to link an object to the scope or code in which it was initially allocated or most recently used. Nevertheless, some crucial pointer usages occur without a proper field or variable context, which leads our used-as analysis to ignore them. Most notably, the Linux virtual file system (VFS) casts pointers of generic inode structures directly to file system specific specializations of this data structure. As a consequence, these specialized inodes are lost to InSight even though they resemble a significant portion of slab allocated memory in our experiments.

In order to improve the slab memory coverage, we examined the usage of several data types that InSight was missing at the source code level, including the inodes and their specializations. For the inodes, we identified a way to reliably determine the specific inode type based on context information within the generic inode itself. This knowledge is now expressed as a set of type rules which retrieve the file system specific inode whenever a generic inode pointer is dereferenced.

To reduce the number of invalid objects, we took a closer look at the types that account for the largest portions of invalid memory and identified the source pointer field where InSight assumes a wrong type for the first time on this path. As it turns out, many problems are caused by objects that the Linux kernel uses for various search trees, for example, the '`radix_tree_node`', the '`prio_tree_node`', or the '`idr_layer`' structure. In contrast to node fields that need to be embedded in data structures, such as '`list_head`' or '`hlist_node`', the search trees typically use the same node type for inner nodes and leaves likewise. The inner nodes point to further nodes of the tree whereas the leaves point to those objects that are actually stored within the tree. Usually, the tree nodes include some special field that is used by the kernel to distinguish between inner nodes and leave nodes. As discussed previously, this kind of conditional pointer usage cannot be captured by our used-as analysis. Thus, we created a number of script rules that take the context information within the tree nodes into account to discriminate between inner nodes and leaves.

Writing type rules for InSight's rule engine is easy and a pragmatic way to further narrow the semantic gap, as our experiments demonstrate. With only 64 hand-written rules, we are able to interpret the slab cache memory almost completely. At the same time, the memory claimed by invalid objects is reduced to only 0.3 percent in the worst case. This clearly shows the power and flexibility of our rule engine and the importance of incorporating user-supplied semantic knowledge in a generic, reusable way.

Note that the experiments which only use the set of manual rules for each snapshot also achieve very good results. This is due to the fact that several of these rules are based on type knowledge that was originally revealed by our used-as analysis. To improve the object coverage and minimize invalid objects, many hand-written script rules override existing expression rules to make better use of the context information, as described previously. In other words, thanks to our efforts to improve the results and leverage all possibilities of the type rule engine, the set of manual rules contains part of the knowledge previously discovered by our used-as analysis.

8.1.6 Comparison to KOP

Since the KOP system [CCL+09] shares some similarities with InSight and is able to build the object graph for a Windows kernel, we are interested in comparing the coverage and accuracy of both systems. According to the authors, KOP achieves an object coverage of over 99 percent for a Windows Vista guest. However, KOP makes use of specific features and architectural knowledge of Windows to reach these numbers, as described below. InSight also achieves a slab cache coverage of over 99 percent in the best case when using the combination of automatically derived and hand-written type rules. When abstaining from any type knowledge that involved an OS expert, the slab coverage in our experiments only gets close to 50 percent. The authors of KOP do not conduct any comparable experiment that relies on knowledge obtained solely from their source code analysis.

Apart from the degree of automation, the most obvious difference between the experiments performed with KOP and InSight is the guest operating system that is being analyzed. In the following, we take a closer look at the differences between the guest OSs and the techniques of KOP and InSight.

First of all, the number of symbols in both experiments differs significantly. For KOP, the authors state that the Windows kernel and 63 loaded drivers use 24 423 distinct data types and 9 629 global variables. The Linux kernel used for our evaluation contains 12 055 distinct data types and 19 548 global variables, not considering any modules. When all modules are taken into account as we did for our experiments, we end up with 73 456 types.[6] The source code analysis of KOP identifies a total of 4 914 type casts while our analysis reveals 9 530 used-as relations in the Linux sources. Thus, we see the complexity of our experiments much higher than the ones performed by KOP.

Second, KOP uses Windows specific domain knowledge to verify the objects and disambiguate multiple candidate types while it constructs the kernel object graph. By querying information from the memory management system of Windows, KOP can check the size and alignment of objects based on the pool allocation sizes. With this information, the system is able to automatically select the correct candidate type for unions or ambiguous type casts with high probability, as their results give evidence. In addition, the pool allocation blocks enable KOP to reliably identify dynamic arrays: when a pointer points at the beginning of an allocated memory area that is larger than the object size, that memory is assumed to contain a dynamic array of the area's size.

[6]Recall that the global variables of module are not accessible without the help of a hand-written script rule.

The information from the pool allocation corresponds approximately to the information we use for slab cache validation described in Section 8.1.1.1. Unfortunately, collecting this information for a Linux system introduces a tremendous performance overhead, making it impractical to rely on for building the object graph. The highly efficient slab cache implementation used by Linux, namely the SLUB allocator, does not collect the same accounting information than the Windows pool allocator. Since the SLUB allocator completely loses awareness of the exhausted memory pages it has been using for allocations [Cor07], there is no efficient way to correlate a virtual address to its object type or allocation size.

