Semantic monitoring mechanisms dedicated to security monitoring in IaaS cloud

Yacine Hebbal

UNIVERSITE BRETAGNE LOIRE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de Doctorat

Yacine HEBBAL

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire
sous le sceau de l'Université Bretagne Loire

École doctorale : Mathématiques & Sciences et technologies de l'information et de la communication

Discipline : Informatique et applications
Spécialité : Informatique
Unité de recherche : Orange Labs Caen & Laboratoire des Sciences du Numérique de Nantes (LS2N)

Soutenue le 18 septembre 2017
Thèse n° : 2017IMTA0029

Semantic monitoring mechanisms dedicated to security monitoring in IaaS cloud

JURY

Président :	**M. Eric TOTEL**, Professeur, CentralSupélec Rennes
Rapporteurs :	**M. Daniel HAGIMONT**, Professeur, INPT Toulouse
	M^me^ Vania MARANGOZOVA-MARTIN, Maître de conférence, Université de Grenoble
Examinateurs :	**M. Eric TOTEL**, Professeur, CentralSupélec Rennes
	M. Frédéric LE MOUËL, Maître de conférences, INSA Lyon
Directeur de thèse :	**M. Jean-Marc MENAUD**, Professeur, IMT Atlantique
Co-encadrante :	**M^me^ Sylvie LANIEPCE**, Ingénieur de recherche, Orange Labs

Acknowledgments

I would like to greatly thank Sylvie Laniepce and Jean-Marc Menaud for supervising me during the past three years and for the countless hours they spent guiding me and discussing the work of this thesis. Without their guidance and insightful comments, this work would not be as good as it is today.

Also, I would like to thank my project manager, Mohamed Kassi-Lahlou, for his attention, support and wise advices during all the different phases of this work.

I would like to thank also Adel Gherbi for working with me as an intern and for sharing with me a part of this difficult journey.

I would like to thank the reviewers of my thesis for their time and for their positive and supporting comments. In addition, I thank all the jury members for accepting to judge my work and for their numerous and fruitful questions during my defense.

I would like to give very special thanks to professor Merouane Debbah for inspiring me and letting me discover and enjoy the world of scientific research in the lab he directed which motivated me to pursue a PhD.

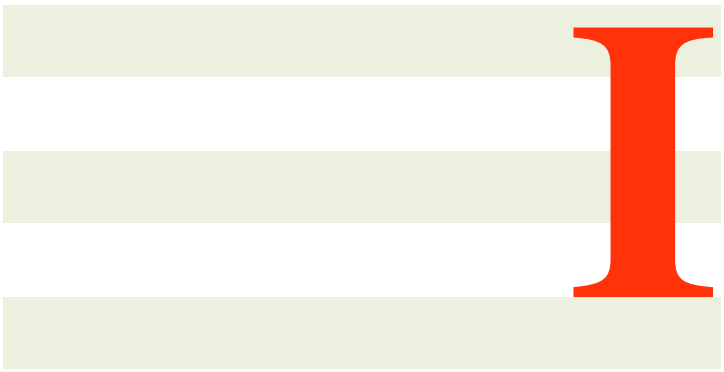
Also, I would like to send my deepest thanks to my wife Inesse, my mother, father, brothers and my close friends Abderrahmane, Youcef, Mehdi and Kamel for their love, infinite support and for being with me in the difficult moments of this journey.

Finally, I would like to thank all my friends and (ex-)colleagues for their support and prayers.

Contents

List of Tables

List of Figures



Context

1

Introduction

1.1 Context

Companies are increasingly migrating their servers and business applications into Virtual Machines (VMs) hosted by providers of cloud computing infrastructures. These infrastructures are attractive because they offer an efficient, flexible and low-cost hardware that may considerably increase performance of the hosted applications. Cloud infrastructures are operated through hardware virtualization technology which divides physical resources (e.g. processor, memory, disk, etc.) in the cloud infrastructure into virtual ones to enable multiple Operating Systems (OSes) to run simultaneously on the same physical hardware thanks to a virtualization layer (called the hypervisor) interposed between the physical resources and the virtual machines.

This massive adoption of virtual machines hosted in remote cloud infrastructures by companies and internet users makes them a privileged target for malware attacks. Virtual machines security is monitored today mostly by traditional security mechanisms such as Host-based Intrusion Detection Systems (HIDS) and Network-based Intrusion Detection Systems (NIDS). These mechanisms provide respectively a good visibility of virtual machines activities and a good isolation (i.e. protection) from them, but neither of them provides both good visibility and a good isolation at the same time.

Hardware virtualization opened the way for a new mechanism of virtual machine security monitoring known as Virtual Machine Introspection (VMI) which consists in monitoring virtual machines from the hypervisor layer. The hypervisor is designed to be isolated from virtual machines and its isolation is enforced by security extensions of the processor on top of which the hypervisor runs (i.e. good isolation). Hypervisor security privileges and its interposition between the monitored virtual machines and the physical hardware enable it to view and inspect virtual machines memory and processor states (i.e. good visibility). Hence, the hypervisor layer combines the good visibility of HIDS and the good isolation of NIDS.

1.2 Problem statement

Despite the advantages offered by the hypervisor for monitoring virtual machines security, the visibility of virtual machine activities and states available to the hypervisor is just raw bits and bytes in memory in addition to hardware states. This visibility does not reflect high level semantic information needed for security monitoring such as abstraction of the OS running inside the VM, process activities, kernel data structures, etc. The difference between semantics of the visibility available to the hypervisor and the one needed for security monitoring is known as the semantic gap. In order to monitor virtual machines security from the hypervisor layer, VMI systems need first to bridge this semantic gap and obtain semantic information on virtual machine activities and states from the raw bits and bytes available to the hypervisor.

1.3 Goals

Several works on cloud-based security addressed the challenge of bridging the semantic gap for VMI systems and proposed a variety of approaches to do so. However, none of these approaches is at the same time automatic, OS independent, secure and reliable, reactive and bridge most of the semantic gap. All these properties are essential in a VMI system designed to be used by real world Infrastructure-as-a-Service (IaaS) cloud operators which are committed to ensure a quality of service all the while hosting thousand of heterogeneous virtual machines running a variety of OS types and versions.

For this reason, the main goal of thesis is to design a new VMI approach for bridging the semantic gap which is applicable to real world IaaS cloud environments running on top of the widely used x86. That is, a VMI approach with an enhanced automation, OS independence, security and reliability and greatly narrow the semantic to protect in real time monitored virtual machines from malicious attacks. Once such technique is designed and the semantic gap is bridged, a second goal of this thesis is to demonstrate through a security use case how the proposed technique can be used to understand virtual machines activities and states to detect and react against real world attacks.

1.4 Contributions

The main contributions of this thesis are:

1.4.1 Technical discussion of major existing VMI techniques and their properties

In this thesis, we first study technically then classify major existing VMI techniques for bridging the semantic gap and their explored applications. For each considered VMI system, we present a technical overview of how it proceeds to bridge the semantic gap for a targeted application then we detail advantages and limitations of this system. We compare after that properties of presented VMI systems and we conclude that when binary analysis and binary code reuse techniques are integrated in a VMI system and applied on a part of virtual machine main kernel code, they considerably help enhancing one or more properties of the VMI system (e.g. efficiently bridging the semantic gap, automation, OS independence, etc.). Based on this conclusion, we explore the idea of applying binary analysis and binary code reuse techniques on all main

kernel code of the virtual machine to obtain a fine-grained understanding of virtual machines states and activities and to enhance at once VMI desired properties for usage in real world IaaS cloud environments.

1.4.2 Hypervisor-based main kernel code disassembler

Our idea of applying binary analysis and binary code reuse techniques on all main kernel code requires first knowledge of location and size of the main kernel code in order to disassemble it and interpret it for binary analysis and binary code reuse based VMI. Location and size of the main kernel code are OS-specific details that are unknown to the hypervisor and they change according to OS type, version and customization. Moreover, when Address Space Layout Randomization (ASLR) mechanism is activated, it changes the kernel location each time the virtual machine is rebooted. Due to these difficulties, it is not obvious how to locate and disassemble the main kernel code from the hypervisor layer without knowing the kernel location and code size. In this thesis, we introduce a new VMI-based mechanism that enables the hypervisor to locate the main kernel code, disassemble it and divide it into code fragments called code blocks. In our main kernel code disassembler, we introduce a novel backward disassembly mechanism that enables the hypervisor to correctly locate instruction boundaries that precede a specified virtual address.

1.4.3 Precise main kernel code version and customization identification

OS-specific addresses such as function addresses can be required when employing binary analysis and binary code reuse techniques for VMI. One way to obtain these addresses is to precisely identify the kernel running inside the VM then use its available kernel symbols files to obtain needed addresses. These addresses change according to kernel version and are different even for two customizations of the same version. Existing approaches for kernel version and customizations are limited in precision and usability due to the difficulty of locating correctly sufficient main kernel code for the identification especially in the presence of compiler optimizations. Hence these existing approaches are not suitable for usage in real world IaaS cloud environments. In this thesis we introduce K-binID, a kernel binary code identification system built on top of our hypervisor-based main kernel code disassembler that enables to precisely identify both version and customization of a virtual machine kernel if it belongs to a set of known kernels. The main contribution of K-binID is that it defines a kernel signature at the level of code blocks which enables precise identification of kernel version and customization at the hypervisor layer despite challenges presented by ASLR and compiler optimizations.

1.4.4 Kernel functions localization and identification

VMI systems based on binary analysis and binary code reuse that require kernel symbols file to bridge the semantic can become unusable in case of an update or a custom patch is applied to a known kernel. To address this case and to support introspection of unknown kernels whose code is similar to known ones, we explore in this thesis the problem of identifying locations and names of main kernel functions in the memory of a running virtual machine from the hypervisor level without necessarily having kernel symbols file of the virtual machine kernel. To do so, multiple challenges should be addressed. The first one is that in the context of VMI, the location of main kernel code is unknown to the hypervisor. The second challenge is

that function start and end addresses are not clear in the binary code and are even blurrier due to compiler optimizations. To address this challenges, we present NoGap, a VMI system built on top of our main kernel disassembler which enables to identify names and addresses of most main kernel functions in the memory of a running VM from the hypervisor level despite challenges presented by compiler optimizations and ASLR. NoGap enables the hypervisor to instrument (e.g. intercept, call and analyze) identified functions to bridge the semantic gap. The main contribution of NoGap is a grouping algorithm that deduces function start and end addresses in addition to their names from properties of code blocks that compose kernel functions.

1.4.5 Global kernel pointers and offsets identification

Some fine-grained kernel details such as process list address are difficult or even impossible to obtain through intercepting or calling main kernel functions. Based on the principle of 'data use tells data semantics', we propose in this thesis a preliminary work on a set of new learning based binary analysis techniques that enables the hypervisor to easily deduce addresses of global kernel pointers (e.g. process list) and field offsets of interest in kernel data structures (e.g. pid field in process descriptor) through instruction analysis of main kernel functions that use these global pointers and field offsets.

1.4.6 Hidden process detection using main kernel functions instrumentation

As a security use case of NoGap, we present at the end of this thesis a new approach of hidden process detection that works automatically on a wide range of Linux kernels thanks to the instrumentation of stable and long-lived functions in the Linux kernel.

1.5 Outline

This thesis is composed of three parts. In the first part, we present the context of our work in chapter 2 in which we present basic notions about the cloud computing paradigm, hardware virtualization and we introduce how virtualization makes possible a new model of virtual machine security monitoring called virtual machine introspection. In chapter 3, we give an overview of virtual machine introspection state of the art in which we present technical details and properties of major existing introspection approaches and their explored applications.

In the second part of this thesis, we present key technical contributions of this thesis. We present in chapter 4 a hypervisor-based main kernel disassembler that enables the hypervisor to locate, disassemble and divide the main kernel code into fragments that we call code blocks. In chapter 5, we present the design of K-binID, a kernel binary code identification system built on top of our main kernel disassembler which enables the hypervisor to precisely identify version and customization of a running kernel inside the VM with the help of hash signatures defined at the level main kernel code blocks. In chapter 6, we present the design of NoGap, a hypervisor-based system that enables to identify locations and names of most main kernel functions in the memory of a running VM so they can be instrumented to bridge the semantic gap for virtual machine introspection.

The third part of this thesis concerns the evaluation of the contributions presented in this thesis in addition to a security use case. In chapter 7, we present a detailed evaluation of each system presented in the second part of the thesis. We present in the first part of chapter 8 how our systems can be used to deduce even more fine-grained details of introspected kernels such as the identification of global kernel pointer and field offset of interest in kernel data structures. In the second part of chapter 8, we present a new automatic and widely OS portable approach for detecting automatically hidden running processes on Linux kernel through the instrumentation of stable and long-lived kernel functions. Finally we the conclusion of this thesis and future perspectives in chapter 9.

1.6 Publications

1.6.1 Patents

FR3038085A1 ; WO2016207533A1 - « **Method for assisting with the analysis of the execution of a virtual machine** » filed in 25th June 2015, PCT extension in 21th juin 2016.

FR3051934A1 ; WO2017203147A1 - « **Method for identifying at least one function of an operating system kernel** » filed in 24th May 2016, PCT extension in 22th mai 2017.

1.6.2 International conferences

Y. Hebbal, S. Laniepce and J. M. Menaud, "**Virtual Machine Introspection: Techniques and Applications**," 2015 10th International Conference on Availability, Reliability and Security, Toulouse, 2015, pp. 676-685.

Y. Hebbal, S. Laniepce and J. M. Menaud, "**K-binID: Kernel Binary Code Identification for Virtual Machine Introspection**", 2017 IEEE Conference on Dependable and Secure Computing (DSC).

Y. Hebbal, S. Laniepce and J. M. Menaud, "**Hidden Process Detection using Kernel Functions Instrumentation**", 2017 IEEE Conference on Dependable and Secure Computing (DSC).

1.6.3 National conferences and workshops

Y. Hebbal, S. Laniepce and J. M. Menaud, "**Semantic monitoring mechanisms dedicated to security monitoring in IaaS cloud**", SEC2 2015 - Premier atelier sur la Sécurité dans les Clouds.

Y. Hebbal, S. Laniepce and J. M. Menaud, "**K-binID: Identification de code binaire noyau à des fins d'introspection de machine virtuelle**", 2017 Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS).

Y. Hebbal, S. Laniepce and J. M. Menaud, "**[Poster] NoGap: Towards Eliminating Kernel Level Semantic Gap**", 2017 Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS). **Awarded as the best system poster at ComPAS conference**.

2

Background

Chapter abstract

The rapid development of efficient yet cheap storage, networking and computing hardware led to the creation of a remote and on demand computing paradigm based on hardware resources sharing called cloud computing. In this chapter we introduce basic notions and an overview of this computing model. We introduce after that hardware virtualization technology which operates cloud computing infrastructures and enables user environments called virtual machines to run concurrently on the shared physical resources. Hardware virtualization technology ensures secure access to allocated physical resources for each virtual machine, but it does not protect them against all security threats. So we cover at end of this chapter possible mechanisms for monitoring virtual machines security and we argue for our interest in new security monitoring opportunities provided by the virtualization technology.

2.1 Introduction

A cloud computing infrastructure hosts multiple virtual machines that belong to different users and run simultaneously on a shared physical hardware. The cloud computing model is possible thanks to hardware virtualization technology which enables to combine or divide physical resources and present them as virtual resources to operating systems running inside virtual machines. Similarly to non virtualized environments, operates systems and applications running inside virtual machines need to be protected from malicious activities and users using a security monitoring system.

In this chapter, we start by presenting the different notions related the cloud computing model. Then we present hardware virtualization technology and its possible implementation techniques. Finally, we describe and discuss existing mechanisms of virtual machines security monitoring.

2.2 Cloud computing

In this section, we present a definition of cloud computing, its essential properties and service models.

2.2.1 Definition

Cloud computing is defined by the National Institute of Standards and Technology (NIST) [95] as new consumption model of distant shared computing resources. This model enables ubiquitous and on-demand network-based access to a shared pool of configurable computing resources such as networks, servers, storage, applications, and services.

2.2.2 Characteristics

The NIST defines five essential characteristics of the cloud computing model: on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service [95].

On-demand self-service: a user can automatically add and remove computing service (e.g. time server) without any human interaction with the service provider. Broad network access: user computing services are accessible over the network through standard mechanisms from a wide range of heterogeneous client platforms (e.g. laptops, mobile phones, tablets).

Resource pooling: the cloud provider is able to transparently aggregate – possibly heterogeneous – resources and present them as resource pools that can be dynamically assigned to consumers according to their demands.

Rapid elasticity: computing resources allocated to a consumer can be rapidly and automatically increased and decreased according to the needed computing capacities which can be variable in time.

Measured service: an automatic measuring system is deployed by the cloud provider to quantify, control and report a consumer consumption of allocated computing resources.

2.2.3 Service models

The NIST defines three service models for cloud computing according to what layer is managed by the cloud provider: software-as-a-service, platform-as-a-service and infrastructure-as-a-service [95]. Figure 2.1 illustrates these three service models and specifies which layers are managed by the cloud provider (yellow layers) and which ones are managed by service consumer (green layers).

Software-as-a-Service (SaaS)

The cloud provider offers a computing capacity service in the form of a software application that is running on the cloud infrastructure and accessible by a consumer client interface such as a web browser. The software service belongs to the cloud provider which manages the underlying operating system, network and storage capacities. The consumer can manage only limited user-specific settings of the provided software service.

Google Drive [11] storage service, Microsoft Office 365 [20] documents edition service and Gmail [9] email service are examples of SaaS cloud service model.

Platform-as-a-Service (PaaS)

The cloud provider offers a software development environment to consumers that are software developers. This environment includes an operating system, programming-language execution environment and APIs that simply software development and deployment. The cloud provider manages the underlying cloud infrastructure including network, servers, operating systems, and storage while the consumer manages configuration settings of the environment hosting the application.

Google App Engine [10] and IBM Smartcloud Application [12] services are examples of PaaS cloud service model.

Infrastructure-as-a-Service (IaaS)

The cloud provider offers fundamental computing capacities such as processor (CPU), memory, storage ... possibly in the forms of VMs on which the consumer can deploy and run arbitrary software including applications and operating systems. The cloud provider manages only the underlying cloud infrastructure while the consumer fully controls the operating systems and the deployed applications.

Amazon Elastic Compute Cloud (EC2) [1] Windows Azure [18] and Google Compute Engine [4] are examples of IaaS cloud service model.

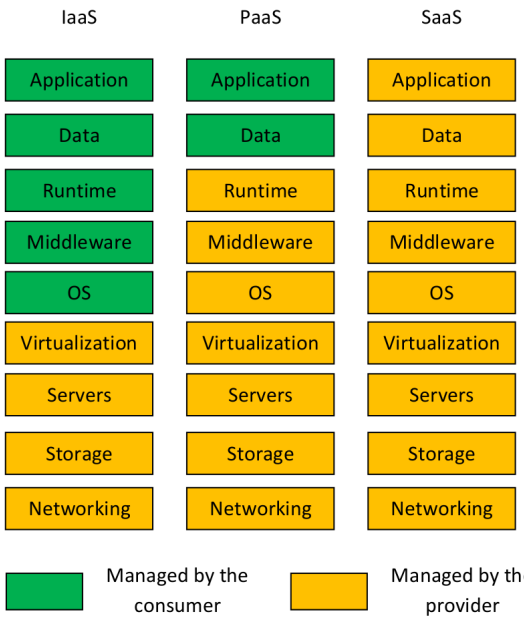


Figure 2.1 – Cloud service models defined by the NIST

2.2.4 Deployment models

The NIST defines four deployment models for cloud computing infrastructures according the kind of consumers that use them. These deployment models are: private cloud, community cloud, public cloud and hybrid cloud.

Private cloud

The provided cloud infrastructure is used exclusively by a single organization that may comprise multiple users (business unit, government agency ...). A private cloud can be hosted, owned and managed by the user organization, a third party organization, or a combination of them. The main advantage of a private cloud is the security of the user organization data as they are hosted in an infrastructure dedicated to the user organization.