Finally, we have identified several instances of dynamic pointer and type manipulations in Section 7.2.2 or conditional type usage in Section 8.1.5 which simply cannot be covered by neither the analysis performed by InSight nor by KOP. As the authors of KOP do not mention similar problems in their work and the results of their evaluation does not indicate otherwise, we can only conclude that such unorthodox implementation techniques are not as prevalent in the Windows kernel as they can be found in the Linux source code.

8.2 Detecting System-Level Attacks

InSight is a view-generating component which is design—but not limited—to detect attacks that happen at the kernel level within the guest. To demonstrate the effectiveness of our framework for this particular VMI application, we use InSight to analyze the effects of six different rootkits for Linux, five of which are publicly available, one developed as a proof-of-concept in our research group. The following experiments are based on work done by Vogl and Kittel with InSight [Vog10, Kit10].

8.2.1 Test Environment

There are only few rootkits publicly available for Linux, most of which have been design for rather old version of the 2.6 kernel series. As a consequence, we had to use an older Linux distribution than for the coverage tests. We set up an Ubuntu 6.06 server (32-bit) in a virtual machine with 256 MB of RAM which comes with a Linux 2.6.15 kernel. Since the distribution's kernel contains hardening patches that would prevent the installation of several rootkits, we installed the original Linux kernel of the same version without any Ubuntu-specific modifications within the guest.

8.2.2 Experiments

The rootkits used for our experiments are listed in Table 8.2. They leverage different techniques for installing themselves into the kernel and for hooking kernel functions to provide their stealth services. Most come in the form of a loadable kernel module (LKM)

Rootkit	*Installation*	*Hooking technique*	*Source*
adore-ng	kernel module	virtual file system	[Ste]
EnyeLKM	kernel module	interrupt handler, system call handler	[Gar07]
IntoXonia-NG	kernel module	system call table	[S_4epen08]
Mood-NT	*/dev/kmem*	system call table	[Fal06]
override	kernel module	system call table	[Als06]
Y	kernel module	system call table	[Vog10]

Table 8.2: Rootkits used for our experiments

and remove themselves immediately from the list of kernel modules after they have been loaded by manipulating the '`next`' and '`prev`' pointers of the embedded '`list_head`' field, as described in Section 2.4.1.2. This allows the rootkits to stay completely functional while disappearing from the output of informative tools such as '*lsmod*'. The only exception here is the Mood-NT rootkit which injects itself into the system using the device node '*/dev/kmem*'. This device node allows direct read and write access to the kernel memory for the user "root" and is used by the rootkit to perform the required manipulations to the kernel's state. As a result, this rootkit can install itself even into kernels for which module loading was disabled.

In order to hide information from the legitimate user, all rootkits alter the control flow of certain kernel functions as depicted in Figure 2.4 on page 18. By hooking the system calls for reading directory contents, for example, a rootkit can filter the return value and exclude certain files within a directory, thus hiding them. While four rootkits directly overwrite function pointers in the system call table with pointers to their own code, EnyeLKM applies detour patches to the interrupt and the system call handler. Adore-ng hooks the VFS of the Linux kernel to filter the output. The VFS describes file systems within the file system hierarchy as objects with file system specific access functions to perform operations on the file system, such as reading and writing of files. These access functions are available as function pointer fields within the object describing the VFS component. It is these function pointers that are changed by adore-ng to point to its own code. In comparison to the other rootkits, these are the hooks that are placed most deeply within the control flow. The Y rootkit was created within our research group to demonstrate DKOM-based process hiding which none of the other rootkits supported.

We installed each of the rootkits individually in a clean VM image as described in the previous section. Each rootkit was then used to hide a process within the system, which was a simple remote shell using the '*netcat*' utility. If the rootkit supported hiding of network connections, the TCP port that '*netcat*' was listening on was also hidden.

To perform the detection, we took one snapshot of the guest physical memory before the rootkit was loaded, after it has been loaded, and one more time after the remote shell was listening on the specified TCP port. We then used InSight to find evidence for the existence of the rootkit, of the hidden process, and of the listening TCP socket.

8.2.3 Results

In all cases, we were able to detect the remote shell that was hidden by the rootkit. This result is not surprising since all network sockets that accept connections must be known to the kernel and are organized in global hash tables. For example, the TCP connections are kept in the variable '`tcp_hashinfo`'. The socket descriptors within that data structure are linked to the '`task_struct`' of the process that opened the socket, which was the '*netcat*' binary in our case. This is a good example for the limitations of DKOM, as already argued in Section 2.4.1: in order to keep the remote shell functional, the process must be able to receive and send network packets. As a consequence, a rootkit cannot remove all references to the process descriptor from the kernel state; instead, it has to omit those that are needed to facilitate the kernel-to-user-space transportation of I/O data.

The hooks that a rootkit places within the system are also straightforward to detect. In order to verify the integrity of the system call table, we use two different approaches. During normal operation of the system, the system call table does not change. When several sequential memory snapshots are available, we can compare the function pointers in the table between each two consecutive states and raise a red flag as soon as we detect any changes. In case only one guest state is available, we can verify that each entry in the table points to the entry of a known kernel function. If we find an entry that points into an unknown memory area, the table has been tampered with. Using the latter approach, we can also detect the hooks that adore-ng places within the virtual file system: We expect all function pointers for the file system specific access functions of all VFS objects to point to known kernel (or module) functions. If this is not the case, it must be the result of some malicious action. In contrast to the previously described hooking techniques, the detour patches applied by EnyeLKM are not directly reflected within kernel objects. Here, we have to verify the integrity of the corresponding handler functions by comparing the (binary) function code or a hash value of this code between consecutive states.

Detecting the rootkit itself is more difficult than detecting the modifications it applies to the kernel state. The LKM-based rootkits are not equally sophisticated in hiding their own tracks. For example, EnyeLKM, IntoXonia-NG, override, and Y remove their own '`module`' object from the '`modules`' list, but only adore-ng unregisters itself also from the '*sysfs*' file system. That is, the former rootkits leave artifacts in the tree of '`sysfs_dirent`' objects whose root is stored in the '`sysfs_root`' variable. By walking this tree, we can find references to the '`module`' objects of these rootkits. However, this is not possible for adore-ng. This rootkit successfully covers its tracks and completely removes all direct pointers to its '`module`' object. This is possible because the corresponding LKM only consists of callback functions which are invoked as the events occur; in contrast to a process, it does not need to be scheduled or wait for I/O operations. The only residues of adore-ng's module are the function pointers that point into the module's code section. The situation is similar for the Mood-NT rootkit. Here, we can successfully detect the inserted hooks that point to code of the rootkit. Since Mood-NT injects itself using the special device '*/dev/kmem*', the installation of this rootkit does not leave any further artifacts within the kernel's state.

8.3 Summary

In this chapter, we have evaluated our VMI framework, InSight, according to its intended purpose. The most crucial aspect for a view-generating component is the ability to create a view of the low-level system state of the guest that contains the kernel objects as the guest OS sees them. To give evidence of the good object coverage and accuracy that InSight provides, we used different knowledge sources to construct the graph of kernel objects for different memory snapshots. We verified our results by comparing them to the objects managed in the slab caches of the Linux kernel.

Our experiments have shown that the knowledge derived by our used-as analysis without any human intervention helps to increase the coverage of identified slab cache memory by factors between two and three, leading to a coverage of almost 50 percent in the best case. However, the results have also demonstrated the limitations of a fully automated approach based on a static code analysis. By adding only 64 hand-written rules to InSight's type rule engine, we were not only able to push the slab cache coverage to over 99 percent. At the same time, we could drastically reduced the number of invalid objects close to zero. These results illustrate that a combination of automatically derived and hand-written added semantic knowledge is a good compromise between automation and manual effort.

One of the typical use cases for our VMI framework is malware detection. To demonstrate the usefulness of InSight for this kind of application, we installed six different rootkits in a Linux guest and analyzed the induced changes within the guest physical memory. In all cases, we were able to identify the remote shell and the network connection that were hidden by each rootkit. We also could reproduce the hooking mechanisms that the rootkits installed to facilitate their stealth services. For four out of the five LKM-based rootkits, we were able to find links to the original '`module`' object describing the rootkit binary. For one LKM-based and the *kmem*-based sample, we only found the corresponding code sections but could not conclude which binary it belonged to. These findings show the effectiveness of VMI-based malware detection using our framework even when the attacks target the deepest corners of a system.

Chapter 9

Conclusion and Future Work

With system virtualization, we can benefit from new ways of countering the thread of compromised systems and detecting system-level attacks. The hypervisor plays a key role for system virtualization as it provides the virtual hardware interface capable of running an entire guest operating system. The virtual machines execute in isolation of the host system and the hypervisor. Even in the event of total compromise of the guest OS, the hardware resources of the host remain in full control of the hypervisor [SN05]. This feature provides the foundation for a technique called virtual machine introspection (VMI), the act of inspecting and interposing a virtual machine's state from the hypervisor level in isolation from the guest.

The introduction of VMI in the original work of Garfinkel and Rosenblum [GR03] opened a whole new area of research. One reason for the broad interest in this topic is the increasing ubiquity of system virtualization in every-day computing [Ros04, RG05]. Through its application on desktops [RW10], on servers [KKL$^+$07, Chi07], on mobile devices [Hei08, ABK09, BBD$^+$10], and in cloud computing [VRMCL08, AFG$^+$10], it clearly lends itself to adding security functionalities to the hypervisor in all of these scenarios.

Most research in this field still proposes rather straightforward application of the VMI paradigm for intrusion detection systems and system monitors [PCL07, PCSL08, SSG08, XJZ$^+$10, SWJX11], intrusion prevention systems [JKDC05, SLQP07, RJX08, RRXJ09, WJCN09], tools for forensic or malware analysis [HN08, MFPC10, RX10, DGPL11], application-aware firewalls [SG08], and kernel-level debuggers [FPMM10]. However, we begin to see work that uses virtualization features for performing VMI in completely new ways, for example, to isolate code within a VM [SLCL09, GDXJ11], to execute guest processes in isolation of the VM [SWJX11, VKSE13], to automatically create view-generating components [DGLZ$^+$11], or to use specific hardware features to trace in-guest events [PSE10, PSE11, VE12].