Community cloud

The provided cloud infrastructure is used exclusively by a community of consumers that share common concerns (e.g. security). A community cloud can be hosted and managed by the user community or a third party organization.

Public cloud

The provided cloud infrastructure is used by a general public that comprises multiple kinds of consumers ranging from companies to private individuals. A public cloud is hosted and managed by the cloud service provider. Hybrid cloud: The cloud infrastructure in this case is a combination of distinct cloud infrastructures (private, community, public) that are linked together by a technology that enables data and application portability among them. The main technology that enables the cloud computing paradigm and provides it with its essential properties is hardware virtualization which we present in the next section.

2.3 Hardware virtualization

In this next section, we present an overview of hardware virtualization and its existing techniques.

2.3.1 Definition

Hardware virtualization is a technique that enables to combine or divide physical hardware resources (e.g. processor, memory, disk, etc.) into virtual ones that can be used concurrently by virtual environments called Virtual Machines (VMs). Hardware virtualization is called emulation in case the architecture of virtual hardware presented to virtual machines is different from the one of the physical hardware. Virtualization appeared in the 60s as a technology that divides a physical main frame computer into isolated virtual machines instances accessible concurrently by multiple users [111]. The first formal models that defines virtual machines and virtualizable hardware architectures were proposed by Golberg and Poepk in the 70s [107]. The use of virtual machines disappeared in the 80s with the widespread of personal computers (PCs) that are equipped with powerful and cheap hardware in addition to multi-task and multi-user operating systems (OSes). In the 90s, it was found that computer hardware resources (e.g. processor, memory, disk, etc.) are greatly underused and idle most of the time [111]. So, hardware virtualization was again considered to optimize resource consumption by dividing computer hardware resources into virtual ones so they can be fully and concurrently used by heterogeneous systems running inside virtual machines.

Hardware virtualization is today the key technology that operates cloud computing infrastructures and enables multiple virtual machines to run simultaneously on the same physical hardware thanks to a virtualization layer called the hypervisor or Virtual Machines Manager (VMM).

The hypervisor is a software layer that divides physical resources (e.g. processor, memory, disk, etc.) and presents them to virtual machines as virtual resources (e.g. virtual processor, virtual memory, virtual disk, etc.). The hypervisor is interposed between virtual machines and the underlying physical resources and responsible for hosting and scheduling execution of the virtual machines on their virtual resources in a way that each virtual machine is strongly isolated from the other ones as if they were on separated physical machines.

2.3.2 Hypervisor types

As illustrated in figure 2.2, hypervisors are classified into two types according to whether the hypervisor is running directly on the physical machine (type 1) or as application or a service on operating system (type 2)

Type 1

A hypervisor of type 1 is a called a bare-metal hypervisor as it that runs directly on the physical machine. A bare-metal hypervisor implements all major operating systems features in order to manage the hosted virtual machines and their operating systems needs (called guest OSes). Xen [38], VMware ESXi [27] and Microsoft hyper-V [19] are examples of bare-metal hypervisors.

Type 2

A hypervisor of type 2 runs as an application or a service on operating system called the host OS. A hypervisor of type 2 may benefit from host OS features to manage virtual machines and their guest OSes. KVM [84], VirtualBox [23] and VMware Workstation [28] are examples of type 2 hypervisors.

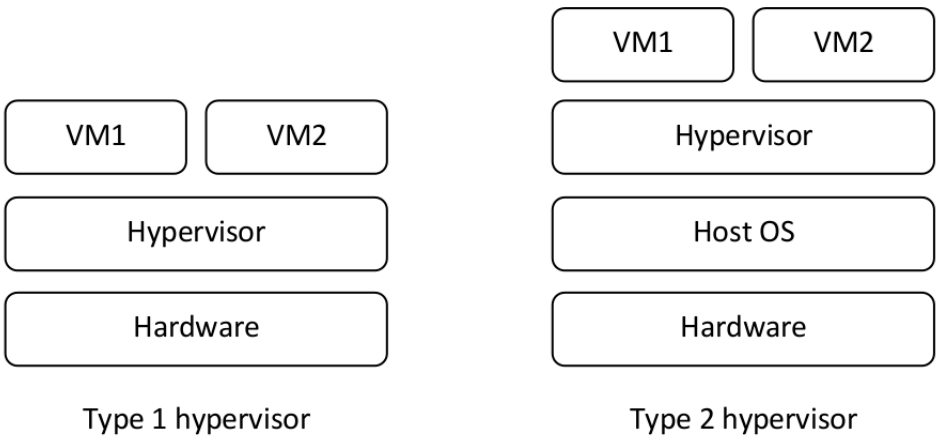


Figure 2.2 – Hypervisor types

2.3.3 Virtualization techniques

Hardware virtualization can be implemented using different approaches based on trap and emulate technique to isolate virtual machines and to enable them to run simultaneously on the same physical hardware. The key idea of trap and emulate technique is to execute the hypervisor on the highest processor privilege level and virtual machines on a processor privilege level that is lower than the required one. This way, the CPU traps VM sensitive instructions which are instructions that impact VMs isolation as they access shared physical resources (e.g. input/output, etc.), read or modify their states. For each trapped instruction, the CPU raises an exception that is forwarded to the hypervisor. The hypervisor inspects the trapped instruction and ensures that this instruction does not try to access resources allocated to another VM to enforce virtual machines isolation. The hypervisor emulates (i.e. executes) the trapped instruction then resumes VM execution as if no exception was raised. Non sensitive instructions do not impact VMs isolation and are executed directly on the CPU without hypervisor intervention.

We present next the main techniques for hardware virtualization:

Full virtualization

Full virtualization consists in using the trap and emulate mechanism to execute virtual machines on virtual resources without prior adaptation of their guest operating systems to the underlying virtual environment. The widely used x86 architecture was considered as non virtualizable for many years [30] because some of its sensitive instructions can not be traped to the hypervisor as they raise no exception when they are executed in a deprivileged level. Full virtualization was brought to the x86 architecture by VMware using a binary translation technique [46]. This technique consists in rewriting on the fly some of the guest OS binary code to replace non virtualizable sensitive instructions with an instruction sequence that can be trapped then emulated by the hypervisor to achieve the desired effect on the virtual hardware.

The main advantage of full virtualization is the transparent execution of legacy operating systems on a legacy hardware. Transparency in this context means that the guest OS runs natively on the virtual hardware with no prior adaptation and that the guest OS is not aware that it is running on a virtual environment. However, trapping and emulating all sensitive instructions of the guest OS in addition to the binary rewriting process impact significantly performance of the hosted virtual machines.

VMware workstation [28] and Qemu [40] are examples of full virtualization solutions.

Paravirtualisation

Full virtualization induces a significant performance loss due to the use of trap and emulate mechanism to transparently execute non modified guest OSes. To reduce this performance loss, the idea of paravirtualisation is to rewrite parts of guest OS source code to optimize it for execution in a virtual environment. In a paravirtualisation model, guest OS code parts that contain sensitive instructions are rewritten and replaced with an optimized code that explicitly calls the hypervisor to handle operations performed by the sensitive instructions. These calls to the hypervisor added in guest OS source code are named hypercalls and their goal is to remove the need for expensive sensitive instructions trapping. While sensitive instructions trapping is replaced by hypercalls in a paravirtualisation model, the hypervisor remains responsible of physical

resource allocation management and isolation of the hosted virtual machines.

The main advantage of paravirtualisation techniques is the enhancement of hosted virtual machines performance while their big limitation is the difficulty of porting closed-source operating systems to a paravirtualisation hypervisor.

Xen [38] is a famous hypervisor that uses a paravirtualisation technique.

Hardware-assisted virtualization

The use of virtualization greatly increased with the widespread of cloud computing infrastructures and services. This attracted hardware vendors such as Intel and AMD to introduce hardware extensions called Virtual Machine Extensions (VMX) to natively support virtualization in their x86 processors [57] [33]. In these processors with virtualization extensions, there is two execution modes: a privileged hypervisor mode called vmx-root in which the hypervisor runs on the highest privilege and a less privileged virtual machine mode called vmx-non-root in which the guest OS of the virtual machine runs on the highest privilege level of this mode.

In hardware-assisted virtualization, some sensitive instructions of the guest OS (i.e. vmx-non-root mode) are handled automatically within the processor and do not generate traps to the hypervisor. The rest of sensitive instructions are automatically trapped by the CPU which switches then to vmx-root mode to give control to the hypervisor in order to handle the trapped sensitive instruction. The switching from vmx-non-root to vmx-root mode is called a VM exit while the switching from vmx-non-root to vmx-root is called VM entry.

Virtualization extensions bring also support for enhanced memory virtualization by facilitating management of virtual machines memory and accelerating access to virtualized memory. More precisely, hardware virtualization extensions provide the hypervisor with hardware mechanisms to easily manage and protect memory page tables of virtual machines like the guest OS manages and protects page tables of running processes. Also, to accelerate memory access and hence the overall performance of virtual machines, the processor walks automatically through guest OS pages tables then hypervisor pages tables for each memory access (e.g. read, write, execution) to translate guest virtual addresses into guest physical address using guest OS managed pages tables, then to host physical addresses using hypervisor managed pages tables. For secure and reliable memory virtualization, page tables managed by the hypervisor are invisible and inaccessible to virtual machines. These page tables are called Extended Page Tables (EPT) by Intel [57] and Nested Page Tables (NPT) by AMD [33] in their hardware virtualization extensions.

Hardware virtualization extensions simplify a lot the development of hypervisors and enables to run non modified guest OSes with a highly improved performance compared to other virtualization techniques.

From now on, we assume that virtual machines are hosted in an IaaS cloud environment that operates with hardware-assisted virtualization on the widely used x86 architecture. In the next section, we present main technique for monitoring the security of virtual machines.

2.3.4 Virtual machine security monitoring

Virtualization today is widely used to execute simultaneously non modified legacy operating systems in isolated virtual machines that are assimilated to distinct physical machines. The goal of isolation is to ensure that virtual machines do not access maliciously or accidentally to resources (e.g. disk and memory) that are not allocated for them such as hypervisor and other VM resources. The hypervisor is responsible for ensuring isolation in the cloud by managing access to allocated resources for each virtual machine. While inter-VM security (i.e. isolation) is ensured by the hypervisor, intra-VM security is left to the responsibility of virtual machine users. Intra-VM security consists mainly in protecting VM guest OS and applications running on top of it from intrusions and malicious activities using an intrusion detection system. From now on, we will use the term VM security to refer to intra-VM security.

Intrusion detection systems

An Intrusion Detection System (IDS) is defined by the NIST as a system that monitors and analyzes automatically activities of a computer system (e.g. virtual machine) or network to detect signs of unauthorized activities which violate defined computer security policies [120]. An IDS operates generally in three main phases where the first phase (called information collection phase) consists in collecting information about activities of the monitored system. When this phase is carried without stopping monitored system activities, the monitoring system is called a passive IDS. In the opposite case, it is called an active IDS. In the second operating phase (called a decision phase), the IDS analyzes the collected information about system activities and checks whether they respect the defined security policies or not. In case a security policy violation is detected, the IDS activates the last phase (called a reaction phase), reports an alert that may be sent to a security administrator and finally triggers suitable counter measures. For instance, the activated counter measures can be the shutdown of the unauthorized network connection or stopping the application that performed the malicious activity. An IDS is called an Intrusion Prevention System (IPS) when its goal is to react to unauthorized activities before they impact the monitored system.

Existing intrusion detection techniques can be classified into multiple categories according to several criteria [120] [59][89]. We present next two possible classifications of intrusion detection systems according to their intrusion detection method and deployment level.

Intrusion detection methods

Signature-based This approach consists in comparing information collected during execution of the monitored system against a set of patterns (called signature database) that corresponds to known threats. A threat signature can be for instance a special sequence in the content of a file or a network packet. When information collected during execution of the monitored system match against a signature of known threat, the intrusion detection system triggers an intrusion alert and reacts according to the detected threat. Signature-based intrusion detection systems are efficient for detecting threats whose signatures are in the database of known threats. However, this detection method is ineffective against evasion techniques that produce different variants of a same threat and whose signatures are different from the one known to the detection system.

Behaviour-based Behaviour-based intrusion detection is sometimes referred to in the literature as anomaly based detection [59]. The idea of behaviour-based intrusion detection is to construct a model that defines the normal (i.e. authorized) behaviour of the monitored system and then consider each behaviour deviation from that model as a threat. The main advantage of behaviour-based intrusion detection is the possibility of detecting and countering non seen threats that do not respect the authorized behaviour. However, due to the complexity of monitored systems and to the huge number of possible behaviour combinations, this intrusion detection method suffers from a high rate of false positives. That is, a high number of non malicious behaviours is considered as a threat because of the difficulty of construing a complete and reliable model of non malicious behaviours.

We present here after a second classification of intrusion detection systems depending at which level of the computing infrastructure they are deployed to apply the presented detection methods.

IDS deployment levels

In this second classification, there are three intrusion detection approaches in which two are traditional and used in non virtualized environments. These traditional approaches are host-based and network-based intrusion detection systems. In virtualized environments, interposition of the hypervisor between virtual machines and physical resources provides a new opportunity for security monitoring and makes possible a third approach called hypervisor-based monitoring.

Host-based Intrusion Detection Systems Host-based Intrusion Detection System (HIDS) is software agent (e.g. antivirus) that runs inside a host machine and monitors only activities of this latter one. A HIDS can monitor a variety of host machine activities such as process execution, system calls, file access, network connections, system logs, etc. In cloud infrastructure, a HIDS can be deployed inside a virtual machine to monitor its activities. Being inside the monitored machine, the main advantage of a HIDS is that it has a rich and complete view of the monitored activities. However, because the HIDS runs inside the same environment that it should protect from threats, the HIDS itself can be attacked and compromised due to the weak isolation between the attack surface and the HIDS.

Network-based Intrusion Detection Systems Network-based Intrusion Detection System (NIDS) is a physical or virtual device that is deployed at the network level to monitor network activities of multiple connected machines. A NIDS intercepts and analyzes network packets of monitored machines to detect patterns and behaviours that characterize known attacks (e.g. denial of service, malicious code, etc.) or violate defined security policies. Being at the network level makes a NIDS tamper-resistant thanks to the strong isolation between the NIDS and the attack surface and enables a single NIDS device to protect multiple machines at once. The main limitation of network-based intrusion detection systems is that their visibility of monitored machines activities is limited to the network traffic. Hence, a NIDS cannot detect non network attacks and security policies violation that are executed inside a monitored machine. For instance, a NIDS cannot detect a malware that is executed from an infected USB disk. Another limitation is that efficiency of NIDS becomes limited in case the monitored network traffic is encrypted as the content of network packets becomes non understandable by the NIDS.

Hypervisor-based intrusion detection systems Virtualization makes possible a new form of security monitoring in cloud environments which consists in monitoring states and activities of virtual machines

from the hypervisor level. Hypervisor-based intrusion detection is generally referred to in the literature as Virtual Machine Introspection (VMI). Virtual Machine Introspection was introduced for the first time by Garfinkal and Rosenblum in 2003 as a hypervisor-based intrusion detection technique that combines visibility of HIDS and isolation of NIDS [70]. VMI systems inherit isolation, inspection and interposition properties of the hypervisor:

Isolation: in a virtualized environment, the hypervisor runs in the highest privilege level on the processor to protect itself such that virtual machines cannot accidentally or maliciously tamper the hypervisor or its resources. Hence, a VMI system runs also on the highest privilege level and it is protected from the threats inside virtual machines.

Inspection: hypervisor privilege level enables VMI systems to directly access and modify all virtual machines resources such as CPU registers, memory and disk content. This offers VMI systems a complete view of virtual machines states and activities.

Interposition: as the hypervisor is interposed between virtual machines and the underlying physical resources, it allows VMI systems to intercept and analyze actively virtual machine activities of interest for security monitoring.

Despite that VMI systems have interesting properties for security, they face a significant challenge since the hypervisor view of virtual machine states and activities is raw bits and bytes in addition to hardware states. Security monitoring systems require an understandable and semantic view of virtual machine states and activities such as OS abstraction, data structure addresses and definition, process information, system calls, etc. The difference between the semantic view needed for security monitoring and the binary raw view available at the hypervisor level is called the semantic gap challenge [50]. To monitor virtual machines and protect them from threats, a VMI system needs to reconstruct a semantic view of virtual machines states and activities from the available binary view. That is, it needs to bridge the semantic gap.

	Visibility	Isolation	Centralization
HIDS	strong	weak	weak
NIDS	weak	strong	strong
VMI	strong but semanticless	strong	strong

Table 2.1 – Comparison between intrusion detection systems based on their deployment level

Table 2.1 recaps properties of intrusion detection systems classified according to their deployment level in the computing infrastructure. HIDS provides a rich and complete view of states and activities of the protected machines. However, because it runs in the environment where the threats operates, a HIDS can be attacked and compromised. In addition, the HIDS agent must be deployed in each machine in the infrastructure to protect it which increases hardware resources consumption.

Contrary to a HIDS, NIDS is tamper-resistant as it runs outside monitored machines at the network level and protects all machines connected inside that network. NIDS visibility of monitored machine activities is weak because it is limited only to network traffic.

Finally, a VMI system has a visibility of HIDS in addition to isolation and centralization of NIDS which makes VMI an interesting approach for security monitoring in the cloud. However, the visibility of a VMI system is semanticless and the semantic gap should be first bridged in order to understand VM activities for security monitoring.

2.4 Conclusion

In this chapter we gave an overview of cloud computing technology in which we presented characteristics of cloud infrastructures, the different service models they provided and finally the possible deployment models depending on cloud users needs. We also presented in this chapter the concept of hardware virtualization which is the technology that operates a cloud infrastructure and enables to run simultaneously multiple operating systems on a shared physical hardware. Moreover, we described different approaches that can be used to implement hardware virtualization and the properties of each approach. Finally, we discussed intrusion detection techniques and how hardware virtualization provides an new form of virtual machine security monitoring called virtual machine introspection. VMI is a promising technique for virtual machine security monitoring as it leverages hypervisor properties and combines visibility of HIDS and isolation of NIDS. However, VMI systems face the challenge of the semantic gap due to hypervisor semanticless view of virtual machines. In the next chapter, we describe and classify approaches used by existing VMI systems to bridge the semantic gap. We present also in the next chapter applications explored by existing VMI systems.

3

State of the art

Chapter abstract

Hardware virtualization makes possible a new security monitoring mechanism that leverages the virtualization layer and inherits from the hypervisor strong visibility and isolation from virtual machines. This mechanism called virtual machine introspection faces a significant challenge for monitoring virtual machines security due to the semantic gap between needed information for security monitoring and the available one about virtual machines states and activities at the hypervisor level.

Various approaches were proposed in virtual machine introspection state of the art to bridge this semantic gap and to explore novel hypervisor-based security and non security applications. In this chapter, we technically describe and classify major virtual machine introspection techniques for bridging the semantic gap and their explored applications. We present after that a detailed discussion about strength points and limitations of each described system and its category.

3.1 Introduction

Multiple existing works on virtual machine introspection addressed the problem of bridging the semantic gap using different techniques. They also presented proof-of-concepts of several VMI-based security and non security applications. In this chapter we present technical insights and a classification of major VMI techniques for bridging the semantic gap. This chapter is organized as following: we propose first a classification of existing VMI techniques that bridge the semantic gap. In this classification we provide examples of systems that belong to each proposed category and we give technical insights of how they work. We present after that security and non security applications explored by existing VMI systems.

3.2 VMI approaches for bridging the semantic gap

Several existing works classified VMI techniques [39] [104] [123] according to multiple criteria. In this section we refine Pfoh et al. taxonomy [104] to classify existing VMI techniques according to from where and how the semantic information is obtained to bridge the semantic gap into four main categories: in-VM, out-of-VM-delivered, out-of-VM-derived and hybrid [75]. In-VM techniques avoid the semantic gap problem by using an in-VM agent (software application, hook, hook handler, etc.) that cooperates with the hypervisor to obtain semantic information on VM activities and states. On the other hand, out-of-VM techniques are considered as real VMI operate exclusively at the hypervisor level and bridge the semantic gap using semantic information that is delivered by a security administrator (kernel symbols, technical reference, etc.) or derived from the hardware states.