9.1 Contributions

One challenge that all VMI approaches have to face is the semantic gap [CN01], which describes the hypervisor's lack of inherent knowledge to interpret the low-level data that compromises the state of the guest OS. As a consequence, all VMI approaches have in common that they apply some sort of semantic knowledge to the guest's low-level state to derive a meaningful view of that state which is then used according to the scenario.

In Chapter 3, we have introduced a formal model that allows one to describe VMI approaches in a universal way. Based on the kind of semantic knowledge that is used, we can reason about the properties of the resulting system, such as hardware or software portability, the binding nature of the knowledge, or full state visibility and applicability. Using our model, we have identified three fundamental patterns to generate views from the guest's state, namely the out-of-band delivery, the in-band delivery, and the derivative pattern. Most of the time, these patterns are used in combination to benefit from different properties. To this date, we were able to describe any VMI approach we came across with our model and used it successfully in Chapter 4 to categorize and compare related work in this field.

Full state applicability is one of the properties that a VMI approach can possess. It combines the facts that the view-generating function not only has access to the entire state of the guest, but also has potential access to the semantic knowledge that is required to interpret that information. We have argued that in order to achieve this property, view generation has to combine the out-of-band delivery pattern together with the derivative pattern, that is, it has to run completely isolated from the guest and apply both hardware knowledge λ and software knowledge μ to the state. When the full state is applicable, the resulting view potentially reflects any activity that is carried out at any level of the hardware/software stack within the virtual machine, both benign and malicious. This is a very strong property, especially for detecting kernel-level attacks, for example, such as performed by rootkits.

For this reason, Chapter 5 has investigated on the challenges that a view-generating component has to overcome in order to apply the full state and to generate detailed views with a high coverage of state information. First, we have taken a look at all parts of the state information and identified the guest physical memory as the most difficult one to interpret. Then, we have detailed how a view of the memory can be generated and what semantic information is required in order to do so. The examples of real-world kernel code that we have given indicate that dynamic pointer and type manipulations are prevalent in highly efficient C code. We have shown that, for these examples, the semantic knowledge used by most out-of-band delivery VMI approaches is insufficient to bridge the semantic gap when full state applicability is desired.

In Chapter 6, we have introduced our VMI framework, InSight, at the first stage of its evolution. InSight has been one of the first tools that focuses on stand-alone view generation with full state applicability instead of applying just enough manually maintained expert knowledge for a given use-case. This allows our framework to support a broad spectrum of applications. With its rich set of interfaces, including a JavaScript engine for fast and safe development of new VMI approaches, InSight is well suited for a wide variety of introspection tasks, such as intrusion detection, digital forensics, secure monitoring, and kernel debugging.

While InSight was already useful and has been successfully applied within our research group at this stage [Kit10, Vog10], it required the inclusion of expert knowledge on a per-application basis to handle the dynamic pointer and type manipulations occurring in the kernel. In an attempt to automate the collection of semantic knowledge, we looked into approaches that perform source code analysis, foremost into the area of pointer and alias analysis [And94, Ste96, HT01, PKH07]. In this course, we have proposed a new analysis technique for full C source code based on the points-to analysis of Andersen [And94], which we call used-as analysis. This analysis is described in Chapter 7 and consists of two steps: first, a points-to analysis similar to Andersen's analysis, and, second, a type-centric analysis that establishes used-as relations between data types. The used-as relations describe which global variables or fields of data structures are used as types that contradict their defined type and how a source value has to be transformed to derive a target pointer.

We have implemented this kind of analysis as an extension to InSight to augment the semantic knowledge it has been using before. With this extension, our view-generating component is able to cope with type casts and perform the required pointer arithmetic to reveal kernel objects hidden behind '`void*`' pointers, generic lists, and even integer values holding a memory address. However, even with this major step towards achieving full state applicability, the used-as analysis has not been able to resolve type usages that depend on further context information or runtime behavior, such as dynamic arrays, per-CPU variables, unions, or other type ambiguities. This kind of dynamic behavior requires a dynamic solution. To address this issue, we have further enhanced InSight and added a type rule engine to the framework that allows the flexible and centralized specification of expert knowledge about data types and their usage within the kernel. By defining type rules in an XML format, the user can now guide the resolution of types and pointers depending on context and runtime information. For example, he can specify which field of a union is valid in a given situation. In other words, the user can now easily extend InSight's semantic knowledge on the fly in a generic, reusable way, independent of the application at hand.

In Chapter 8, we have evaluated the performance of our framework with different sets of available semantic knowledge, which corresponds to the evolutionary steps during the course of its development. Our experiments have shown how the object coverage and accuracy of InSight have been increasing with every extension, beginning with the semantic knowledge contained in the debugging symbols over the used-as relations to the type rule engine with manually created rules. With all knowledge combined, we have been able to increase the object overage to over 99 percent in the best case and still 97 percent in the worst case. At the same time, the number of invalid objects has dropped drastically to a value close to zero. In addition, the hand-written type rules have helped to speed up the process of building the whole graph of kernel objects substantially. By using InSight to identify various real-world kernel rootkits in running Linux systems, our view-generating component has proven to be useful for VMI-based security applications such as detecting system-level attacks, one of the intended use cases for our framework.

9.2 Future Research Directions

So far, our research has brought forward new ways of collecting semantic knowledge about an operating system and resulted in InSight, an elaborated and useful framework for building VMI applications. In this section, we outline some possible research directions to continue our work and to go forward towards bridging the semantic gap with full state applicability.

First of all, there are some worthwhile extensions of InSight that would mostly require implementation work, such as adding support for Windows guests by implementing a reader for debugging symbols in Microsoft's PDB format. Due to the closed source nature of the Windows kernel, we would not be able to analyze the source code, thus lacking the semantic knowledge resulting from the used-as analysis. As a consequence, more type rules would need to be created manually; however, this only emphasizes the importance and flexibility of InSight's type rule engine. Further improvements on the implementation could be achieved by replacing the current C parser that is employed for the used-as analysis with a more advanced compiler infrastructure, such as the LLVM/-Clang framework [LA04]. In addition, the type invariants that are already available internally should be made available via InSight's interfaces to even better support security applications such as intrusion detection. Finally, it would be interesting to explore the possibilities of InSight on embedded hardware platforms with virtualization support, such as systems equipped with the ARM Cortex-A15 processor [ARM12].

Another area for future research is the improvement of the used-as analysis to extract semantic knowledge with richer context information from the source code. This topic requires the solution of some interesting algorithmic challenges. To achieve a higher precision for the initial points-to set in the first step of our algorithm, the points-to analysis could be extended to model control flow sensitive pointer locations. This extension would also allow us to address an open problem in the second step, namely the lack of runtime context information for the used-as relations. By making the used-as analysis itself control flow sensitive, we could potentially detect runtime dependencies that we are currently missing. For example, we could identify the data fields that are tested before a particular field of an embedded union is accessed, thus automatically disambiguating the union. Or we could detect dynamic arrays and the corresponding length field based on their usage within the program's control flow. This could also lead to more specific type invariants for better validation of the kernel objects that we find. In summary, such an extension provides the potential for the next evolution in bridging the semantic gap.

With all this semantic information at hand, the externally generated view of a guest system's state can be almost as complete as the guest's internal view. However, the most notable difference is that external view generation is not dependent on the guest's cooperation and its examination does not interfere with the observed system — both properties which are extremely useful for security applications. When such a precise view is available to event-based monitoring approaches with a feedback channel to the guest OS, plenty of new possibilities open up, from hypervisor-based policy enforcement over application sandboxing to malware removal tools. A complete understanding of the state of an introspected system, that is, successfully bridging the semantic gap, is the key enabler for new VMI-based security applications.

Bibliography

A

[ABK09] A. Acharya, J. Buford, and V. Krishnaswamy. Phone virtualization using a microkernel hypervisor. In *Internet Multimedia Services Architecture and Applications (IMSAA), 2009 IEEE International Conference on*, pages 1–6. IEEE, 2009.

[AFG$^+$10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[Als06] Amir Alsbih. override rootkit, 2006. http://packetstormsecurity.org/UNIX/penetration/rootkits/override.tar.bz [cited December 20, 2012].

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Pearson/Addison Wesley, 2nd edition, 2007.

[AMD05] AMD. Secure virtual machine architecture reference manual. Technical report, Advanced Micro Devices, May 2005.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.

[ARM12] ARM Ltd. *Cortex-A15 MPCore Technical Reference Manual*, r3p2 edition, July 2012.

B

[BBD$^+$10] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The VMware mobile virtualization platform: is that a hypervisor in your pocket? *SIGOPS Oper. Syst. Rev.*, 44(4):124–135, December 2010.

[BC05] Daniel Pierre Bovet and Marco Cassetti. *Understanding the Linux Kernel.* O'Reilly, Sebastopol, CA, USA, 2005.

[BDG$^+$12] Zheng Bu, Toralv Dirro, Paula Greve, Yichong Lin, David Marcus, Francois Paget, Craig Schmugar, Jimmy Shah, Dan Sommer, Peter Szor, and Adam Wosotowsky. McAfee Threat-Report: Erstes Quartal 2012. Technical report, McAfee Labs, 2012. https://www.mcafee.com/de/resources/reports/rp-quarterly-threat-q1-2012.pdf [cited July 31, 2012].

[BGI08] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Computer Security Applications Conference (ACSAC 2008)*, pages 77–86, dec. 2008.

[BJW$^+$10] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, and D. Xu. DKSM: Subverting virtual machine introspection for fun and profit. In *Proceedings of the 29th IEEE International Symposium on Reliable Distributed Systems (SRDS 2010)*, New Delhi, India, oct 2010.

[BKI07] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *Proc. of IEEE Symposium on Security and Privacy. SP '07.*, pages 246–251, May 2007.