3.2.1 In-VM

In-VM VMI techniques employ a software agent inside an introspected virtual machine. This In-VM agent is responsible from exposing virtual machine activities and states to the hypervisor. The role of the hypervisor is to protect the in-VM agent and enforce desired security policies for the introspected VM. Since in-VM VMI systems avoid the semantic gap challenge, their goal is to propose a secure architecture of coordination between the hypervisor and the in-VM agent.

Lares [101] is an in-VM VMI architecture in which the hypervisor injects hooks inside the guest OS code to detect events (e.g. system calls) of interest for monitoring. For each injected hook, the hypervisor injects in the guest OS a handler that is called by its corresponding hook each time a monitored event is triggered. Hook handlers extract information about monitored events in the VM and communicate them to the hypervisor using hypercalls. The hypervisor forwards extracted information about triggered event to VMI security applications which decide if the triggered event respect security policies or not. To protect injected hooks and their handlers from in-VM attacks, the hypervisor write protects memory pages that contain them.

SIM [118] is an in-VM VMI architecture is very similar to Lares. The main difference between them is that in SIM architecture, hook handlers log information about monitored activities in a secure memory that is shared by the VM and the hypervisor. Security monitoring applications at the hypervisor level can access this shared memory space to retrieve information about monitored activities In SIM architecture, the hypervisor protects hooks, their handlers and the shared memory space.

The usage of an in-VM agent to avoid the semantic gap makes in-VM VMI approaches too OS specific and exposes them to in-VM attacks similarly to HIDS. For these reasons, in-VM VMI approaches did not gain a considerable interest in research on VMI. Out-of-VM VMI systems operate exclusively at the hypervisor level and use delivered or derived semantic information to bridge the semantic gap. We present hereafter an overview of VMI techniques that use delivered semantic information to bridge the semantic gap.

3.2.2 Out-of-VM-delivered

Out-of-VM delivered VMI techniques include principally early and passive VMI systems in which semantic information is explicitly delivered to the VMI system to bridge the semantic gap. Semantic information about guest OS internals, location and definition of kernel data structures of interest is: i) integrated manually in the VMI system, ii) extracted from OS source code or a technical reference or, iii) obtained from available kernel symbols.

LiveWire [70] is the first VMI system which was proposed by Garfinkal and Rosenblum in 2003 for hypervisor-based intrusion detection. To bridge the semantic gap, LiveWire uses a hypervisor-adapted version of crash [16], a Linux kernel dump analysis tool that can locate and interpret Linux kernel data structures with the help of debugging information. Periodically, LiveWire uses crash to obtain at the hypervisor level information about guest OS states and activities (network connections, running processes, etc.). Obtained information about guest OS states and activities at the hypervisor level is called a trusted view. LiveWire compares them with information retrieved by OS native tools (e.g. ps, ifconfig, netstat, etc.) through a remote shell connection (called untrusted view). Inconsistencies between the two view indicate the presence of an intrusion in the VM.

To bridge the semantic gap, **VMwatcher** [78] uses available kernel symbols to locate VM data structures of interest for monitoring (i.e. process list head, etc.) and kernel source code to extract definition (i.e. template) of these data structures. By having location and definition of data structures of interest, VMwatcher interprets raw bits and bytes in VM memory and constructs a semantic view of VM states (e.g. running processes, loaded modules, opened files, etc.). VMwatcher can also interpret VM virtual disk with the help of delivered information about VM disk file system. Constructed semantic view of VM states is exposed to security systems (e.g. antivirus) on the host OS which transparently inspects them to detect possible intrusions. VMwatcher can also detect rootkits and hidden malware processes by comparing hypervisor trusted view of VM states with the untrusted view obtained with OS native tools.

Payne et al. proposed a VMI library for Xen hypervisor [38] called **XenAccess** [102] which facilitates access to a VM memory during the implementation of VMI tools. XenAccess enables VMI systems to map (i.e. get a pointer to) a memory page of an introspected VM that contains i) a given kernel symbol, ii) a given kernel virtual address or, iii) a given virtual address in the memory space of a process specified by its PID. XenAccess extracts kernel symbols from a delivered system.map file for Linux and from debugging libraries from Windows guest OSes.

To access memory space of a given process, XenAccess locates first the data structure that describes that process (i.e. process descriptor) then its page directory and finally the page the contains the desired virtual address. The descriptor of the desired process is located by walking through descriptors in guest OS process list until reaching the descriptor that contains the PID of the desired process. XenAccess locates guest OS process list using a kernel symbol (e.g. init_task in Linux and PsActiveProcessHead in Windows) and interprets its descriptors using information extracted from kernel source code for Linux or a technical reference for Windows [113].

Based on XenAccess, Payne created an enriched library called **LibVMI** [100] that supports KVM hypervisor [84] in addition to Xen and enhances XenAccess performance. LibVMI enriches XenAccess by integrating Volatility [29], an open source memory forensics framework that offers hundreds of tools for

analyzing memory contents. This allows VMI systems built on top of LibVMI to benefit from memory analysis capabilities provided by Volatility.

Dolan-Gavitt et al. advanced considerably VMI state of the art with **Virtuoso** [63] by automating both semantic gap bridging and the generation of VMI tools. Virtuoso operates in two phases: a training phase (i.e. offline) and an introspection phase (i.e. online). In the offline phase, Virtuoso runs a training program that performs a desired inspection task such as obtaining PID of the current running process. Virtuoso records then executed instructions of the training program and generates a binary code that contains them. In the online phase, the recorded instructions are replayed from a security VM (or the hypervisor) on the memory of a monitored VM to reproduce the learned inspection task. Generated inspection programs by Virtuoso can be used to monitor only a guest OS that is similar to the one on which the training program was executed.

Out-of-VM delivered VMI techniques use semantic information that is too OS-specific to bridge the semantic gap and require a manual effort to obtain them. Consequently, this limits their practicality as semantic information of a given OS are generally unusable on other OS types and versions (e.g. after OS update or for an unknown OS). In addition, providing manually semantic information for a VMI system to monitor VMs in real cloud IaaS environments is not realistic because of the big number of different OS versions running in the VMs. Also, the passive and periodic nature of most of out-of-VM delivered VMI systems expose them to evasion attacks where a malware can perform its malicious activities between two security scans without being detected [77] [129]. Finally, because out-of-VM delivered VMI techniques assume that a monitored guest OS exposes trusted information about its states and activities, they are vulnerable to Direct Kernel Structures Manipulation (DKSM) [36] and Dynamic Kernel Object Manipulation (DKOM) attacks [5]. DKSM attacks modify kernel code to alter syntax or semantics of data structures fields to make obsolete delivered semantic information to VMI systems. In this case, VMI system view of manipulated guest OS data structures becomes incorrect while the monitored VM continues to run and access correctly these data structures. DKOM attacks on the other hand target kernel data structures that change dynamically in size, length, etc. over time. For instance, a rootkit can remove a malicious process descriptor from process list but leave it in the scheduler list. This way, the malicious process will continue to run correctly but will be invisible both in VMI and in-VM views.

To avoid weaknesses of out-of-VM delivered VMI, a second category of VMI systems relies exclusively on the hardware architecture to derive information about VM states and activities.

3.2.3 Out-of-VM-derived

Today hardware architecture provide mechanisms that assist operating systems functionalities such as multi-tasking, memory management and protection, etc. These hardware mechanisms improve operating systems performance and security and simplify OS design. The idea behind out-of-VM derived VMI systems is to monitor states and events reflected in the hardware architecture to derive semantic information about VM states and activities. This makes VMI systems OS-agnostic as they rely only on the knowledge of hardware architecture and resistant to malware evasion as each VM activity is executed and reflected by the processor. Out-of-VM derived VMI systems operate either by handling traps (i.e. vm exits) triggered

natively by guest OS activities (trap handling based) or by forcing traps on VM activities of interest using hardware-based hooks (trap forcing based).

Trap handling based

VMI systems in this category derive semantic information about VM states and activities when the hypervisor handles traps (i.e. vm exits) that are triggered natively by VM activities due to the execution of sensitive instructions. In hardware-assisted virtualization some sensitive instructions do not trap to hypervisor and are handled directly by the processor. The hypervisor can configure the processor to raise a trap for these sensitive instructions by setting corresponding control bits in the Virtual Machine Control Structure (VMCS) [57].

Antfarm [80] is the first VMI system that proposed to derive semantic information about VM activities from hardware states. Antfarm tracks process switches and creation by handling traps caused by writes to CR3 register and observing values to be written in that register. CR3 register is used in the x86 architecture to point to page directory used to manage the address space of the currently running process [57]. So the value of CR3 register can be considered as hardware-level process identifier. Antfarm detects creation of a new process when it observes a new value to be written in CR3 register. Otherwise, Antfarm concludes that the VM is about to schedule an existing process whose page directory address is about to be written to CR3 register. Antfarm identifies a process termination when all pages in page directory pointed to by CR3 of that process become marked as non present. This way, Antfarm maintains a hypervisor view of VM processes only by observing traps triggered by the processor.

Antfarm authors proposed later **Lyscosid** [81], a VMI system that extends Antfarm to detect execution of hidden running process in monitored VMs by comparing a hypervisor-level and an in-VM process lists. Lycosid detects the existence of a hidden running process if the two process lists are not equal in length. In this case, Lycosid identifies the hidden running process by correlating CPU consumption time of each process in both process lists.

Trap forcing based

To expand the semantic view that can be derived from hardware states, trap forcing based VMI systems use hardware mechanisms such as breakpoints and exceptions to intercept VM activities of interest (e.g. system calls) that natively do not trap to the hypervisor.

Ether [61] is the first VMI-based malware analysis system that exploits hardware architecture to transparently trace system calls invoked by an analyzed malware and to detect and extract malware dynamically generated code. To intercept system calls (syscalls for short) invoked by the analyzed malware, Ether saves the value of sysenter_eip_msr register (system call handler) then replaces it with null. This forces the processor to raise a trap on each syscall invoked by the malware. Information about the invoked syscall (i.e. number and parameters) can be retrieved from CPU context (i.e. registers). Regarding detection and extraction of dynamically generated code, Ether achieves this in three steps: first, it tracks modifications of malware memory pages by write protecting them to force each page modification to trap to the hypervisor. Second, Ether sets modified pages as non executable to trap execution attempts of these modified pages.

Finally, Ether identifies and dumps pages that contain dynamically generated code when their execution causes a trap to the hypervisor. To gain transparency during malware analysis, Ether manipulates hypervisor level pages tables which are transparent and inaccessible to the guest OS. Also, Ether sets CPU debug flag [57] to force a trap to the hypervisor after execution of each instruction in the malware code in order to detect and tamper malware instructions that intent to detect Ether presence.

In the x86 architecture, system calls can be invoked using interrupts mechanism or sysenter and syscall instructions. **Nitro** [105] is a VMI-based syscall tracing system that enables to intercept all system call mechanisms. Nitro intercepts (i.e. forces to trap) syscall invocations and returns made using sysenter and sysexit instructions by replacing sysenter_cs_msr register value with null after saving it. Regarding syscall invocations and returns made with syscall and sysret instructions, Nitro intercepts them by clearing SCE bit in the Extended Feature Enable Register (EFER) which deactivates these instructions. For interrupt-based syscalls, AMD implementation of hardware virtualization extensions [33] enable to natively intercept any interrupt. However, Intel implementation [57] enables to intercept only system interrupts whose number is lower than 32. To intercept interrupt-based syscalls independently of the hardware implementation, Nitro sets interrupts table size in the Interrupt Descriptor Table Register (IDTR) to 32 entries. This way, system interrupts can be trapped natively on both x86 architecture implementation, while the rest of interrupt is forced to trap the hypervisor as their number exceeds the size of interrupts table. For each intercepted syscall, Nitro logs syscall information reflected in CPU registers then emulates this intercepted syscall by reverting changes made in CPU registers then executing the trapped instruction and finally reactivating syscalls interception.

3.2.4 Hybrid

To expand application range and visibility of VMI systems, hybrid VMI approaches use a combination of in-VM, out-of-VM delivered and derived VMI techniques.

Trap forcing based

SecVisor [116] is a VMI system that aims to protect kernel code of a Linux guest OS from malicious executions and modifications. SecVisor requires rewriting kernel bootstrapping and kernel modules loading and unloading in Linux source code – delivered – to make them trigger hypercalls in order to enable the hypervisor to detect execution of these events and to approve kernel code to be loaded/unloaded using hash signatures-based whitelist policy. During execution of the guest OS, SecVisor holds two copies of VM Nested Page Tables (NPTs). A kernel mode NPT in which only pages that contain approved kernel code are marked as executable and a user mode NPT in which only user code is marked as executable. Kernel code is write protected in both NPTs which are synchronized when there is a modification to kernel code (loading/unloading kernel modules). The use of these two NPTs forces a trap to SecVisor – derived – each time the CPU switches between user and kernel modes. SecVisor handles this trap by loaded the NPT that correspond to the mode to which the CPU intends to switch. SecVisor enables this way to protect the guest OS from kernel code injection attacks. However, requiring changes to kernel source code is not practical especially for closed source OSes.

Data redirection / memory shadowing

NICKLE [110] is a VMI system that is built on top of Qemu and that aims to protect VM kernel code from illegal executions and modifications with the help of a shadow memory. Instead of modifying kernel source code, NICKLE uses OS-specific information – delivered – to detect and approve execution of functions that load kernel code into VM memory. NICKLE uses also a hash signatures-based whitelist policy for kernel code approval. Approved kernel code is copied by NICKLE from its location in the standard memory of the VM into a shadow memory invisible and inaccessible from the VM. NICKLE detects that the VM is about to execute a kernel instruction when processor Current Privilege Level (CPL) – derived – is in kernel mode. In this case, NICKLE checks whether the instruction to be executed exists or not in the shadow memory (i.e. belongs to an approved kernel code). If so, NICKLE manipulates guest to host address translation to execute the instruction from the shadow memory. If the instruction to be executed does not exist in the shadow memory, NICKLE detects an illegal code execution and terminates execution of the malicious code by replacing the illegal instruction with return -1.

NICKLE evaluation shows that this approach enables to detect and stop several known rootkits on Linux and Windows that inject malicious kernel code. The implementation of NICKLE on Qemu to intercept all executing instructions is not a practical approach as today IaaS cloud environments use hardware-assisted virtualization and the interception of all instruction executions decreases dramatically virtual machines performance.

VMST [66] is a dual-VM-based introspection system that is built on top of Qemu and that aims to automate both bridging the semantic gap and generation of VMI tools. The idea of VMST is to automatically convert inspection tools of a Secure VM (SVM) into introspection tools of a monitored Guest VM (GVM) that has exactly the same kernel code as the SVM. For instance, execute ps command on the SVM to list running process of the GVM. Having the same kernel code implies that global kernel data addresses (e.g. process list head init_task, etc.) and data structure definitions are similar in both VMs and that only content and addresses of dynamic kernel data such as running processes, opened files, etc. are different. Because the kernel code knows how to locate and interpret dynamic kernel data (e.g. process descriptor) starting from global kernel data (e.g. process head list) and since the SVM and GVM have exactly the same kernel code, VMST idea is to introspect the GVM from the SVM by detecting execution of kernel code (i.e. syscalls) in SVM inspection tools (e.g. ps, ls, getpid, etc.) then redirecting kernel code access to kernel data towards the memory of the GVM instead of the SVM. VMST detects start and end of syscalls execution in SVM inspection tools when it detects execution of instructions such as sysenter, sysexit, int 0x80, iret, etc. During syscall execution in SVM inspection tools, VMST tracks kernel instructions that access to kernel data starting from a global kernel data pointer using a taint analysis technique [97]. VMST implements kernel data access redirection for SVM instructions of interest by translating accessed virtual addresses using page table of the GVM instead of the one of the SVM. This way SVM inspection tools are automatically converted into introspection tools that automatically bridge the semantic gap for GVM introspection thanks to the knowledge integrated in SVM kernel code.

VMST was later extended to build **Exterior** [67], an out-of-VM shell that enables a cloud administrator to introspect, configure and recover a GVM in case of a successful attack with no need to be authenticated on the GVM.

VMST induces a huge performance overhead in the SVM because it runs entirely on a software emulator and uses a heavy taint analysis technique. To reduce the performance loss caused by VMST, **Hybrid-Bridge** [114] proposes a triple-VM-based VMI architecture that comprises two SVMs running exactly the same kernel code as the monitored GVM. The first SVM, called Slow-Bridge, runs a modified version of VMST whose role is to analyze execution of SVM inspection programs and generate meta data about instructions for which kernel data access redirection should be applied. The second SVM, called Fast-Bridge, runs on hardware-assisted virtualization (e.g. KVM), uses Nitro [105] to detect execution of syscalls in inspection tools and meta data already generated by Slow-Bridge to identify with – no online analysis – instructions concerned by kernel data access redirection. The idea of Hybrid-Bridge to introspect the GVM mainly from Fast-Bridge to reduce performance loss in the SVM by avoiding to perform the expensive taint analysis. Hybrid-Bridge switches to Slow-Bridge only when meta data is missing for a given execution of an inspection program. Once the missing meta data becomes available, Hybrid-Bridge switches back to Fast-Bridge. Hybrid-Bridge achieves the same goal as VMST but with an enhanced performance.

Multi-VM-based introspection architectures may not be practical to use in real word IaaS cloud environments as they consume a lot of hardware resources and require one or two sibling VMs for each monitored kernel version. In addition, these techniques perform passive introspection only and can not be used to for real time intrusion prevention.

Process transplanting

Gu et al. propose a VMI technique called **process implanting** [74] that consists in implanting (i.e. injecting) automatically from the hypervisor level a self-contained inspection program in the monitored guest OS. The self-contained inspection program is written on a trusted OS then compiled statically to generate a binary program that include all the needed OS libraries – delivered – to perform the desired inspection task. To perform introspection, the hypervisor chooses randomly a running process in the VM, saves its context and replaces its code with the one of the self-contained inspection program. Hence, scheduling the victim process will execute the code of the inspection program.

The advantage of process implanting technique is avoiding the semantic gap problem. However, selecting randomly the victim process can make this approach harmful or ineffective if the victim process is an important application (e.g. main business application) or a process that is rarely scheduled. Also, proces implanting technique is vulnerable to DKOM [5] and DKSM [36] attacks because OS libraries statically linked to the inspection program may be unaware of malicious manipulation of kernel code or data.

Srinivasan et al. propose **Process out-grafting** [121], a VMI technique that consists in transplanting a suspicious process from the guest OS out to the host OS to avoid the semantic gap and analyze the behaviour of the suspicious process with security tools available at the host OS (e.g. antivirus, syscall tracer, etc.). To implement this idea, the hypervisor creates a dummy process on the host OS to hold context and image of the suspicious process (i.e. code, data, processor state and memory mapping information). In this way, the out-grafted process runs on the host OS when the dummy process is scheduled. To protect the host OS during the analysis of the out-grafted process, the hypervisor allows only execution of user level code of the out-grafted process. Syscalls and page faults made by the out-grafted process are intercepted by the hypervisor – derived – and forwarded back to the guest OS. Execution end of forwarded syscalls and page

faults in the guest OS are intercepted by the hypervisor which retrieves returned values, forwards them the out-grafted process and finally resumes execution of user level code of the out-grafted process.

The main advantage of this approach is that it enables to analyze a suspicious process in a monitored VM with traditional security tools of the host OS. However, this approach is not suitable for real time monitoring of an entire VM.

Kernel functions binary code reuse

IntroVirt [82] is a VMI system that aims to check if there is an exploitation attempt of a known and non-patched vulnerability in code running inside a VM. IntroVirt does so by executing – at the hypervisor level – a vulnerability-specific predicate (i.e. program) when the vulnerable code is about to be executed. IntroVirt locates the vulnerable code in VM memory using debug information of the vulnerable program and intercepts execution of the vulnerable code by replacing its start instruction with a breakpoint instruction (i.e. int3). To support complex predicates that require fine understanding of VM states, IntroVirt is confronted to the semantic gap problem. Based on the insight that the guest OS contains functions (e.g. system calls) that enable to obtain information of interest, IntroVirt proposes a mechanism to call these functions from the hypervisor in order to bridge the semantic gap and obtain information needed in predicates. To implement this Function Call Injection (FCI) mechanism, IntroVirt saves CPU context (i.e. registers) when it enters kernel mode, manipulates RIP register to point to the entry point of the function to be called, passes function parameters to their corresponding locations (stack or registers), sets a breakpoint on a return address and finally launch VM execution to call the function of interest. When the called function returns, the IntroVirt is notified due to the execution of the breakpoint in the return address. At this point, the IntroVirt restores saved CPU context then resumes the normal execution of the VM.