[Blu09] Bill Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System.* Wordware Publishing, Inc., 2009.

[BPSM$^+$97] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML). *World Wide Web Journal*, 2(4):27–66, 1997.

[BUP03] J. Butler, J.L. Undercoffer, and J. Pinkston. Hidden processes: the implication for intrusion detection. In *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, pages 116 – 121, june 2003.

[But04] Jamie Butler. DKOM (direct kernel object manipulation). Presentation slides, HBGary, 2004. http://www.blackhat.com/presentations/bh-europe-04/bh-eu-04-butler.pdf [cited December 14, 2012].

[BY07] Paul Barford and Vinod Yegneswaran. An inside look at botnets. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff

Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 171–191. Springer US, 2007.

C

[CCL+09] 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, pages 555–565, New York, NY, USA, 2009. ACM.

[Chi07] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall International, 1 edition, 2007. 978-0-13-234971-0.

[CN01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *HOTOS '01: Proceedings of the Eigth Workshop on Hot Topics in Operating Systems*, page 133, Washington, DC, USA, 2001. IEEE Computer Society.

[Cor07] Jonathan Corbet. The SLUB allocator, April 2007. https://lwn.net/Articles/229984/ [cited September 10, 2012].

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005.

[CTS+08] Bin-Hui Chou, Kohei Tatara, Taketoshi Sakuraba, Yoshiaki Hori, and Kouichi Sakurai. A secure virtualized logging scheme for digital forensics in comparison with kernel module approach. In *ISA '08: Proceedings of the 2008 International Conference on Information Security and Assurance*, pages 421–426, Washington, DC, USA, 2008. IEEE Computer Society.

[CWZ90] David R. Chase, Mark Wegmanl, and F. Kenneth Zadeckj. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 296–310. ACM, 1990.

[CXS+05] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium (SSYM'05)*, 2005.

D

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI ’00, pages 35–46, New York, NY, USA, 2000. ACM.

[deb] Debian project website. http://www.debian.org/ [cited December 7, 2012].

[DGLZ+11] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proc. of the IEEE Symp. on Security and Privacy*, May 2011.

[DGPL11] Brendan Dolan-Gavitt, Bryan Payne, and Wenke Lee. Leveraging forensic tools for virtual machine introspection. Technical Report GT-CS-11-05, Georgia Institute of Technology, 2011.

[DGSTG09] 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, pages 566–577, New York, NY, USA, 2009. ACM.

[Die11] Cornelius Diekmann. A VMI-based sandbox environment. Bachelor’s thesis, Technische Universität München, 2011.

[DKC+02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Operating Systems Review*, 36(SI):211–224, December 2002.

[DRSL08] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proc. of the 15th ACM conf. on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM.

E

[Eck12] Claudia Eckert. *IT-Sicherheit.* Oldenbourg, 7th edition, 2012.

F

[Fal06] Pierre Falda. Mood-NT rootkit, 2006. http://darkangel.antifork.org/codes.htm [cited December 20, 2012].

[Fer06] Peter Ferrie. Attacks on virtual machine emulators. Technical report, Symantec Corporation, 2006. http://service.symantec.com.my/avcenter/reference/Virtual_Machine_Threats.pdf.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM.

[FJ08] Marc Fossi and Eric Johnson. Report on the underground economy. Technical report, Symantec Corporation, November 2008. http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_underground_economy_report_11-2008-14525717.en-us.pdf [cited July 31, 2012].

[FPMM10] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 417–426, New York, NY, USA, 2010. ACM.

[Fre05] Free Standards Group. *DWARF Debugging Information Format, Version 3*, 2005.

G

[Gar07] David Reguera Garcia. EnyeLKM rootkit, 2007. http://www.fr33project.org/pages/projects/enyelkm.htm [cited December 12, 2012].

[GDXJ11] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 147–156. IEEE, 2011.

[GR03] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[GR10] Sergey Golovanov and Vyacheslav Rusakov. TDSS. Internet, August 2010. http://www.securelist.com/en/analysis/204792131/TDSS [cited December 15, 2012].

H

[HB06] Greg Hoglund and James Butler. *Rootkits*. Addison Wesley, 2006.

[HEF09] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. In Michael Backes and Peng Ning, editors, *Computer Security – ESORICS 2009*, volume 5789 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin / Heidelberg, 2009.

[Hei08] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, IIES '08, pages 11–16, New York, NY, USA, 2008. ACM.

[HFS98] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[HL07] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[HN08] Brian Hay and Kara Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, 2008.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 254–263, New York, NY, USA, 2001. ACM.

I

[IADB11] Hajime Inoue, Frank Adelstein, Matthew Donovan, and Stephen Brueckner. Automatically bridging the semantic gap using C interpreter. In *Proc. of 6th Annual Symposium on Information Assurance (ASIA'11)*, 2011.

[ins] InSight project website. https://code.google.com/p/insight-vmi/ [cited July 25, 2011].

[Int09] Intel, Inc,. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2009.

[ISO90] ISO/IEC 9899:1990 International Standard. *Programming Languages – C*, 1990.