IntroVirt advanced VMI state of the art with FCI mechanism which facilitates bridging the semantic gap for VMI systems. However, for its proposed application, using debug information is non practical for real world usage as they are generally unavailable.

Hypershell [69] is a practical VMI-based shell that enables a cloud administrator to launch shell commands from the host OS and redirect their effects on the guest OS as if they were executed from inside the VM. Its idea is to use system call interface of the guest OS from the hypervisor level to manipulate low level OS states (e.g. processes) similarly to user level applications in the VM. To do so, Hypershell intercepts and inspects system calls invoked by shell commands launched by the administrator using a library interposition technique [58]. System calls related to process creation, memory management and screen output are resumed and executed on the host OS while the rest of them is injected and executed in the guest OS as software interrupts (int 0x80) that are triggered by a helper process in the VM. The hypervisor and the helper process exchange syscall number, parameters and return values after syscall execution via a secure shared memory. This way, the effect of commands executed on the host OS are reflected in the guest OS.

Hypershell approach is very practical for real world usage, however it focuses on VM management and configuration rather than security. In addition, Hypershell supports only host and guest OSes that share a compatible system call interface.

Based on the insight that data use tells data type, Zeng et al. developed **Argos** [133], a VMI system that aims to automatically deduce types (i.e. semantics) of dynamically allocated kernel data based on kernel

code that uses them. To this end, Argos tracks addresses of dynamic kernel data by intercepting memory allocation functions such as kmalloc whose addresses are provided by Argos users from kernel symbols. Then, based on what system calls lead to read/write the dynamic kernel data, Argos deduces the type of this kernel data. The list of system calls that lead to access each kernel data type of interest is provided in advance by Argos users.

Argos approach is straightforward, however its implementation is based on a complex and heavy dynamic binary code analysis of executing kernel instructions on Qemu emulator. This may limit Argos practicality for real world cloud environments.

3.3 VMI applications

Existing VMI systems explored a variety of security and non security applications based on presented approaches to bridge the semantic gap. In this section, we introduce explored VMI application types and provide a brief technical insights of how they work.

3.3.1 Intrusion detection

Thanks to VMI approaches for bridging the semantic gap, the hypervisor becomes capable of understanding VM activities and observing changes in their states. This enables the hypervisor to detect VM activities and states that violate security policies because of intrusions.

LiveWire [70], VMwatcher [78], XenAcces [102], LibVMI [100] and Virtuoso [63] locate and walk through kernel data structures to compare information about VM states (e.g. running processes, opened files, sockets, etc.) against security policies. Antfarm [80], Lycosid [81] and HyperTap [106] trap writes to CR3 register to observe process executions and detect hidden running processes that are missing from guest OS process list. VMST [66], Exterior [67], Hybrid-Bridge [114], Syringe [49], ShadowContext [131] use respectively data access redirection, system call redirection and function call injection to read and modify VMs states (e.g. processes, files, sockets, etc.) in order to detect security policies violations. RootkitDet [136] compares hash signatures of kernel modules to be loaded against a white-list hash signatures to detect kernel rootkits that inject code into kernel space.

3.3.2 Intrusion prevention

Hardware architectures provide a variety of mechanisms to intercept events of interest. Breakpoints, memory pages protection flags, the undefined instruction, etc. are examples of such mechanism. Combining these hardware-based hooks with hypervisor semantic view of VM states and activities allows VMI system to prevent intrusion attempts and malicious activities as soon as they occur.

Lares [101], SIM [118], Process Implanting [74] and Process out-of-grafting [121] hook function calls (e.g. syscalls) to monitor process behaviours. SecVisor [116] and NICKLE [110] intercept then authenticate loaded and executed kernel code to prevent code injection and illegal execution attacks in kernel space. U-hipe [93] locates memory pages that contain code, data, stack and heap of an application to protect with the help of its process descriptor. Then, by manipulating memory page protection flags according to their

content, U-hipe protect the concerned application against various attacks such as buffer overflow. VMwall [122] intercepts network connection attempts then allows or denies them according to information exposed in the descriptor of the process that created the concerned connection. PHUKO [125] intercepts buffer allocation functions to track buffer addresses and sizes then monitors writes to obtained addresses to detect and prevent buffer overflow attacks. vPatcher [135] inspects received network packets to look for signatures of known exploits to prevent network-based vulnerability exploitations.

3.3.3 Malware analysis

Hypervisor semantic awareness of VM activities and hardware-based hooks enables also VMI systems to track, report and analyze behaviour of malicious processes inside virtual machines.

Ether [61] and Nitro [105] manipulate CPU registers used by system call dispatcher to intercept and trace system calls invoked by an analyzed malware. Ether [61] and Maitland [41] track writes to malware memory pages then intercept their executions to detect and extract dynamically generated malware code. DRAKVUF [87] intercepts kernel functions exported in kernel symbols to track user-level and kernel-level activities made by the analyzed malware.

3.3.4 Live memory forensics

In addition to locating and interpreting kernel data in VM memory, VMI systems enable also the hypervisor to get more semantic information about VM memory content.

Manitou [92], Patagonix [91] and OS-Sommelier [71] compute hash signature of located code pages to identify respectively benign and malicious code pages, identities of running processes and version of the guest OS kernel. [52] tracks writes to memory pages to generate a memory modification map that reports what process did what modification.

3.3.5 Virtual machine subverting

VMI was initially proposed to monitor and protect VM security. However, because virtual machines trust the underlying hypervisor, a malicious hypervisor can use VMI techniques to attack virtual machines.

Fu et al. [68] demonstrated using two different techniques how a malicious hypervisor can attack guest OS authentication system and get unauthorized access to the VMs. The first technique consists in manipulating CPU registers make CMP instruction return true when comparing an incorrect password with the correct one. The second technique consists in intercepting return of an invoked system call that reads correct authentication information (e.g. password, fingerprint, etc.). Then by replacing in VM memory the read authentication information with the entered one (e.g. wrong password or fingerprint), the authentication system is fooled and forced to give access to the malicious user.

3.3.6 Virtual machine management and configuration

As we presented in the previous section, Exterior [67] and Hypershell [69] are examples of VMI-based management and configuration of virtual machines.

3.3.7 User-level application introspection

While most VMI works focus on bridging kernel-level semantic gap, some VMI works explored bridging the semantic gap related to user-level applications to locate in VM memory application data structures of interest.

Tappan Zee [62] proceeds by analysing temporal and spatial memory access patterns to locate encryption key and URLs. SigPath [126] and [44] use data structure fuzzing to locate respectively game score and application settings in memory.

3.3.8 Resource allocation optimization

In virtual environments, the hypervisor manages VM resources with no need for information about their internal states and activities. However, multiple VMI works demonstrated that when the hypervisor has fine understanding of VM states and activities, it can optimize resource allocation for VMs.

KairosVM [47] hooks VM functions related to real time task creation and termination to deduce information about real time scheduling requirements of each process in the VM. Obtained information is then taken into account by the hypervisor when scheduling VM executions to perform the best scheduling decision which respects application real time requirements.

[53] calls a kernel function to identify VM memory pages whose content is irrelevant (e.g. pages in kernel and processes free memory pools). The hypervisor considers then these pages equivalent to zero pages. Accordingly, it de-duplicates [128] these pages and ignores them during VM migration [56] to optimize VM memory allocation.

[31] walks through kernel and process memory descriptors to identify used and free memory pages. Thanks to this information, the hypervisor saves to disk only used pages during VM check-pointing [99] to reduce the check-pointing latency and save disk space when VM memory is underused.

3.4 Discussion and conclusion

In this chapter, we presented a classification of existing VMI approaches for bridging the semantic gap to enable the hypervisor to understand and monitor VM states and activities. We also enumerated in this chapter security and non security applications explored by existing VMI systems.

Table 3.1 recaps properties of presented VMI systems for bridging the semantic gap. Each line of table 3.1 is specific to a VMI system whose category is specified in column 1, its name in column 2 and the key technique employed to bridge the semantic gap in column 3. The rest of columns indicate respectively whether the proposed approach to bridge the semantic gap in each VMI system – column 4 – enables to obtain a rich semantic view, – column 5 – is automatic (i.e. does not require human intervention to bridge the semantic gap), – column 6 – is OS-independent (i.e. designed to work only on specific OSes or uses a provided information that is too OS-specific), – column 7 – is secure, – column 8 – can react to detected attacks and – column 9 – induces an important performance.

In-VM VMI techniques represented by SIM and Lares use an in-VM agent to help the hypervisor to bridge the semantic gap. Because the in-VM agent is generally a software hook deployed manually at

Category	VMI system	Key technique	Semantic view	Automatic	OS independent	Secure	Reactive	Overhead
In-VM	Lares	hardware hook	∼	×	×	∼	✓	✓
In-VM	SIM	software hook	∼	×	×	∼	✓	✓
Delivered	LiveWire	data traversal	∼	×	×	×	×	✓
Delivered	VMwatcher	data traversal	∼	×	×	×	×	✓
Delivered	XenAccess	data traversal	∼	×	×	×	×	✓
Delivered	LibVMI	data traversal	∼	×	×	×	×	✓
Delivered	Virtuoso	binary analysis	✓	∼	∼	×	×	✓
Derived	Antfarm	hardware hook	×	✓	✓	✓	✓	✓
Derived	Lycosid	hardware hook	×	✓	∼	✓	✓	✓
Derived	Ether	hardware hook	∼	✓	✓	✓	✓	×
Derived	Nitro	hardware hook	∼	✓	✓	✓	✓	∼
Hybrid	SecVisor	hardware hook	×	×	×	✓	✓	∼
Hybrid	NICKLE	software hook	×	×	×	✓	✓	×
Hybrid	VMST	binary analysis	✓	✓	✓	∼	×	×
Hybrid	Exterior	binary analysis	✓	✓	✓	∼	∼	×
Hybrid	Hybrid Bridge	binary analysis	✓	✓	✓	∼	∼	×
Hybrid	Process Implanting	PI	∼	∼	×	∼	✓	✓
Hybrid	Process Out Grafting	POG	×	×	∼	✓	×	×
Hybrid	IntroVirt	FCI	✓	∼	✓	∼	∼	✓
Hybrid	Hypershell	FCI	✓	✓	∼	∼	∼	✓
Hybrid	Argos	binary analysis	∼	✓	✓	∼	×	×

✓: Good. ∼: Moderate. ×: Bad.

Table 3.1 – Comparison between VMI systems properties.

specific addresses to monitor activities of interest, it is difficult for in-VM VMI technique to provide a rich semantic view for the hypervisor due to the important needed manual effort. So, in addition to the limited

semantic view, in-VM VMI techniques are OS-dependent and not automatic. Also, the use of the in-VM agent presents a risk of tampering with the semantic view returned to the hypervisor as the in-VM agent runs inside an untrusted kernel. On the other hand, having the in-VM agent presents advantages such as the ability of stopping unauthorized activities with a minimum overhead as the VM execution is not suspended while hooking monitored activities.

Out-of-VM delivered VMI techniques use information delivered by a security expert to bridge the semantic gap. Most of these techniques rely on data traversal to bridge the semantic gap and obtain information about VM states and activities. Similarly to in-VM VMI techniques, the needed manual effort is tedious and makes it difficult for these techniques to obtain a rich semantic view. Because this manual effort should be repeated for each supported OS, out-of-VM delivered VMI techniques are generally too OS-specific and not automatic. In presented systems of this category, Virtuoso is the only system that enhances automation of bridging the semantic gap and creating VMI tools thanks to its approach based on binary analysis. Out-of-VM delivered VMI techniques are designed based on the assumption that most OS details are invariants. This makes them vulnerable to DKSM and DKOM attacks and their semantic view can be wrong after a major OS update. Generally, out-of-VM delivered VMI techniques simply reads monitored data structures and do not modify them as it may be difficult to leave the OS in a consistent state after kernel data modifications which limits their reactivity against attacks. Finally, out-of-VM delivered VMI techniques do not induce an important performance overhead as they do not actively intercept VM activities.

Concerning out-of-VM derived VMI techniques, they rely on the hardware architecture and virtualization extensions to drive semantic information about VM states and activities. This leads to a full automation as there is no need for a human intervention and also to a strong OS independence as the hardware architecture is generally used in the same way in modern operating systems. Semantic information reflected in the hardware architecture is trusted because the OS can not operate correctly if its states in the hardware architecture are incorrect. Out-of-VM derived VMI techniques can be reactive as the hypervisor can modify the hardware states to stop detected malicious activities. The overhead induced by out-of-VM derived VMI techniques is variable and dependent on the VMI system or its application. For instance, the performance overhead induced by Ether is important due the interception of all executing instruction. On the other hand, the performance overhead of Nitro is moderate as it intercepts only invoked system call. The semantic view that can be obtained by out-of-VM derived VMI techniques is generally very limited because the hardware architecture reflects only limited information on few VM states and activities.

Hybrid VMI systems use a combination of techniques from other categories. This is why contrary to systems of other categories, properties of hybrid VMI systems are very variable and dependent on techniques employed by each VMI system. SecVisor and NICKLE target a very specific application (prevent illegal code execution). So they offer a very limited semantic view. Also, SecVisor requires kernel source code modification while NICKLE uses manually provided and too OS-specific information. This limits their automaticity and OS independence. Both systems are secure and reactive as they rely on trusted code signatures and use hooks to intercept malicious activities. The performance overhead induced by SecVisor is moderate compared to NICKLE whose approach is based on a software emulator.

VMST, Exterior and Hybrid-Bridge use dynamic binary analysis to implement a VMI approach based on data access redirection. These systems offer a rich semantic view thanks to the automatic instrumentation

of system calls. Also, while these systems were evaluated on Linux kernels, their design is largely OS independent. These systems trust information returned by instrumented system calls, so their semantic view can be tampered with using DKOM attacks. Concerning their reactivity to malicious activities, VMST is not designed to modify kernel data to stop attacks. While this is not the case for Exterior and Hybrid-Bridge, these systems do not offer real time protection against malicious attack. So we judge their reactivity as moderate as they can only recover from some attacks and not prevent them. The three systems use a heavy dynamic binary code analysis on top of a software emulator, so they induce an important performance overhead.

Regarding the process implanting approach, while it enhances the automaticity of injecting an in-VMI agent its properties are very close to in-VM approaches. Process out grafting approach is designed for malware analysis in a controlled environment, so its goal is not to obtain automatically a rich semantic view about VM states and activities and to protect against attacks.

IntroVirt and Hypershell employ Function Call Injection (FCI) mechanism to obtain automatically rich semantic information about VM states and activities with the help VM system calls. This mechanism is OS independent and enables to react against attacks with a minimum overhead. IntroVirt is designed to work with some manually provided information contrary to Hypershell which is more automatic. On the other hand, Hypershell is less OS independent than IntroVirt as it requires a guest and host OSes with a compatible system calls interface.

Finally, Argos uses a dynamic binary code analysis to deduce data semantics based on their use context. While this approach is not intended to fully bridge the semantic gap and to react to attacks, it helps to infer data semantics in an automatic and OS independent way. However, the design of Argos based on a dynamic binary code analysis on top of a software emulator induces an important execution time and performance overhead.

In the various system descriptions presented in this chapter, we observe that VMI systems that employ binary analysis techniques and make use of guest kernel binary code in a way or another to bridge the semantic gap (e.g. intercept, call and analyze system calls and internal kernel functions) surpass other VMI systems. These VMI systems that leverage – only a part of – guest kernel binary code automate bridging the semantic gap and ease design of powerful VMI applications that require fine-grained understanding of VM states and activities. However, none of them fully bridges the semantic gap and enhances all the discussed properties at once. Therefore, in this thesis we explore new binary analysis techniques that enable VMI systems to reuse automatically all the guest kernel binary code to enable the hypervisor to obtain fine-grained understanding of VM states and activities in the most practical way for usage in real world IaaS cloud environments.

II

Contributions

4

Main kernel binary code disassembly

Chapter abstract

Despite that explored binary analysis and kernel binary code reuse VMI techniques leverage a part of VM kernel binary code, each one of them enables to considerably enhance a VMI system property depending on the employed technique. To considerably enhance all properties of a VMI system at once, we explore the idea of applying binary analysis and kernel binary code reuse techniques on all the kernel binary code. To this end, we introduce in this chapter a background about binary analysis and its basic notions. We also present in this chapter a new disassembly approach that enables the hypervisor to fully locate and disassemble VM kernel binary code in an automatic and OS independent way. Our disassembly mechanism employs a novel backward disassembly technique that enables a binary analysis system to interpret binary code in a reverse way despite challenges presented by the ambiguity of the x86 disassembly.

4.1 Introduction

In the last chapter, we saw that automatic and efficient VMI approaches and applications are based on reusing some of the VM kernel binary code (e.g. syscall interception and call injection). To better bridge the semantic gap, namely to obtain a more fine-grained understanding of VM states and activities and to expand the possible range of – security and non security – applications in a practical way for IaaS cloud environments, our insight is to reuse automatically all VM kernel binary code from the hypervisor layer with the help of binary analysis techniques. To this end, the hypervisor should first be able to disassemble VM kernel binary code before analyze it.

Despite that VM kernel binary code is in the VM memory which is available and accessible to the hypervisor, multiple challenges must be addressed in order to disassemble and analyze VM kernel binary code. In particular, kernel space in the VM memory contains kernel modules and data in addition to the

main kernel code. Only this latter one is of interest for binary code reuse VMI as it contains system calls in addition to the other kernel functions. Because all these kernel parts are seen by the hypervisor as zeros and ones due to the semantic gap, it is not obvious how to differentiate between them automatically. Moreover, the location (i.e. start address) of VM main kernel code is an OS-specific information which is different between OS families (e.g. Linux and Windows) and may change across versions of the same OS family. For instance, main kernel code starts at the – virtual – address 0xc0100000 in Ubuntu 4 and at 0xc1000000 in latter versions. In addition, today kernels employ Address Space Layout Randomization (ASLR) mechanism [15] which causes main kernel code location to be shifted by a random offset after each VM reboot.

These challenges complicate the process of disassembling VM main kernel binary code which requires to be started at a correct instruction boundary (i.e. a correct instruction start address). Disassembling a binary code from an incorrect instruction boundary such as an address that points to the middle of an instruction, returns an incorrect instruction sequence and tampers with correct binary code reuse for VMI. Therefore, the main challenge for reusing all kernel binary code for VMI is how to automatically locate and disassemble correctly VM main kernel binary code in an OS-independent way.

In this chapter, we focus on the problem of disassembling correctly VM main kernel binary code from the hypervisor layer. We address the precise identification of VM main kernel binary code boundaries (i.e. start and end addresses) in the next chapter. We propose in this chapter a new binary analysis mechanism that given a virtual address of an arbitrary instruction in the main kernel code, it enables the hypervisor to automatically locate and disassemble correctly the rest of VM main kernel binary code despite the presented challenges. In this mechanism, we introduce a novel backward disassembly technique that enables a binary analysis system to locate correctly instruction boundaries that precede a virtual address of an arbitrary instruction in VM memory. Before we detail our disassembling mechanisms, we introduce in the next section basic notions in binary analysis.

4.2 Background and problem statement

4.2.1 Binary analysis

Software source code contains rich and high level information about how the software works. These high level information are translated to low level information (zeros and ones) and their semantics are lost when the software source code is compiled into a binary program. Binary analysis is a wide field in computer science that aims to reconstruct high level information about a given software from its binary code. It has a variety of applications such as reverse engineering [55], cyber-forensics [98], program debugging [8], performance profiling [42], etc. In this section, we will present binary analysis concepts that are relevant to techniques presented in this thesis.

Binary code disassembly is the process of translating zeros and ones in the analyzed binary code into understandable sequences of assembly instructions. This process is called decompilation if the binary code is translated into high level language such as the C programming language [88].

4.2.2 Static VS dynamic binary analysis

Disassembling a binary code can be done statically or dynamically. These two modes are referred to as static and dynamic binary code analysis. Static binary code analysis is the process of disassembling and interpreting a binary code without requiring its execution such as binary code in an executable file or loaded code in memory that belongs to a running or sleeping process. In contrast, dynamic binary code analysis consists in disassembling and interpreting a running binary code as it executes [88]. Each one of these approaches has advantages and limitations.

First, static binary code analysis can enable to disassemble and interpret all assembly instructions in the analyzed binary code (i.e. full code coverage) while dynamic binary code analysis is limited only to the executed instructions. Second, static binary code analysis is very fast and its execution time depends mainly on the size of analyzed code. On the other hand, execution time of dynamic binary code analysis is dependent on the one of the analyzed binary code. Third, static binary code analysis is not resistant to binary code obfuscation and can be tampered with and forced to return incorrect instruction sequences. Dynamic binary code analysis is not vulnerable to this attack because instructions are necessarily disobfuscated when they are executed. Finally, static binary code analysis can not extract and analyze dynamic information such as register and memory contents whose values are determined in execution time. Again, this is not a problem for dynamic binary code analysis as instructions (and their operands) are analyzed during their execution.

We choose static binary code analysis to design and implement techniques presented in this thesis for the following reasons. First, our goal is to reuse all kernel binary code to better bridge the semantic gap for virtual machine introspection. Static binary code analysis is likely to achieve this as it can enable to disassemble and interpret all binary code contrary to the dynamic analysis. Second, static binary code analysis is faster than the dynamic analysis, so it can help to reduce the introspection overhead as the VM execution is not required to be suspended. Third, today kernels do not employ code obfuscation, so an efficient static binary code analysis is possible. Finally, most information of interest for introspection such as syscall, function and global kernel data addresses (e.g. process list) are static information. So they could be obtained with no need for kernel code execution using a static binary code analysis.

Static binary code analysis can be performed using two famous algorithms, linear sweeping and recursive traversal [88].

The linear sweeping algorithm (illustrated the left part of the figure 4.1) reads binary code bytes and translates them into instructions in a purely sequential way. The main advantage of this algorithm is its simplicity while its main limitation is that this algorithm does not differentiate between code and data bytes. So when reaching data bytes (e.g. end of code section or data embedded in code section), the linear sweeping algorithm continues to – mistakenly – translate data bytes into instructions until it encounter data bytes that do not correspond to any valid instruction. At this point, there is no obvious way for the linear sweeping algorithm to determine precisely from which byte it started to translate non code bytes into instructions. GNU objdump [22] is a famous static binary analysis tool that uses the linear sweeping algorithm.

To address the linear sweeping weakness of confusing data and code bytes, the recursive traversal algorithm (illustrated the right part of the figure 4.1) takes into account information about code execution flow

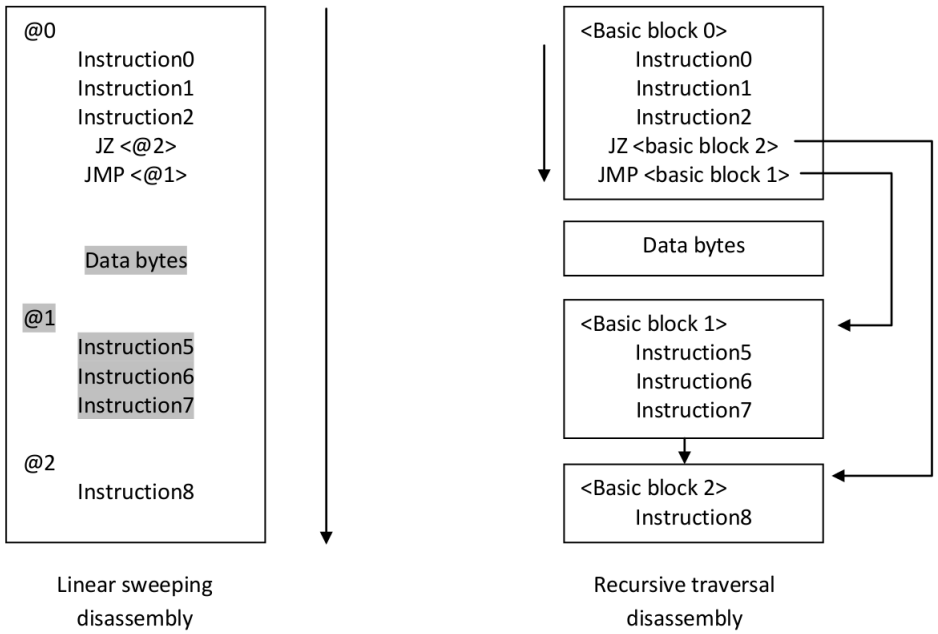


Figure 4.1 – Illustration of disassembly algorithms. The arrows represent the disassembly flow and gray areas represent possible disassembly errors.

embedded in assembly instructions (called control flow information). More specifically, the recursive traversal algorithm disassembles sequentially binary code bytes until reaching a branch instruction (e.g. call, jmp, conditional jumps, etc.). At this point, the recursive traversal algorithm inspects the branch instruction to determine possible addresses of instructions to be executed just after the branch instruction. The disassembling process is then repeated recursively from each address of instructions reachable from the branch instruction until all reachable instructions in the binary code are disassembled.

The recursive traversal algorithm divides hence the disassembled binary code into code fragments linked with information about their execution flow. These code fragments are referred to in the literature as basic blocks, which are defined as an instruction sequence whose first instruction is the only destination of a branch instruction (i.e. only entry point) and their last instruction is the only instruction that points to the entry point of another basic block (i.e. only exit point, such as branch instructions). The relationship between the basic blocks forms a graph called Control Flow Graph (CFG). The recursive traversal algorithm enables hence the disassembler to translate only code bytes and to get around data bytes as the execution flow does not reach them.

However, contrary to the linear sweeping algorithm, the recursive traversal algorithm can not disassemble valid but unreachable code. Also, indirect branch instructions whose destination addresses may change dynamically at run time as they are computed based on values of a register operand, confuse the recursive traversal algorithm and cause it to miss valid code segments. IDA Pro Disassembly [14] and Dyninst [43] are famous disassemblers that are based on the recursive traversal algorithm. Some exiting works such as [115] combines the two disassembly algorithms to enhance their binary analysis.

We choose to design and implement mechanisms presented in this thesis based on the linear sweeping algorithm for the following reasons. First, linear sweeping algorithm is very simple to implement. Second, a recent work [34] on 981 realistic binaries disassembly shows that the linear sweeping algorithm achieves

100% disassembly accuracy on binaries compiled with GCC [7] and Clang [3], and 99.92% in the worst case for binaries compiled with Microsoft Visual Studio [21]. The perfect performance of linear sweeping algorithm on GCC and Clang is due to the fact that these two compilers do not insert data bytes inside code sections, so the linear sweeping algorithm translates correctly all code bytes into valid instructions. As we evaluate VMI techniques we propose in this thesis on Linux kernel whose code source can be compiled – for the moment – only with GCC compiler, the use of the linear sweeping algorithm is relevant. Finally, while the linear sweeping algorithm does not require control flow information, using it does not prevent later from extracting control flow information and build a control flow graph if needed.

4.2.3 Problem definition

Almost all works in the literature of static binary code analysis focus on analysing executable files and their running images. In this case, location and size of the file portion that contains valid binary code (called code section) are known and indicated in headers of the executable file whose format is well documented. In our case, location and size of the main kernel binary code in a VM memory are completely unknown to the hypervisor due to the semantic gap. In addition, these information are too OS specific and highly dynamic as they may change due to randomization mechanisms or OS updates. Without these information, binary analysis can not be performed because the analysis system does not know from where it should start and when to stop the analysis. This present a significant challenge for reusing main kernel binary code from the hypervisor layer.

We present in the next section how we proceed to locate and disassemble correctly main kernel binary code from the hypervisor layer.

4.3 Main kernel code disassembly

Discovering correctly instructions of the main kernel code in VM memory is a vital step for reusing all main kernel binary code for VMI. Doing so is very challenging due to the ambiguity of the x86 disassembly and the absence of information about the location and the size of the main kernel binary code. To address this challenge, we detail in this section our main kernel binary code mechanism.

For the need of binary analysis systems presented in the next chapters, we design our disassembly mechanism to discover main kernel instructions by fragments that we call *code block*. In this thesis, we define a *code block* as a set of consecutive instructions whose last instruction is the only exit point (branch or UD2 undefined instruction) and its first instruction (can be any instruction) is an entry point that comes after an exit point of a precedent code block excluding padding instructions (e.g NOPs, etc.) and data bytes. Compared to a basic block (defined in §4.2.2), a code block can have more than one entry point. The advantage of this notion in the context of this chapter is its simplicity because its discovery does not require identifying and inspecting entry and exiting points similarly to a basic block. So, the simplicity of the code block notion goes with the simplicity of the linear sweeping algorithm on which we design our disassembler.

We present next an overview of our main kernel binary code disassembly mechanism then its detailed description.

4.3.1 Overview

Our mechanism to locate and disassemble main kernel binary code is composed of three steps. A first step in which we locate correctly one instruction that surely belongs to the main kernel code. We call this instruction I0. The two other steps consist in disassembling forward and backward from the address of that instruction (i.e. address of I0) until reaching main kernel code boundaries to discover code blocks that respectively succeed and precede I0. Our disassembler detects that it went beyond main kernel code boundaries when it encounters invalid instruction bytes (i.e. a sign of data bytes) or a page that contains only null bytes (i.e. a sign of an empty page outside main kernel code). Very probably, using these rules will slightly over approximate the size of the main kernel code but helps to stop the disassembly process. As we mentioned earlier, we will address precise identification of main kernel code boundaries in the next chapter. We present hereafter details of each step in our main kernel code disassembler.

4.3.2 Locate I0 instruction

Locating an instruction that certainly belongs to the main kernel code is straightforward thanks to the hardware architecture and virtualization extensions [57]. For example, the address in SYSENTER_EIP_MSR register which points to the start instruction of the system call handler can be chosen as an address of I0. Some other possible addresses of I0 can be obtained from instruction pointer register (RIP/EIP) when intercepting – with the help of virtualization extensions – execution of instructions specific to main kernel operations such as ‘mov cr3, eax’ instruction used for process switching.

4.3.3 Locate succeeding blocks

Locating code blocks (B1, B2, ..., Blast) that succeed I0 instruction is also a straightforward step. This is because disassembling forward from a correct instruction boundary (I0) until reaching the end of main kernel code (e.g. encountering an invalid instruction) is very simple using the linear sweeping algorithm.

4.3.4 Locate preceding blocks

Contrary to the previous step, locating code blocks (B-1, B-2, ..., Bfirst) that precede I0 instruction is very challenging. Backward disassembly is thought to be impossible [88] as there is no obvious way to locate correctly boundaries of instructions that precede a given address. This is because instructions in the x86 architecture have a variable length and they are not aligned in memory. In addition, as almost any byte in the memory can be interpreted as a possible start of a valid instruction, it is extremely difficult even for a human being to tell correctness of an instruction sequence obtained by disassembling from a random precedent address until I0. We present hereafter our approach to address this challenge.

To solve the problem of locating correctly boundaries of instructions that precede I0, a key observation is that disassembling from an incorrect instruction boundary (i.e. unintended instruction) produces usually an instruction sequence in which correct instructions (i.e. intended instructions) reappear just after few incorrect ones. This phenomenon is known as disassembly self-repairing [90].

<sys_getpid>	Kernel 3.13.1:		
0xc106be00	push ebp		
0xc106be01	mov ebp, esp	0xc106be02	in eax, 0x3e
0xc106be03	lea esi, [ds:esi]	0xc106be04	lea esi, [esi]
0xc106be08	mov eax, [fs:0xc1a4dfd4]	0xc106be08	mov eax, [fs:0xc1a4dfd4]
0xc106be0e	mov eax, [eax+0x248]	0xc106be0e	mov eax, [eax+0x248]
0xc106be14	mov eax, [eax+0x264]	0xc106be14	mov eax, [eax+0x264]
0xc106be1a	call 0x7cc0	0xc106be1a	call 0x7cbe
0xc106be1f	pop ebp	0xc106be1f	pop ebp
0xc106be20	ret	0xc106be20	ret

Figure 4.2 – An example of disassembly self-repairing

Figure 4.2 illustrates an example of disassembly self-repairing phenomenon where the left instruction sequence is obtained by disassembling from the correct beginning of the sys_getpid function in the Linux kernel until its end (from the address 0xc106be00 to the address 0xc106be20). The sequence in the right is obtained by disassembling from an incorrect instruction boundary at the address 0xc106be02 (i.e. unintended instruction). In this latter sequence, correct instructions reappear just after two incorrect ones and starting from 0xc106be08, both instruction sequences are synchronized and correct. Disassembly self-repairing phenomenon is possible only thanks to the properties of x86 instruction set, namely variable instruction length and the non alignment of instructions in memory. Therefore, this phenomenon is independent from any implementation of disassembling libraries.

To locate code blocks that precede I0 instruction, we begin by locating correctly their end instructions with the help of disassembly self-repairing phenomenon. We deduce then block start instructions from located block ends according to our definition of code blocks (§4.3). To illustrate the idea of our approach, let's consider two instruction sequences (sequence1 and sequence2), obtained respectively by disassembling until a same end address starting from two random addresses address1 and address2, with address2 sufficiently lower in memory than address1. If instruction at address1 appears at the same address in sequence2, then starting from this instruction both sequences are synchronized and assumed to be correct thanks to disassembly self-repairing phenomenon. In this case, if address1 is chosen such that it contains an opcode of a block end instruction, then we have located correctly one block end instruction. In the light of this illustration, we detail hereafter how we proceed to locate correctly block ends that precede I0 instruction.

Our backward disassembly approach illustrated in figure 4.3 is composed of three steps: the first step consists in scanning virtual addresses that precede I0 byte by byte (i.e. from I0 virtual address towards lower virtual addresses) and locating multiple different offsets of block end instruction opcodes (e.g. conditional jump, unconditional jumps and return). In the current implementation of our prototype, we chose to locate three different block end instruction opcodes to accelerate locating multiple code blocks at once. At this point, we do not known if these located block end instructions are correct (i.e. intended) or incorrect (i.e. unintended) instructions due to the ambiguity of the x86 disassembly. We validate – if possible – the correctness of one of these located located block end instructions with the help of the two remaining steps.

In the second step, we order the located offsets such that offset3 (is at a lower virtual address than) < offset2 < offset1. Consequently, we obtain three instruction sequences sorted according to their length in term of instruction number. More specifically, the instruction sequence between offset1 and I0 (the smallest sequence) has usually less instructions than the instruction sequence between offset2 and I0 (the medium sequence). Similarly, the medium sequence also has usually less instructions than the one between

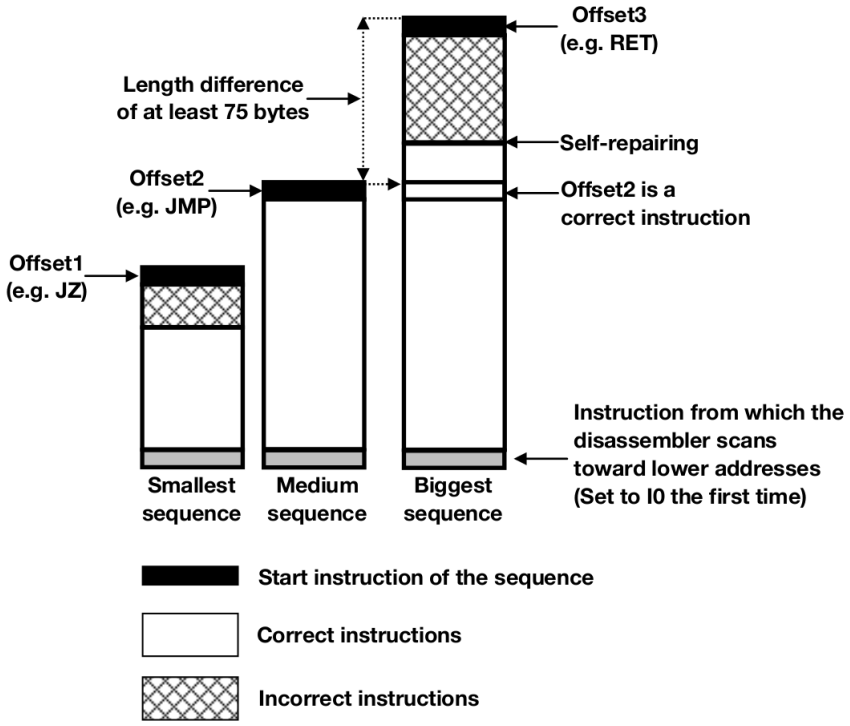


Figure 4.3 – Locating precedent block end instructions using disassembly self-repairing phenomenon

offset3 and I0 (the biggest sequence). Figure 4.3 presents an example of these three sequences where their incorrect instructions are illustrated by the cross hashed areas while the correct instructions are illustrated by the white areas. Figure 4.3 illustrates also that in some sequences the correct instructions reappear thanks to disassembly self-repairing phenomenon. Because correct instructions of the smallest sequence (if there is any) are usually present also among instructions of the other sequences, we ignore this sequence in the next step to accelerate main kernel code disassembly.

After ignoring the smallest sequence, we come to a case similar to the illustration we presented before detailing our backward disassembly approach. So, in the last step we consider that the start of the medium sequence (i.e. located block end instruction) is correct if the two following conditions are satisfied. Condition 1: the start instruction of the biggest sequence has at least N bytes more than the medium sequence (i.e. biggest sequence start instruction is sufficiently low in memory) to ensure that the disassembly self-repairing phenomenon took place in the biggest sequence. According to an existing work on binary analysis, disassembly self-repairing phenomenon takes effect generally within two or three incorrect instructions [90]. In the implementation of our backward disassembly, we set N to 75 bytes which is a value that corresponds to at least five instructions of maximum length (15 bytes) and at more 75 instructions of minimum length (1 byte) in the x86 architecture. We found that in practice this value is largely enough to ensure that the disassembly self-repairing took effect in the biggest instruction sequence. Condition 2: the start instruction of the medium sequence belongs to the biggest sequence (i.e. appears at the same virtual address like in the case illustrated in figure 4.3).

If one of these conditions is not satisfied, meaning that the length difference between the biggest sequence and the medium sequence is less than 75 bytes or/and the start instruction of the medium sequence does not belong to the biggest sequence, we assume that this latter instruction is incorrect.

The backward disassembly steps are then repeated from the start instruction of the biggest sequence (i.e. offset3) until condition 1 and 2 are satisfied. In this case, we consider that all instructions of the medium sequence are correct. Hence, they can be linearly disassembled in forward way from the start of the medium sequence until I0 and divided into code blocks which can be analyzed by a binary analysis system. To discover more code blocks that precede I0 instruction, the whole process of the backward disassembly can be repeated from each correct start instruction of the medium sequence until exceeding the start of the main kernel code (e.g. encountering a page that contains only null bytes).

4.4 Conclusion

In this chapter we detailed encountered challenges by a VMI system that aims to analyze all main kernel binary code to reuse it for VMI. To solve these challenges, we first introduced in the beginning of this chapter basic notions in binary analysis. After that, we detailed the design a new binary analysis mechanism that enables the hypervisor to locate and disassemble correctly main kernel binary code located in the memory of a running VM. In the design of this mechanism, we presented a novel backward disassembly mechanism that enables a binary analysis system to locate correctly boundaries of instructions that precede a specified virtual address, a procedure that was thought to be impossible.

5

Kernel binary code identification

Chapter abstract

Kernel symbols file is a valuable source of information about kernel data and function addresses. This is why most existing VMI systems rely on it to bridge the semantic gap. Kernel symbols file is unique for each kernel and symbol addresses change according to kernel version and configuration options during kernel compilation. Hence, in IaaS cloud environments, VMI systems must first identify the kernel running inside the VM to determine appropriate kernel symbols file to use for VMI.