J

[JADAD06] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 2006. USENIX Association.

[JADAD08] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 91–100, New York, NY, USA, 2008. ACM.

[JKDC05] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM symposium on Operating systems principles*, SOSP '05, pages 91–104, New York, NY, USA, 2005. ACM.

[JWX07] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proc. of the 14th ACM conf. on Computer and communications security*, pages 128–138, New York, NY, USA, 2007. ACM.

K

[Kas08] Eugene Kaspersky. *Malware*. Hanser, 2008.

[KH97] Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5):35–42, 1997.

[Kit10] Thomas Kittel. Design and implementation of a VMI-based intrusion detection system. Diploma thesis, Technische Universität München, October 2010.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, Ontario, Canada, June 2007.

[KS07] Aditya Kapoor and Ahmed Sallam. Rootkits part 2: A technical primer. Technical report, McAfee, Inc., 2007. http://www.mcafee.com/us/local_content/white_papers/wp_rootkits_0407.pdf [cited June 16, 2012].

L

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[Lee12] Thorsten Leemhuis. What's new in Linux 3.5. http://h-online.com/-1637461 [cited August 1, 2012], July 2012.

[LL06] Lionel Litty and David Lie. Manitou: a layer-below approach to fighting malware. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 6–11, New York, NY, USA, 2006. ACM.

[LLCL08] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 243–258, Berkeley, CA, USA, 2008. USENIX Association.

M

[Maf08] Robin Maffeo. AMD-V nested paging. Technical report, Advanced Micro Devices, Inc., 2008.

[MD73] Stuart E. Madnick and John J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the workshop on virtual computer systems*, pages 210–224, New York, NY, USA, 1973. ACM.

[MFPC10] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, pages 297–316, Berlin, Heidelberg, 2010. Springer-Verlag.

[Mic] Microsoft Corporation. A history of Windows. http://windows.microsoft.com/en-US/windows/history [cited August 1, 2012].

[Mis] Mission Critical Linux. Crash core analysis suite. http://mclx.com/projects/crash/ [cited June 11, 2012].

N

[Nam09] Yury Namestnikov. The economics of botnets. Technical report, Kaspersky Lab, July 2009. https://www.securelist.com/en/downloads/pdf/ynam_botnets_0907_en.pdf [cited July 31, 2012].

[NBH09] Kara Nance, Matt Bishop, and Brian Hay. Investigating the implications of virtual machine introspection for digital forensics. *Availability, Reliability and Security, International Conference on*, 0:1024–1029, 2009.

[NSL$^+$06] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, Vol. 10 Issue 3:167–177, August 2006.

P

[PCL07] B. D. Payne, M. D. P. Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Proc. of 23rd Anual Computer Security Applications Conference (ACSAC 2007)*, pages 385–397, December 2007.

[PCSL08] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. *IEEE Symposium on Security and Privacy*, 0:233–247, 2008.

[PFMA04] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaug. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. of USENIX Security Symposium*, San Diego, CA, 2004.

[PFWA06] Nick L. Petroni, Timothy Fraser, AAron Walters, and William A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proc. of 15th USENIX Security Symposium*, pages 289–304, San Diego, CA, 2006.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[PH07] Nick L. Petroni, Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 103–115, New York, NY, USA, 2007. ACM.

[PKH07] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1), November 2007.

[PO10] Enrico Perla and Massimiliano Oldani. *A Guide to Kernel Exploitation.* Syngress, 2010.

[PSE09] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In *Proc. of the 2nd ACM Workshop on Virtual Machine Security*, New York, NY, USA, 2009. ACM.

[PSE10] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Exploiting the x86 architecture to derive virtual machine state information. In *Proceedings of the 4th International Conference on Emerging Security Information, Systems and Technologies*, Venice, Italy, July 2010. IEEE Computer Society. Best Paper Award.

[PSE11] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, LNCS. Springer, November 2011.

[PSE13] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Leveraging string kernels for malware detection. In *Proceedings of the 7th International Conference on Network and System Security*, Lecture Notes in Computer Science. Springer, June 2013.

[qt] Qt project website. http://qt-project.org/ [cited September 10, 2012].

R

[RG05] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38:39–47, may 2005.

[Ric08] Robert Richardson. CSI computer crime & security survey. Technical report, Computer Security Institute, 2008. http://gocsi.com/sites/default/files/uploads/CSIsurvey2008.pdf [cited July 31, 2012].

[RJX08] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *Recent Advances in Intrusion Detection*, volume 5230 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2008.

[Ros04] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):34–40, July 2004.

[RRXJ09] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. *International Conference on Availability, Reliability and Security*, 0:74–81, 2009.

[RTWH09] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. Technical Report 18-2009, Berlin Institute of Technology, 2009.

[Rut06] Joanna Rutkowska. Introducing stealth malware taxonomy. Technical report, COSEINC Advanced Malware Labs, 2006.

[RW10] Joanna Rutkowska and Rafal Wojtczuk. Qubes OS architecture. Technical Report v0.3, Invisible Things Lab, jan 2010. http://media.caballe.cat/2012/09/arch-spec-0.3.pdf [cited December 12, 2012].