In this chapter, we address the problem of kernel fingerprinting in IaaS cloud environment. We start by showing limitations of existing kernel fingerprinting systems. Then we detail the design of our system K-binID, which uses our kernel disassembler to divide VM main kernel binary code into code blocks that facilitate precise identification of VM main kernel binary code despite challenges presented by compiler optimizations and ASLR.

5.1 Introduction

In the last chapter we presented a binary analysis mechanism that enables the hypervisor to locate main kernel code in the memory of a running VM and correctly disassemble its binary code in an automatic and OS-independent way. In order to instrument (i.e. intercept, call and analyze) system calls and kernel functions in the located main kernel code, the hypervisor should first obtain their addresses in the main kernel code. To do so, one possible way is to fingerprint (i.e. identify) version and customization (i.e. configuration options during compilation) of the main kernel binary code then use system call and function addresses exported in available symbols file associated to the identified kernel.

In this chapter, we present K-binID, a Kernel binary code IDentification system that enables the hypervisor to precisely fingerprint the main kernel binary code in the memory of a running VM despite – the to

be detailed – challenges presented by kernel customization options such as ASLR. K-binID enables then a VMI system to identify which symbols files should be used to extract addresses of interest for VMI. K-binID uses mechanisms presented in the last chapter to locate and disassemble the main kernel binary code then divide it into code blocks that as a whole form a fingerprint of VM main kernel binary code.

This chapter is organized as following: we introduce in the second section the topic of kernel fingerprinting and detail challenges that can be encountered in real world IaaS cloud environments. We present then existing approaches for kernel fingerprinting and highlight their limitations. We detail then K-binID approach for kernel fingerprinting, its main limitations and finally we conclude in the last section.

5.2 Background and problem statement

Most existing VMI approaches for bridging the semantic gap make use of available kernel symbols (found in system.map file in the Linux kernel) to obtain data and function addresses of interest for introspecting VM states and activities (e.g. init_task for process list head and kmalloc/kfree for memory allocation operations). In these approaches, addresses of interest were either manually extracted then provided to the VMI system (e.g. by a security expert) or extracted by the VMI system from manually specified kernel symbols file. Hence, human intervention is generally required to enable the VMI system to bridge the semantic gap.

In real world IaaS cloud environments, human intervention to provide such addresses that are too kernel-specific for a VMI system leads to a huge manual effort due to the possible big number of different kernel images present inside the hosted VMs. Moreover, doing so presumes that the security administrator is aware of version and customization of each kernel present in introspected VMs. In practice, this assumption is hard to satisfy as the cloud operator can not constrain its client to install specific kernel versions and customizations. In addition, despite that the cloud operator can propose a list of predefined kernel versions and customizations, the known information about the VMs can become outdated as the cloud clients can install, change and upgrade to kernels of their choice. Hence, in real world IaaS cloud environments, kernel properties should be considered unknown and addresses of interest for VMI should not be hard coded or provided manually.

Kernel fingerprinting is a possible way to identify the appropriate kernel symbols file from which functions and data addresses of interest should be extracted. Kernel fingerprinting for VMI implies the identification of main kernel binary code version and also its customization (i.e. configuration options during compilation). This is because addresses of kernel functions and data are very sensitive to changes in the kernel version and also to changes in its customization during kernel compilation. Generally, kernels that share the same version but differ in their customization options (e.g. compiler optimization level) have function and data addresses that are completely different.

Existing kernel fingerprinting approaches are limited in precision and usability and they are not reliable and suitable for VMI approaches destined for real world IaaS cloud environment. The limitation of their precision is cause by their insufficient code coverage, meaning that these works deduce the kernel version and customization based on a limited part of its binary code. Insufficient code coverage may induce to confound kernel versions and customizations that have minor differences at the binary level [54][72].

We consider kernel fingerprinting techniques that require access to VM disk [79] as limited in usability. Accessing VM disk in a IaaS cloud is too intrusive and may not be feasible or helpful in many cases. First, systems such as TrueCrypt [6] enables a cloud client to totally encrypt VM disk except its bootloader. Despite that the hypervisor is still able to launch such VMs, it can not access their disk for kernel fingerprinting. A second case is when the VM disk contains multiple OSes or kernels. The VM disk in this case can not tell which kernel is the one loaded and executed from VM memory. Finally, to access VM disk the hypervisor must be able to understand the file system of VM disk. Doing so may not be feasible in case of an undocumented file system. Hence, reliable kernel fingerprinting from the hypervisor level should be based on VM memory which is accessible to hypervisor and should not involve access to VM disk.

In the context of real world IaaS cloud environments, we also consider as limited in usability kernel fingerprinting techniques that require the kernel to be compiled with function prologues to easily locate parts of kernel binary code in the VM memory [71]. Actually, kernel binary code of multiple versions of various famous Linux distributions are optimized and configured to remove kernel function prologues and epilogues. We observed this optimization in multiple versions of Debian, CentOS, OpenSuse distributions, etc. For instance, we found that 99% of kernel function in Debian 7 do not start with specific prologue instructions.

Hence, in real world IaaS cloud environments, kernel fingerprinting systems should not presume specific kernel compilation options especially that any Linux kernel can be recompiled and configured to present a highly optimized binary code.

5.2.1 Problem definition

In the literature of kernel binary code fingerprinting, the main challenge was generally the difficulty of locating automatically the main kernel binary code with practical assumptions that are convenient to real world cases. hopefully, the disassembler of main kernel binary code presented in the last chapter addresses this challenge and enables the hypervisor to locate and disassemble main kernel binary code in a VM memory in an automate and OS-independent way. Moreover, the design of our disassembler mechanism makes it resistant to challenges presented by ASLR and compiler optimizations. After we present in more details major existing approaches for kernel fingerprinting, we present how we build K-binID on top of our disassembler mechanism.

5.3 Related works

Multiple works in the literature address the problem of operating system fingerprinting. These works can be classified into four categories depending on the source of information from which they derive signatures that characterize a specific kernel.

Network-based systems such as Nmap [94] inspect network packets received from kernel services in response to packets send to the target machine to identify response patterns that are specific to each OS version.

On the other hand, CPU-based systems such as UFO [108] inspect values of CPU registers (e.g. segment

selectors) to identify value patterns that are specific to each OS version. Systems of these two presented categories have no coverage on kernel binary code. Hence, they can not precisely differentiate between kernel versions and customizations that present exactly the same patterns.

Disk-based fingerprinting systems such as Virt-inspector [79] access to VM disk to read kernel files and deduce the kernel version and customization. As we detailed earlier, this approach is too intrusive and may not be feasible in many cases.

Regarding memory-based fingerprinting system, the basic idea is to access VM memory, locate fragments of the main kernel binary code then deduce the kernel version and customization based on signatures (e.g. hashes) of the located code fragments.

Christodorescu et al. [54] deduces the kernel version and customization based on hashes of interrupt handlers binary code. Limiting the fingerprinting process only on interrupt handlers leads surely to confound kernel versions and customization with minor differences at the binary level.

OS-Sommelier [71] walks through page table entries of a running process and identifies memory pages that belong to kernel space. OS-Sommelier scans then these pages and locates fragments of main kernel binary code that are marked by function prologue instructions. For each located fragment, OS-Sommelier computes its hash signature and compares it to a reference database to fingerprint the main kernel code. The usability of OS-Sommelier is limited in real world IaaS cloud environment as it works only kernels that are compiled with function prologues.

OS-Sommelier+ [72] is a latter version of OS-Sommelier that aims to fingerprint the kernel version independently of compiler optimization level during the kernel compilation. OS-Sommelier+ uses data access invariants (in particular, the offsets of circular linked lists in kernel data structures) as signatures to identify the kernel version. Being insensitive to compiler optimizations, OS-Sommelier+ does not differentiate hence between different customizations of a same kernel version. So in our use case, OS-Sommelier+ can not identify the appropriate kernel symbols file that should be used for VMI.

A recent fingerprinting system called CodeID [32] investigated the use of relocation information present in executable files as signatures to fingerprint application or kernel code images loaded in memory. The idea of CodeID is to scan all memory pages, consider each four bytes in each page as a relocation information then compare them to a reference database of relocation information extracted from executable files. The identity of the executable file whose relocation information match the most with the possible ones in the memory is considered as the identity of the code image in the memory.

This brute force approach of CodeID does not differentiate between data and code pages. Hence it parses pages that are not related to the code to identify. This leads to an important computation time and exposes the fingerprinting process to attacks that insert or duplicate binary code and relocation information from other versions.

K-binID outperforms existing fingerprinting approaches thanks to code block signatures and to its complete and noiseless coverage of main kernel binary code independently of compiler optimizations and address space randomization.

5.4 K-binID approach

5.4.1 Assumptions and threat model

The goal of K-binID is to identify kernel version and customization with the help of a multi-OS database of kernel code blocks signatures.

Because the kernel code is not static, we assume that our multi-OS signatures database is complete and and enables to identify any form of a known code block after changes in its code. Dynamic changes in the kernel code are due to ASLR (i.e. different addresses after each reboot) and to instruction patching to (1) activate kernel functionalities such as function tracing, (2) transform some kernel code parts into critical sections in case the (virtual) machine has more than one processor and (3) replace some instructions with alternative ones that are more optimized for the current processor [73][83]. These dynamic kernel code patches change code block signatures and cause K-binID to fail to identify the code block concerned with the dynamic patch (i.e. ignore to which kernel version and customization it belongs).

There are two possible ways to address dynamic kernel code changes. The first one is to generate normalized signatures that are insensitive to these dynamic changes. Normalized signatures generation consists in identifying then ignoring the dynamic parts of a code block during signature computation (e.g. replace them by zeros) to produce the same signature for the different forms of a same code block. Because the dynamic changes that can be applied to kernel instructions are limited and known (limited numbers of possible randomization offsets and alternative instructions which are detailed and listed in kernel source code), the second approach consists in generating signatures of all possible forms of a code blocks to create a complete signatures database. We use the first approach to handle ASLR and we perform the fingerprinting on machines similar to those on which we generate the signatures database to ignore handling other dynamic changes.

In order to safely use exported addresses in kernel symbols file for VMI, we assume that kernel code integrity is verified by a VMI system (such as SecVisor [116]) that uses K-binID.

5.4.2 Overview

Figure 5.1 illustrates K-binID approach for fingerprinting VM kernel binary code. This approach is composed of two phases: an offline phase and an online phase.

In the offline phase (detailed in section §5.4.3), K-binID uses available kernel symbols files to generate code blocks signatures of known kernels and build a multi-OS signatures database.

In the online phase (detailed in section §5.4.4) which is launched after the introspected VM is booted, K-binID (1) locates main kernel code blocks in the VM memory with the help of our disassembler presented in the last chapter, (2) identifies for each located code block the list of possible kernels in which it belongs with the help of signatures generated in the offline phase and finally (3) pinpoints the kernel version and customization that includes all located code blocks.

Once the VM kernel is identified, K-binID loads addresses exported in available kernel symbols file that corresponds to the identified kernel. In case K-binID detects the presence of kernel base address randomization, it adapts addresses exported in kernel symbols file to the running kernel instance so they

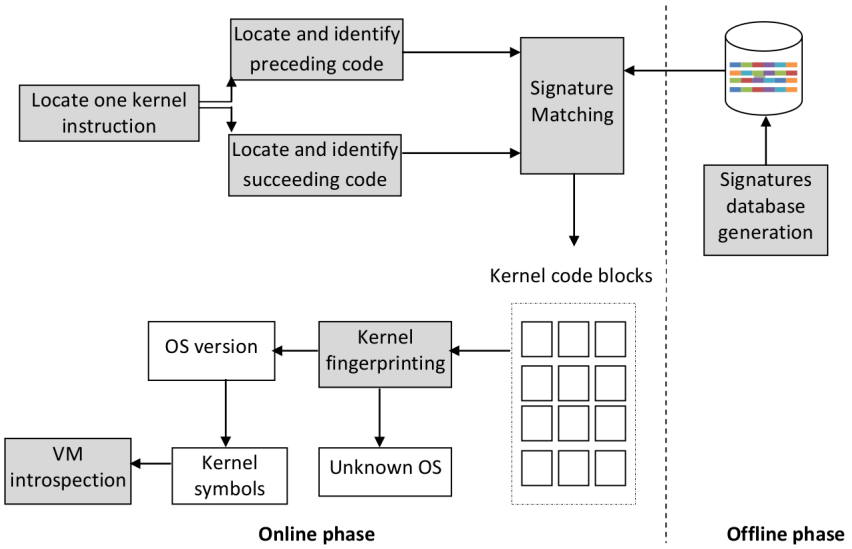


Figure 5.1 – An Overview of K-binID Approach for kernel binary code fingerprinting

can be used safely for VMI.

We detail in the next two sections how K-binID generates the multi-OS database for Linux kernels and how it pinpoints correctly VM kernel binary code version and customization.

5.4.3 Signatures database generation

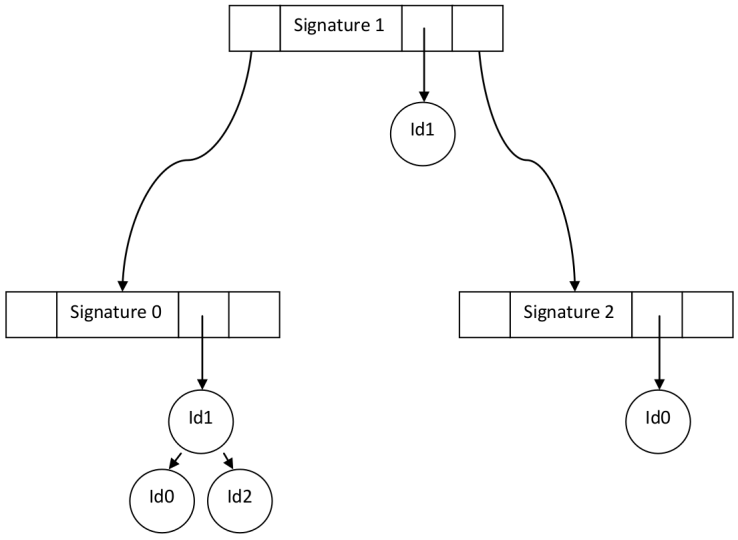


Figure 5.2 – Illustration of the multi-OS signature database structure. ID_i refers to a kernel version and customization identifiers

K-binID generates in the offline phase a multi-OS database of code blocks signatures with the help of kernel symbols file which enables K-binID to obtain easily main kernel code start and end addresses in the memory of a booted kernel. In kernels that does not employ base address randomization mechanism, kernel symbols are obtained from system.map file. For randomized kernels, kernel symbols are obtained from /proc/kallsym file which contains symbol addresses specific to the running kernel instance. To generate signatures database for a given kernel, K-binID disassembles main kernel binary code located between

startup_32 and _etext symbols which denote respectively start and end addresses of the main kernel code.

During the disassembly process, K-binID (1) divides the binary code into code blocks as we defined them in the last chapter, (2) computes hash signatures of each code block by taking into account only instruction opcodes and register operands to make signatures insensitive to kernel base address randomization and finally (3) associates in a data structure the computed code block signature to the identifier (i.e. version and customization) of the kernel for which signatures are generated. Each data structure is inserted in a red-black tree [124] that is sorted according to code blocks signatures.

At the end of this process, the created red-black tree which represents a mono-OS signatures database is written to disk and fused with a multi-OS signatures database that is also structured as a red-black tree and sorted by code blocks signatures. The use of a red-black tree structure enables to accelerate signature lookup during kernel fingerprinting and de-duplicate entries that have similar signatures. In case the multi-OS signatures database contains already a signature that is present in the mono-OS database, we add the identifier of the current kernel to kernel identifier list associated with concerned signature.

To accelerate pinpointing the kernel identifier that is shared all located code blocks in the online phase, we also structure kernel identifiers list associated with each code blocks signature as a red-black tree. In summary, each entry of the multi-OS signature database contains two fields whose structure is illustrated in figure 5.2: a key field which is a code block signature, and a second field which is the list of kernel identifiers that contain a code block with that signature.

5.4.4 Kernel Fingerprinting and Derandomization

During the online phase, K-binID identifies kernel version and customization of a running VM. K-binID approach is composed of two steps.

In the first step, K-binID uses our disassembly mechanism detailed in the last chapter to locate code blocks that succeed and precede a correctly located kernel instruction (I0). When locating each code block, K-binID identifies possible kernels to which belongs each located code block. This is done by computing the located code block hash signature then comparing it with the multi-OS signatures database. We refer to code blocks whose signatures match with the reference database as identified code blocks. The other blocks are referred to as non identified code blocks. To make code block hash signatures insensitive to ASLR, K-binID computes them by taking into account only instruction opcodes and register operands.

K-binID detect precise boundaries (i.e. start and end addresses) of main kernel code and hence stops backward and forward disassembly when it encounters a block signature that does not match with the multi-OS signatures database or a memory page that contains only null bytes. The result of this step is a list illustrated in figure 5.3 that indicates possible kernel to which belong each located code block.

In the second step, K-binID walks through the list produced in the last step and pinpoints the kernel identifier (i.e. version and customization) that is shared by all located and identified code blocks. In case there is an identified code block that does not share a kernel identifier with the rest of code blocks, K-binID assumes that VM main kernel code is unknown and has not been profiled the multi-OS signatures database. K-binID leaves investigating and handling the non identification of the main kernel code to other security VMI systems that use K-binID. In this step, we avoided looking for the identifier that is shared by most code blocks (i.e. rate-based matching algorithm) to avoid false positives which lead to using inappropriate

0xc900066f 0xc9000679 linux_3.14.4_o3r | linux_3.14.5_o3r
0xc900067e 0xc900068b linux_3.14.4_o3r | linux_3.14.5_o3r
0xc900068d 0xc900068d linux_3.14.1_notoptnotrand | linux_3.14.1_o1r | linux_3.14.4_o3r | linux_3.14.3_optrand | ...
0xc9000698 0xc90006a5 linux_3.14.4_o3r | linux_3.14.5_o3r

Figure 5.3 – Kernel version and customization fingerprinting from located and identified blocks of a VM running with Linux kernel 3.14.4 compiled with O3 optimization level and kernel base address randomization

kernel symbols for VMI. Actually, even if different kernels have code block signatures that are extremely similar, their symbol (i.e. function and data) addresses are generally completely different.

Once the VM main kernel binary code is identified, K-binID loads symbol addresses exported in the corresponding kernel symbols file. These addresses can be shifted from the actual ones in the running instance of the kernel due to ASLR mechanism. To identify the randomization offset, K-binID simply subtracts address of ia32_sysenter_target (system call handler) in symbols file from its corresponding instance address available in sysenter eip_msr_register. The obtained randomization offset (zero if kernel base address is not randomized) is added to loaded symbol addresses to adapt them to the running instance of the identified kernel so they can be used for VMI.

5.5 K-binID limitations

K-binID approach relies heavily on the signatures database to identify main kernel binary code in order to obtain symbol addresses of interest for VMI and in particular for reusing system calls and kernel functions. This implies that K-binID is unusable on kernels whose signatures can not be obtained such as kernel with custom patches for which there is no signatures and whose kernel symbols are unavailable.

The second limitation is that in kernels such as Linux, there are a lot of different versions and each one of them can be customized in multiple ways. This results in a big number of unique and different kernel symbols files which facilitates evading VMI systems that require using kernel symbols and presents a difficulty for K-binID to obtain all possible kernel signatures. This limitation can be reduced by automating downloading and compilation of existing kernel versions with all their possible customizations then generating the multi-OS signatures database from the generated object files. This procedure must however handles automatically load and run time kernel code patching [83][73] to generate reliable code block signatures.

5.6 Conclusion

In this chapter, we used our kernel disassembly mechanism to build K-binID, a kernel binary code identification system that enables the hypervisor to precisely fingerprint kernel version and customization regardless of challenges presented by ASLR or compiler optimizations. We showed that K-binID enables

the hypervisor (or another VMI system) to determine appropriate kernel symbols file and adapt exported symbol addresses to the kernel instance running inside the introspected VM so they can be used correctly for VMI and in particular for binary code reuse based approaches. We believe that K-binID approach is very suitable for usage in real world IaaS cloud environments thanks to its automation and OS independence.

6

Main kernel functions localization and identification

Chapter abstract

Kernel symbols file of a known kernel is very helpful for bridging the semantic gap. However, relying heavily on it limits the usability of VMI systems as symbols file can become unusable due to kernel updates and custom patches even if they modify only a small part of the known kernel binary code. In this chapter, we investigate locating main kernel functions directly in the memory of a running VM without necessarily having symbols file of the VM kernel. Located functions can be then instrumented by a VMI system to bridge the semantic gap for VMI with no need to access kernel symbols file.

We first motivate in this chapter our interest for locating main kernel functions and detail encountered challenges to do so. We introduce then major existing works for locating functions in a binary code and show their limitations. Finally, we present the design of NoGap; a system based on our main kernel disassembler which identifies most kernel function names and addresses in the binary code from properties of main kernel code blocks.

6.1 Introduction

In the last chapter we presented a kernel fingerprinting system that enables the hypervisor to determine appropriate kernel symbols from which function addresses can be extracted for binary code reuse VMI. However, the problem with kernel symbols is that they are very sensitive to modifications in main kernel code. So, even a small patch in the kernel code can shift almost all function addresses and symbols file of the kernel before the patch becomes unusable for introspecting the new kernel despite that most kernel code is still unchanged. This case is representative of kernel code modifications due to applying minor OS

updates and adding or removing kernel functionalities during kernel compilation. For instance, enabling or disabling machine hibernation functionality during kernel compilation adds or removes binary code from the main kernel and modifies hence most functions addresses without necessarily changing their binary code.

We focus in this chapter on the problem of locating and identifying directly kernel functions in the main kernel binary code so they can be used for VMI. Locating functions consists in discovering their start and end addresses (i.e. boundaries) in the binary code while identifying them is determining their names. We believe that locating and identifying directly functions in the main kernel binary code then instrumenting them (i.e. intercept, call and analyze) for VMI is the most efficient and practical approach to bridge the semantic gap. In particular, this approach relaxes the constraint of possessing and accessing kernel symbols to obtain function addresses of interest for VMI.

In this chapter we present NoGap, a VMI system that enables the hypervisor to locate and identify most main kernel functions in the memory of a running VM despite – the to be detailed – challenges presented by ASLR and compiler optimizations. NoGap uses our main kernel disassembler to divide the kernel binary code into code blocks from which kernel function boundaries and names are deduced using a novel code blocks grouping mechanism.

6.2 Background and problem statement

Functions in a binary code are of interest to many security applications based on binary analysis such as binary instrumentation [86] [43], vulnerability detection [103] [64] and control flow integrity [127] [51] [134]. In the context of VMI, VM activities related to all kernel aspects (memory, process, network, files, etc.) that the hypervisor is not aware of due to the semantic gap are performed by main kernel functions. In addition, kernel data which reflect VM states (e.g. process descriptors) are manipulated (i.e. created, accessed or deleted) by main kernel functions. So, the hypervisor can track VM activities of interest by intercepting execution of kernel functions that perform them. Similarly, addresses of kernel data can be tracked by intercepting execution of kernel functions that manipulate them. Also, the hypervisor can manipulate VM states of interest by injecting a call to suitable kernel functions. Finally, as demonstrated in VMI states of the art, the hypervisor can deduce information about data semantics by analyzing instructions of functions that manipulate these data. Hence, the hypervisor can considerably narrow the semantic gap by locating and identifying VM main kernel functions then instrumenting them.

However, locating boundaries of main kernel functions in VM memory from the hypervisor level is not obvious due to multiple difficulties. The first one is that contrary to a source code, a binary code does not explicitly mark boundaries of functions it contains. So there is a lack of semantics about function boundaries in the binary code. Also, despite that compilers can generate instructions that are specific to function starts (called function prologues), these instructions may not be present in the binary code due to compiler optimizations.

Figure 6.1 shows in the upper code snippet a non optimized code of sys_getpid kernel function and in the bottom code snippet an optimized code of the same function. Contrary to the non optimized version of sys_getpid, the optimized one does not include a function prologue and epilogue which mark a function

```
<sys_getpid> Linux kernel 3.13.1:
1:    0xc106be00    push ebp ⟵ start
2:    0xc106be01    mov ebp, esp
3:    0xc106be03    lea esi, [ds:esi]
4:    0xc106be08    mov eax, [fs:0xc1a4dfd4]
5:    0xc106be0e    mov eax, [eax+0x248]
6:    0xc106be14    mov eax, [eax+0x264]
7:    0xc106be1a    call 0x7cc0
8:    0xc106be1f    pop ebp
8:    0xc106be20    ret ⟵ end

<sys_getpid> Linux kernel 2.6.32:
1:    0xc103b7c5    mov eax, [fs:0xc1414454] ⟵ start
2:    0xc103b7cb    mov eax, [eax+0x150]
3:    0xc103b7d1    mov eax, [eax+0x170]
4:    0xc103b7d7    jmp 0x6c62 ⟵ end
```

Figure 6.1 – Code snippets of sys_getpid in non optimized code of Linux kernel 3.13.1 and in optimized code of Linux kernel 2.6.32

start and end. Moreover, the optimized version of sys_getpid ends with a jmp instruction instead of a ret instruction due to tail calling [96] compiler optimization. As we mentioned in the last chapter, these optimizations can be found in many version of famous Linux distributions and any Linux kernel can be configured and compiled to present these kind of non obvious function boundaries. In addition to these difficulties, there is in the context of VMI the challenge of locating the main kernel code itself which is solved by our disassembler presented in chapter 4.

6.2.1 Problem definition

Our main kernel disassembler enables the hypervisor to divide VM main kernel binary code and hence the functions it contains into code blocks that can be located regardless of compiler optimizations and ASLR. So, the remaining challenge that we address in this chapter is how to deduce function boundaries and names based on code blocks that compose them.

6.3 Related works

Several existing works on binary analysis addressed the problem of discovering function boundaries which are required in many security applications. Most of these works are based on the idea of searching for instructions that mark function starts, namely for function prologues. While these works share the same principle, they differ either in how they implement it and parse the analyzed binary code or in the functionalities they provide beyond discovering function boundaries (e.g. function parameters identification, library function naming).

IDA Pro Disassembler [14] and Dyninst [43] use recursive traversal to disassemble binary code of an executable file and manually provided patterns that denote function prologues. BAP [45], Rosenblum et al.

system [112], ByteWeight [37] and Shin et al. system [119] use learning techniques (e.g. neural networks) on a training dataset to collect automatically function start and end patterns (i.e. prologues and epilogues) in a training phase. In the analysis phase, theses systems use the collected function boundaries patterns to discover functions in the binary code.

The efficiency of these works for discovering function boundaries drops considerably when they are applied on optimized binary code [35] in which known function start and end instructions are not patterns generated by the compiler (e.g. to enhance code performance) but instructions that are specific to each function logic.

To address this limit, recent works considered relying only on control flow information which link basic blocks in the binary code in order to identify function boundaries. The first one called Nucleus which is a part of a PlayStation3 emulator that aims to discover function starts only in PS3 binary files. The approach of this system consist in dividing the binary code into code blocks similar to our definition then considers blocks that are called or unreachable as function start points. This lightweight approach focus only on finding function starts on PS3 platform. Finding function ends is not addressed by this system.

A recent system named also Nucleus [35] aims to discover function boundaries in x86/64 binaries in a compiler agnostic way by generating a Control Flow Graph (CFG) of basic blocks obtained by linearly disassembling the analyzed binary code. Nucleus partitions then the CFG into sets of linked basic blocks that denote discovered functions. To do so, Nucleus considers basic blocks that are unreachable or reached by a call instruction as function entry points (i.e. starts). On the other hand, Nucleus considers basic blocks in which the control flow stops as function exit points (i.e. ends). While this approach produces excellent results in terms of accuracy of function boundaries discovery, its execution time seems to take few minutes on main kernel binary code which is not practical in real world cloud environment especially if the VM execution is suspended during the analysis.

REV.NG [60] uses a similar approach to discover function boundaries in an architecture agnostic way. So, instead of working directly on the binary code which is architecture dependent, REV.NG uses Qemu to obtain an Intermediate Representation (IR) of that code which is architecture agnostic. REV.NG converts then Qemu IR to the intermediate representation of LLVM which is a collection of open source compilation technologies that facilitates code analysis and recompilation [25]. At this point, REV.NG builds a CFG from the architecture-agnostic IR of LLVM then partitions the CFG into functions similarly to Nucleus. Similarly to Nucleus, REV.NG performs well in discovering function boundaries, however it takes 45s to analyze a binary file of 150KB. This leads hence to an execution time of several minutes to analyze binaries of several megabytes such as main kernel code.

Systems presented in this section are all designed to work on executable files in which the start of the code section is known. So these systems are not directly applicable from the hypervisor to analyze main kernel binary code for function boundaries discovery. Also, systems that rely on function prologues including those based on learning techniques are limited in efficiency and hence are not suitable for real world IaaS cloud environments.

Despite that recent systems which deduce function boundaries from CFG shows accurate results, their execution time is still not convenient for real world IaaS cloud environments Finally, all these systems focus mainly on function boundaries discovery and do not aim to name discovered functions except IDA

pro which uses a lightweight approach [13] specific for naming library functions used in the analyzed binary code.

Compared to these systems, NoGap approach is lightweight and very fast as it takes only from 1 to 6 seconds to analyze binary code of several megabytes. Moreover, NoGap approach based on our main kernel disassembler is designed to deduce at the same time function boundaries and names from code blocks that compose these functions regardless of compiler optimizations. We present details of this approach in the next section.

6.4 NoGap approach

6.4.1 Assumptions and threat model

We designed NoGap based on similar assumptions as K-binID. Namely, (1) having a signatures database that covers changes in some code blocks due to dynamic kernel code patches and (2) assuming that main kernel code integrity is verified. To instrument (i.e. intercept, call and analyze) main kernel functions, we assume knowledge of locations and semantics of function parameters. An existing work on binary analysis can help to obtain automatically this information [48], but details of this work are out of scope of the thesis. Regarding Function Call Injection (FCI), we focus in this thesis on handling safe calls to functions that leave the kernel in a consistent state after their execution (e.g. manipulate synchronization locks safely). Safely calling all kernel functions from the hypervisor is an unresolved challenge which – to our best knowledge – has not been addressed by any work on VMI.

6.4.2 Overview

NoGap approach for locating and identifying main kernel functions in the memory of a running virtual machine is composed of an offline and an online phases similarly to K-binID. In the offline phase, we generate a multi-OS database of hash signatures that cover code blocks of all main kernel functions from multiple OSes (detailed in section 6.4.3).

In the online phase, NoGap locates and divides the main kernel code into code blocks using our main kernel disassembly mechanism. NoGap identifies then names of possible functions to which belongs each located code block. Finally, NoGap groups into kernel functions code blocks that share similar associated function names and that are linked with control flow information. NoGap considers then the start address of the first block in each set as the start address of a kernel function. Similarly, the end address of the last block in each set is considered as the end address of a kernel function and the name shared by each set of code blocks is considered as the name of a kernel function.

NoGap enables then the hypervisor to intercept, call and analyze located and identified main kernel functions to bridge the semantic gap for VMI. We present hereafter details of each phase.

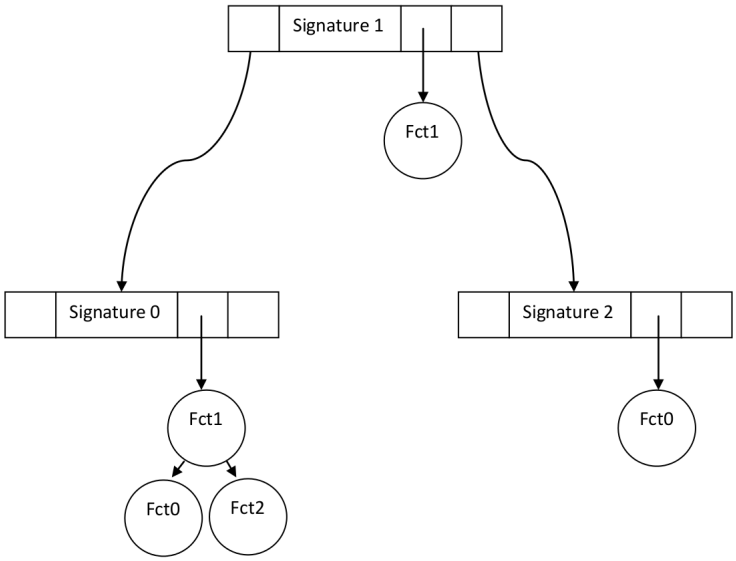


Figure 6.2 – Illustration of the multi-OS signature database structure. Fct$_i$ refers to a name of a kernel function

6.4.3 Signatures database generation

Multi-OS signatures database used by NoGap is generated from the memory of a booted kernel with the help of kernel symbols file in a process similar to the one of K-binID. More specifically, NoGap considers name and address of each text symbol between startup_32 and _etext as the name and entry point (i.e. start address) of a kernel function and the binary code between each text symbol and the next one as the body of that function. NoGap disassembles the binary code of each function and divides it into code blocks. The last non padding instruction that precedes the next symbol is considered as function last instruction. NoGap computes MD5 hash signature of function code blocks by taking into account only instruction opcodes and register operands to make signatures insensitive to ASLR. At this point, NoGap associates each computed hash signatures and the current function name in a data structure that is written in red-black tree sorted by code block signatures. In case the red-black tree contains already a similar signature, NoGap adds current function name to function names already associated with the existing signature in the red-black tree. The list of function names associated to each code block signature is also structure as a red-black tree to accelerate function name comparison during code blocks grouping.

Each element of the resulted red-black tree which represents a mono-OS signature database is written in a multi-OS signature database which is also structured as a red-black tree. Similarly, if the multi-OS database contains already a signature similar to the one associated with a code block of the current OS, NoGap fuses the red-black tree of function names of the current element with the one associated to the existing signatures in the multi-OS signatures database. In summary, entries of the multi-OS database illustrated in figure 6.2 are structured as following: a key field that represent a code block hash signature and a second field which is the list (red-black tree) of function names that contains a code block with that signature.

We present hereafter how NoGap uses this multi-OS signatures database to locate and identify main kernel function in the memory of a running VM.

6.4.4 Function boundaries and names identification

The online phase of NoGap comprises two steps: the first one consists in dividing the main kernel code into code blocks while the second step consists in deducing function boundaries and names from these code blocks. We detail next each step.

In the first step, NoGap uses our main kernel disassembler (introduced in chapter 4) to locate VM main kernel code from the hypervisor level and divide it into code blocks. For the need of the second step, NoGap records during main kernel disassembly encountered call instruction operands which designate function entry points.

After locating each code block, NoGap identifies possible names of functions in which belong the located code block by computing its signature in an ASLR insensitive way then comparing it with the multi-OS signatures database. We refer to code blocks whose signatures match with the NoGap database as identified code blocks. In order to avoid stopping the disassembler too soon due to code blocks signatures non matches and to support locating and identifying known main kernel functions (whose code block signatures are available in the database) in the memory of VMs running unknown kernels, NoGap stops forward and backward disassembly when its encounters a page that contains only null bytes or a signature non match of a code block that contains an invalid instruction.

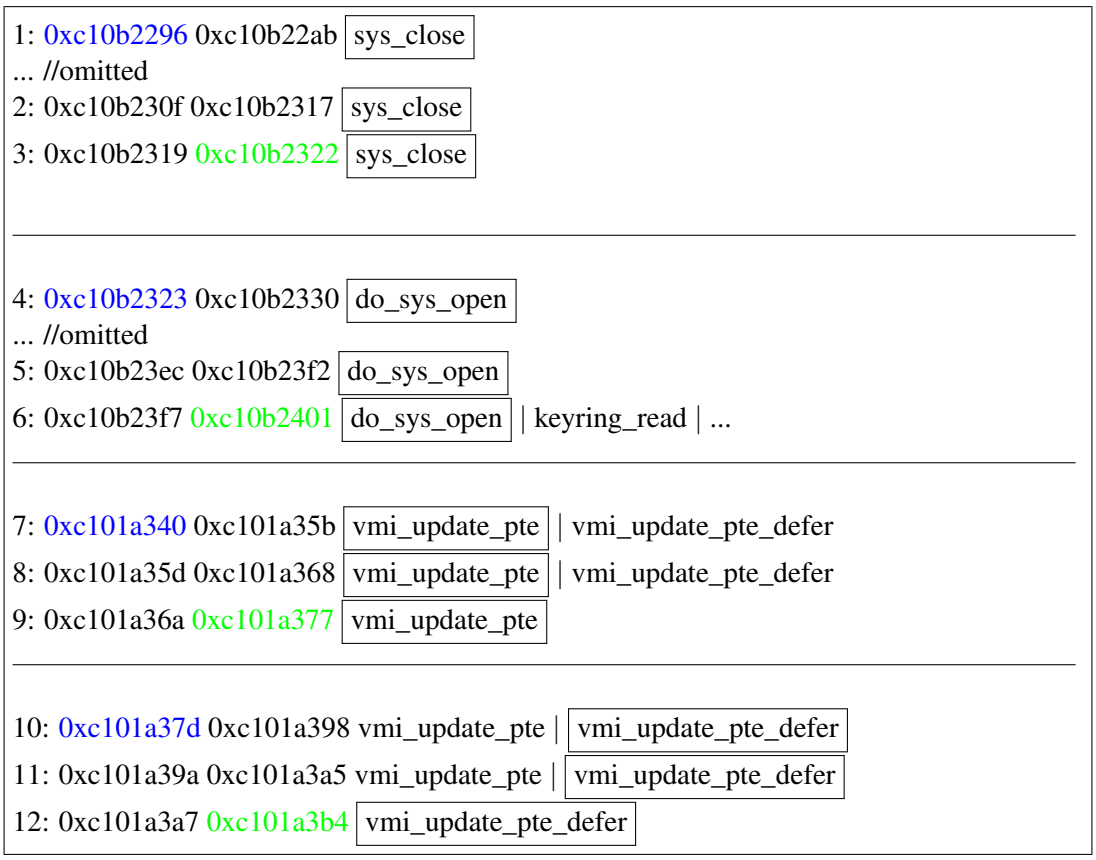


Figure 6.3 – Grouping identified blocks of Debian 6 (kernel 2.6.32) into functions

At the end of this step, NoGap produces a list that describes each located and identified function code block. A part of this list is illustrated in in figure 6.3 where in each line we can see respectively the start and end addresses of each identified code block in addition to a name list of functions that contain a code block whose signature is similar to the one of the identified code block.

In the second step, NoGap uses this list to deduce function boundaries and names by grouping located and identified code blocks into functions based on function name list associated to them and their control flow information. To do so, NoGap walks through the list of located and identified code blocks and considers as a function each set of successive code blocks that (1) share at least one function name and (2) where each code block in this set is reachable from the previous ones. NoGap considers the start address of the first code block in each set as the one of a main kernel function and the end address of the last code block in this set as the end address of that function. Also, the function name shared by code blocks of each set is considered by NoGap as the name of the function they represent.

In more details, NoGap considers the first code block in the list of located and identified code blocks as a function composed of one code block (we refer to this function hereafter as the current function and to this block as the current block). At this point, this function has the same attributes as the current block. Meaning that its start and end addresses are the same as the current block and its name is the list of function names associated with the current block. The goal of the next steps is to determine the rest of code blocks that belongs to this function and to refine – if possible – the list of function names of the current function into one function name.

NoGap moves forward to the next code block in the list (we refer to this code block as the current code block and to the previous one as the previous code block) and determines if this current code block belongs to the same function as the precedent block by checking if the current block (condition 1) has at least one function name in common with the name list associated to current function and (condition 2) is reachable from the previous code blocks. If it is the case, NoGap considers that these two code blocks belong to the same function. Accordingly, NoGap expands the current function and considers that its end address is equal to the one of the current code block. Also, NoGap reduces the name list of the current function to the set of function names in common between the current function and the current block.