[RX10] Junghwan Rhee and Dongyan Xu. LiveDM: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging. Technical report, Purdue University, 2 2010.

S

[S_4epen08] ShadOS and _4epen. IntoXonia-NG rootkit, 2008. http://toolbase.blogspot.de/2008/06/intoxonia-2-lkm-rootkit-for-linux.html [cited December 20, 2012].

[SG08] Abhinav Srivastava and Jonathon Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *Recent Advances in Intrusion Detection*, volume 5230 of *Lecture Notes in Computer Science*, pages 39–58. Springer, Berlin, Heidelberg, 2008.

[SH97] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 1–14, New York, NY, USA, 1997. ACM.

[Sha07] 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, New York, NY, USA, 2007. ACM.

[Sha08] Amit Shah. Kernel-based virtualization with kvm. *Linux Magazine*, 86:37–39, 2008.

[She09] Alisa Shevchenko. Case study: the TDSS rootkit. *Virus Bulletin*, 5:10–14, May 2009.

[SLCL09] Monirul Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2009. ACM.

[SLQP07] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[SN05] James E. Smith and Ravi Nair. *Virtual Machines.* Elsevier Ltd., 2005.

[SPE11] Christian Schneider, Jonas Pfoh, and Claudia Eckert. A universal semantic bridge for virtual machine introspection. In *Information Systems Security*, volume 7093 of *LNCS*, pages 370–373. Springer, 2011.

[SPE12] Christian Schneider, Jonas Pfoh, and Claudia Eckert. Bridging the semantic gap through static code analysis. In *Proceedings of EuroSec'12, 5th European Workshop on System Security.* ACM Press, April 2012.

[SSG08] Abhinav Srivastava, Kapil Singh, and Jonathon Giffin. Secure observation of kernel behavior. Technical Report GT-CS-08-01, Georgia Institute of Technology, 2008.

[St10] Richard M. Stallman and the GCC developer community. *Using the GNU Compiler Collection*, chapter 6, pages 309–310. GNU Press, 2010. http://gcc.gnu.org/onlinedocs/gcc-4.7.1/gcc.pdf [cited September 10, 2012].

[Ste] Stealth. adore-ng rootkit. http://stealth.openwall.net/rootkits/ [cited December 12, 2012].

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *The 23rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[SWJX11] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 363–374, New York, NY, USA, 2011. ACM.

[Szo05] Peter Szor. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.

T

[Tan07] Andrew Tanenbaum. *Modern Operating Systems*. Prentice Hall, 3rd edition, 2007.

U

[UNR+05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando Martins, Andrew Anderson, Steven Bennett, Alain Kaegi, Felix Leung, and Larry Smith. Intel virtualization technology. *IEEE Computer Society*, 5:48–56, July 2005.

V

[VE12] Sebastian Vogl and Claudia Eckert. Using hardware performance events for instruction-level monitoring on the x86 architecture. In *Proceedings of EuroSec'12, 5th European Workshop on System Security*. ACM Press, April 2012.

[VH11] Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, New York, NY, USA, 2011. ACM.

[VKSE13] Sebastian Vogl, Fatih Kilic, Christian Schneider, and Claudia Eckert. X-TIER: Kernel module injection. In *Proceedings of the 7th International Conference on Network and System Security*, Lecture Notes in Computer Science. Springer, June 2013.

[Vog10] Sebastian Vogl. A bottom-up approach to VMI-based kernel-level rootkit detection. Diploma thesis, Technische Universität München, October 2010.

[VRMCL08] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[win] Debugging tools for Windows. http://www.microsoft.com/whdc/devtools/ddk/default.mspx [cited September 10, 2012].

W

[WJCN09] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM.

[Woo12] Paul Wood. Internet security threat report. Technical Report 17, Symantec Corporation, April 2012. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_2011_21239364.en-us.pdf [cited July 31, 2012].

[WS12] Sascha Wessel and Frederic Stumpf. Page-based runtime integrity protection of user and kernel code. In *Proceedings of EuroSec'12, 5th European Workshop on System Security*. ACM, 2012.

[WZS11] Jinpeng Wei, Feng Zhu, and Y. Shinjo. Static analysis based invariant detection for commodity operating systems. In *Proceedings of 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 287–296, October 2011.

X

[XJZ$^+$10] Guofu Xiang, Hai Jin, Deqing Zou, Xinwen Zhang, Sha Wen, and Feng Zhao. VMDriver: A driver-based monitoring mechanism for virtualization. In *Proc. of IEEE Symposium on Reliable Distributed Systems*, pages 72–81. New Delhi, India, IEEE, October 2010.

Y

[YHR99] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 91–103, New York, NY, USA, 1999. ACM.

Z

[ZHS+09] Jianwei Zhuge, Thorsten Holz, Chengyu Song, Jinpeng Guo, Xinhui Han, and Wei Zou. Studying malicious websites and the underground economy on the chinese web. In *Managing Information Risk and the Economics of Security*, pages 225–244. Springer US, 2009.