If one of the two conditions is not verified, meaning that the current code block does not share any name with the name list of current function or it is not reachable from the previous code blocks, NoGap considers that the current function ends indeed at the previous block. In this case, NoGap considers that the current code block denotes a new function (which becomes the current function) whose attributes (i.e. start and end addresses in addition to its name) are the same as the current code block. NoGap moves then to the next code block and repeats this process of grouping located and identified code blocks into functions.

The code blocks grouping mechanism used by NoGap is hence based on name lists comparison and control flow inspection. While comparing name lists is a simple task, control flow inspection however is more complicated. We detail hereafter how NoGap checks if a current block is reachable or not from previous ones. In the following description, we assume that the current code block and the current function have some names in common, otherwise there is no need to check if the current code block is reachable from the previous ones as the first condition is not verified.

NoGap inspects the operand of a branch instruction that ends a code block each time it advances in the list of located and identified code blocks. The operands of branch instructions can be static immediate values or calculated dynamically at runtime based on the value of a register operand. Branch instructions are referred to as direct when their operands are static and indirect in the opposite case. The current prototype of NoGap focuses mainly on inspecting control flow of direct branch instructions and has a limited support for

indirect branch instructions. This limitation may lead to imprecise identification of boundaries and names of few functions.

	Direct		Indirect	
Conditional jump	Operand value*	Next instruction address	-	Next instruction address
Unconditional jump	Operand value*	-	Inspect jump table	Next instruction address
Call	Next instruction address*	-	Next instruction address*	-
Ret	-	-	-	-

*: Ignore if the address is a known function entry point.

Table 6.1 – Rules used by NoGap to deduce reachable addresses from a block end instruction

Table 6.1 recaps the different rules used by NoGap to determine reachable addresses inside a function (i.e. intra-procedural flow) according to the type of instruction that ends a code block. Each line details rules that concern a type of a branch instruction while each column specifies rules for direct and indirect branch instructions of that type. We detail hereafter each one of these rules.

Conditional jump: transfers the execution flow to the address in the instruction operand if specific flags are set in the state register of the CPU (i.e. checked condition is verified). Otherwise, the execution flow continues to the next instruction. For a direct conditional jump, NoGap record both reachable addresses but ignores the address in the instruction operand if it points to a known function entry point. That is, this address is not an intra-procedural flow. For an indirect conditional jump, NoGap records only the address of the next instruction.

Unconditional jump: transfers the execution flow to the address in the instruction operand independently of flags of the state register. For a direct unconditional jump, NoGap record the address in the instruction operand except when it points to a known function entry point. NoGap ignores then this address as it represents a tail call instruction (not an intra-procedural flow).

For an indirect unconditional jump, NoGap checks if the instruction operand contains a jump table address or not. A jump table is an array that contains addresses of basic blocks used to implement switch statement based on the value of a register operand [88]. NoGap detects a jump table if the operand of an unconditional jump contains a pointer sized value (designates the address of the jump table) added to a register that is multiplied by a pointer size (i.e. 4 bytes for 32bits kernels). To identify the size of a jump table, NoGap employs the approach used in [90]. More specifically, NoGap locates the last instruction that precedes the unconditional jump and compares the used register in the unconditional jump with an immediate value. This latter value is considered by NoGap as the size of the jump table. Having the address and the size of the jump table, NoGap records addresses contained in the jump table entries.

If no jump table is detected in an indirect unconditional jump, NoGap records only the start address of the code block that come after the indirect unconditional jump.

Call instruction: transfers the execution flow outside the current function to the address in the instruction

operand then returns back to the next instruction in the current function. For both direct and indirect calls, NoGap ignores the instruction operand and records the address of the next instruction except if it is reached by a call instruction. In this case, this means that this call instruction is the end of the current function which is a non returning function.

Return-like instruction: the execution flow inside the function stops at return-like instructions (i.e. ret, iret, sysexit, ...). For this type of instructions, NoGap does not record any address (no intra-procedural flow from this instruction).

During code blocks grouping, each time NoGap considers a code block as the start of a new kernel function, it records reachable addresses from this code block using the just detailed rules before it moves to the next one. If the start address of the current code block appears among reachable addresses of the previous code block, then NoGap considers that this code block is reachable from the previous one. Also, if the start address of the current code block is lower than the highest address reachable with an intra-procedural flow, NoGap considers also that this code block is reachable from the previous one and hence belongs to the current function. In these two cases, NoGap adds reachable addresses from this code block to the already recorded ones before moving to the next code block. NoGap continues walking the list of code blocks and recording reachable addresses as long as it considers that the current block is reachable from the previous ones.

When NoGap encounters a code block whose start address does not appear among the set of recorded reachable addresses or is not lower than highest address reachable with an intra-procedural flow, NoGap considers that this current code block is not reachable from the previous ones and hence does not belong to the current function. At this point, NoGap drops all recorded reachable addresses and repeat the just described process.

At the end of the online phase, NoGap inserts functions constructed by grouping located and identified code blocks into two red-black trees sorted by function start address and by function name. The goal of having these two red-black trees is to accelerate finding functions by name and by start address when instrumenting them during VM introspection. We detail hereafter possible function instrumentation mechanisms.

6.5 Kernel functions instrumentation

NoGap enables the hypervisor (or other VMI applications based on NoGap) to bridge the semantic gap by intercepting, calling and analyzing located and identified kernel functions. These instrumentation mechanisms have been already employed by some existing VMI systems as we showed in chapter 3 but mostly on system calls only. We present hereafter how NoGap implements each one of these mechanism to instrument main kernel functions.

6.5.1 Function execution interception

NoGap enables the hypervisor to intercept calls and returns of located and identified main kernel functions to actively monitor VM activities and to track addresses of manipulated (accessed or created) kernel data. NoGap implements function execution interception using a breakpoint (int3) injection mechanism similarly to existing VMI and in-VM systems such as DRAKVUF [87] and Kprobes [85]. DRAKVUF focus

on function calls (i.e. entry points) interception only and Kprobes requires to intercept execution of function starts in order to intercept their returns using the return address in the stack. NoGap enables to intercept both function calls and returns by patching their entry and exit points with a breakpoint instruction. We detail next NoGap implementation for function calls and returns interception.

To intercept calls to a given function, NoGap patches function entry point instruction with a breakpoint instruction which causes the VM to trap to the hypervisor when the injected breakpoint instruction is executed. Following a function call interception, NoGap perform actions (defined by a security application) depending on the intercepted function name or address and the values of its parameters then emulates the saved instruction and resume VM execution.

Function return interception is performed and handled by NoGap using a similar mechanism that is applied for function exit points. In case the function ends tail calling another function, NoGap intercepts returns of the tailed called function instead of the first one.

To disable interception of a function call or return, NoGap replaces back the injected breakpoint instructions with the corresponding saved instructions.

6.5.2 Function call injection

Kernel functions constitute an Application Programming Interface (API) that enables the main kernel (and also kernel modules) to safely read and modify kernel data and to perform different type of tasks to manage among other things memory (e.g. allocation, page protection flags), processes (creation, scheduling) and files (e.g. open, close). Due to the absence of details about guest OS internals at the hypervisor level, it is difficult for the hypervisor to perform actions inside the VM similarly to the kernel. Function Call Injection (FCI) mechanism provides a way for the hypervisor to safely read and modify kernel data and to perform tasks that are feasible by kernel functions. FCI can be done synchronously to respond to a specific event (e.g. function interception) or asynchronously such as periodically or lunched by an administrator.

The current prototype of NoGap support only asynchronous FCI to kernel functions that leave the kernel in a consistent state after their execution. Ensuring automatically kernel state consistency during synchronous FCI is challenging as it depends on manipulated kernel locks, interruptibility and internals of both the currently executing kernel code and the function that the hypervisor wants to call. Supporting both synchronous and asynchronous FCI to all kernel functions is still an unresolved challenge that – to our best knowledge – has not been yet addressed in VMI state of the art.

To avoid causing exception and inconsistencies inside the VM kernel, NoGap triggers asynchronous FCI only when the VM reaches the code block that comes after kernel segment registers are set in system calls handler pointed to by sysenter_eip_msr register. When performing a FCI, NoGap intercepts VM execution when it reaches the address of that code block in system calls handler using the mechanism detailed in the previous section. Following the interception of system calls handler, NoGap injects a call to the specified function using similar mechanism to the one used by IntroVirt [82] detailed in §3.2.4. When no further FCI is needed, NoGap disable interception of the system call handler.

6.5.3 Function instruction analysis

The logic of each kernel function is constituted by semantics of instruction opcodes and operands that form that function. This motivates the idea that the hypervisor can deduce further semantic information about guest OS internals that are revealed by semantics of kernel function instructions. We presented in chapter 3 how Argos deduces kernel data semantics by analyzing instructions of functions that manipulate these data. We present in chapter 8 how NoGap enables the hypervisor to obtain addresses of global kernel data and to deduce offsets of kernel data of interest through the analysis of instructions of suitable kernel functions.

6.6 NoGap limitations

In this section we present limitations of presented NoGap approach for locating and identifying main kernel functions.

Due to some imprecisions during inspection of code blocks control flow (e.g. undetected tail calls), NoGap can misidentify the start or the end of some kernel functions. This limitation can be addressed by improving current prototype control flow inspection.

A second limitation of the current prototype of NoGap is its inability to identify one name for some functions while grouping identified code blocks due to two reasons. The first reason is that in the Linux kernel, some functions have exactly the same code but their different names (i.e. duplicated functions). The second reason is that some functions have similar opcodes and register operands in their instructions (i.e. same logic) but manipulate different addresses. Because NoGap ignores addresses and offsets when computing a code block signature, code blocks of functions that have the same logic will have the same signatures. These two reasons make NoGap incapable of deducing one function name during code blocks grouping as all the code blocks of the concerned functions share multiple names. We believe that this limitation can be solved for most concerned functions by refining their name lists using invariant information about function call and data use relationships shared by kernel functions.

Finally, despite that NoGap reduces the dependence on kernel symbols as it enables to locate and identify known functions in the memory of a VM running an unknown kernel, the multi-OS database should be continuously enriched with function code blocks signatures from different kernel versions and customizations to ensure reliable identification of function boundaries and names.

6.7 Conclusion

In this chapter, we presented the design of NoGap, a VMI system built on our main kernel disassembler that enables the hypervisor to locate and identify main kernel functions in the memory of a running VM. NoGap approach is resistant to challenges presented by compiler optimizations and ASLR and enables to identify boundaries and names of most known functions even on unknown kernels whose kernel symbols are not available. We believe that NoGap approach for bridging the semantic gap is even more suitable for usage in real world IaaS cloud environments than K-binID as it is not strongly tied to the availability of kernel symbols.

Evaluation

7

Evaluation

7.1 Introduction

We implemented a prototype of the different contributions presented in this thesis with 5,700 Lines Of Code (LOC) on KVM hypervisor and 500 LOC on Qemu for the generation of kernel code blocks signatures needed for the different systems. This implementation include our main kernel disassembler, K-binID, NoGap and the mechanisms of kernel functions instrumentation in addition the use case presented in the next chapter. In order to disassemble and inspect kernel binary code from the hypervisor, we ported Udis86 disassembler library [26] to kernel space with a few patches. In this chapter we present evaluation results of our different systems (main kernel disassembler, K-binID and NoGap) presented in this thesis.

The evaluation of our contributions were performed with an HP Zbook 15 mobile workstation configured with an Intel i7 CPU and 16GB of RAM and running Ubuntu 14.04 64 bits with 3.13.0 kernel. We present hereafter a grouped evaluation of K-binID and our main kernel disassembler.

7.2 K-binID evaluation

We present in this section evaluation results of K-binID for main kernel code blocks localization and signature matching (i.e. identification) in addition to the identification of VM kernel binary code (table 7.1). We also evaluated K-binID for the identification of main kernel code boundary addresses, the derandomization of kernel objects addresses for usage in VMI applications and its execution time for the identification of VM kernel binary code (table 7.2). In addition, we report in this evaluation measured similarities of code block signatures in some of our test kernels to demonstrate the precision and the reliability of K-binID approach (table 7.3). Finally, to confirm the precision and the reliability of K-binID approach, we present evaluation results of K-binID when launched on unknown test kernels (tables 7.4 7.5 7.6 7.7).

K-binID evaluation was performed on 34 different old and recent Linux kernels configured with a variety

of compiler optimization levels (O1 - O3) and kernel base address randomization options. Most of our test kernels were compiled with aggressive compilation options that remove function prologues and epilogues and activate tail calling optimization option. All evaluation results related to kernel code blocks localization and identification were obtained by considering an intercepted 'mov cr3, eax' instruction as I0 and by having the multi-OS signatures database in RAM.

7.2.1 Code blocks localization and kernel binary code identification

Table 7.1 shows our evaluation results of K-binID and main kernel disassembler for kernel code blocks localization and main kernel binary code identification. Table 7.1 shows in column 1 and 2 version and customization of our test kernels, in column 3 the number of kernel blocks located when generating mono-OS databases of our test kernels, in column 4 the number of entries in each mono-OS signatures database which corresponds to number of unique code blocks signatures of each kernel and in column 5 the size in megabytes of each mono-OS database. The fusion of the mono-OS databases produced a multi-OS signatures database of our test kernels whose size is 88,4MB. Thanks to the non duplication of common entries between the mono-OS signatures databases, the size of the multi-OS database is 47,8% of the sum of mono-OS database sizes (184,9MB). Table 7.1 shows also in column 6 the number of located code blocks during kernel identification and in column 7 and 8 the number and the rate of identified code blocks, i.e. the code blocks whose signatures match with the multi-OS database. We can see in these later columns that K-binID locates and identifies almost all code blocks of all our test kernels regardless of their customization options. The non identification of some code blocks (at most 2 non matches) is due to data bytes outside the main kernel code that are confused with code blocks and whose signatures does not match with the database (i.e. stopping condition).

In column 10 and 11, table 7.1 shows the number and the rate of code blocks that are located with our backward disassembly presented in section 4.3.4 of chapter 4. To verify location correctness of these code blocks, we compare their start and end addresses with a list of code block locations that are obtained by disassembling – in a normal way – binary code between identified kernel code start and end. As shown in column 12, our backward disassembly makes at most one error per kernel which is very low and insignificant. We inspected these backward disassembly errors, we found that they are due to data bytes that precede the main kernel code which causes the disassembler to miss the first block of concerned kernels. These disassembly errors are hence due to the presence of some data bytes before some kernels code instances and not due to a flaw in the design of our backward disassembler. These data bytes are always located in page directory space that precedes the one of the main kernel code and are interpreted as code blocks which a lot of them contain invalid instructions. They are hence easily detected, discarded during kernel identification and removed from our evaluation results.

These results demonstrate hence the efficiency and the reliability of our main kernel disassembly mechanism presented in chapter 4.

In the last column of table 7.1, we can see that K-binID identifies correctly all our test kernels and accurately differentiates between two customizations of a same version regardless of compiler optimizations and kernel base address randomization. For our test kernels 3.14.2, K-binID outputs identifier of both tested kernels as they as they have 100% of similarity in terms of code blocks signatures and non randomized code

1. Kernel version	2. Options	3. Number of blocks when generating DB	4. Number of DB entries	5. DB size (MB)	6. Number of located blocks	7. Number of matches	8. %	9. Number of non matches	10. Number of blocks located backward	11. %	12. Number of blocks located incorrectly backward	13. OS identification
2.6.32	O2/NR	182657	68830	2,5	182658	182657	99,999	1	9857	5,396	0	OK
3.2.0	O2/NR	218157	81059	3,7	218158	218157	99,999	1	11257	5,160	0	OK
3.2.0	O2/NR	218465	81157	3,7	218466	218465	99,999	1	11251	5,150	0	OK
3.14.1	NO2/NR	326536	110091	3,6	326537	326535	99,999	2	12744	3,903	1	OK
3.14.1	O2/R	320900	116060	5,3	320901	320900	99,999	1	318116	99,132	0	OK
3.14.2	O2/NR	320917	116061	5,3	320918	320916	99,999	2	12492	3,893	1	OK*
3.14.2	O2/R	320917	116061	5,3	320918	320917	99,999	1	318133	99,132	0	OK*
3.14.3	O1/R	322179	105517	5,0	322180	322179	99,999	1	12675	3,934	0	OK
3.14.3	O2/R	320978	116082	5,5	320979	320978	99,999	1	318194	99,132	0	OK
3.14.4	O3/R	369980	125341	5,9	369981	369980	99,999	1	14317	3,870	0	OK
3.14.4	O2/R	320973	116077	5,5	320974	320973	99,999	1	318189	99,132	0	OK
3.14.5	O3/R	370060	125371	5,0	370061	370060	99,999	1	14317	3,869	0	OK
3.14.5	O2/R	321029	116099	5,5	321030	321029	99,999	1	318245	99,132	0	OK
3.14.10	O2/R	321205	116143	5,6	321207	321206	99,999	1	12454	3,877	0	OK
3.15.1	O2/R	323847	116843	5,5	323848	323847	99,999	1	12585	3,886	0	OK
3.15.3	O2/R	323862	116849	5,5	323863	323862	99,999	1	321066	99,136	0	OK
3.15.5	O2/R	323915	116867	5,5	323916	323915	99,999	1	12585	3,885	0	OK
3.15.7	O2/R	324018	116885	5,5	324019	324018	99,999	1	12580	3,882	0	OK
3.15.9	O2/R	324027	116891	5,5	324028	324027	99,999	1	321231	99,137	0	OK
4.0.1	O2/NR	405814	130857	5,9	405816	405814	99,999	2	16894	4,163	1	OK
4.0.1	O2/R	386653	127527	5,9	386655	386653	99,999	2	385048	99,584	1	OK
4.0.2	O2/R	386784	127596	5,9	386786	386784	99,999	2	15679	4,054	1	OK
4.0.4	O2/R	386810	127603	5,9	386812	386810	99,999	2	82448	21,315	1	OK
4.0.5	O2/R	386876	127646	5,9	386878	386876	99,999	2	385271	99,585	1	OK
4.1.5	O2/R	391129	128871	5,9	391130	391129	99,999	1	16465	4,210	0	OK
4.1.6	O2/R	391135	128868	5,9	391137	391136	99,999	1	389531	99,589	0	OK
4.1.7	O2/R	391082	128866	5,9	391084	391082	99,999	2	16465	4,210	1	OK
4.1.8	O2/R	391145	128859	5,9	391147	391145	99,999	2	16473	4,211	1	OK
4.2.1	O2/R	395626	130315	6,0	395628	395626	99,999	2	394034	99,597	1	OK
4.2.2	O2/R	395658	130320	6,0	395660	395658	99,999	2	16690	4,218	1	OK
4.2.3	O2/R	395762	130369	6,0	395764	395762	99,999	2	394170	99,597	1	OK
4.2.4	O2/R	395839	130393	6,0	395841	395839	99,999	2	16689	4,216	1	OK
4.2.5	O2/R	395876	130409	6,0	395878	395876	99,999	2	394283	99,597	1	OK
4.4.0	NO2/NR	482087	143228	6,9	482088	482087	99,999	1	34971	7,254	0	OK

Table 7.1 – Evaluation results for code blocks localization and identification of main kernel code. O: Aggressively optimized, NO: Not aggressively optimized, R: Randomized, NR: Not Randomized

and global data symbol addresses. We do not consider this is as a misidentification as it does not lead to an introspection with incorrect addresses.

K-binID approach enables hence the hypervisor to precisely identify VM kernel binary code among a set of known ones and hence the appropriate kernel symbols file that can be used for introspection.

7.2.2 Main kernel boundaries identification

Kernel version	Options	Kernel start address	Kernel end address	Kernel code size (MB)	Identified start	Rectified identified start	Identified end	Randomization offset	Execution time(s)
2.6.32	O2/NR	c1000000	c127379b	2,45	c1000000	c1000000	c127379b	0	0,63
3.2.0	O2/NR	c1000000	c12cb100	2,79	c1000000	c1000000	c12cb100	0	0,92
3.2.0	O2/NR	c1000000	c12cbe8c	2,80	c1000000	c1000000	c12cbe8c	0	0,95
3.14.1	NO2/NR	c1000000	c1523ae5	5,14	c100000f	c1000000	c1523ae5	0	1,02
3.14.1	O2/R	dd000000	dd52a483	5,17	dd000000	dd000000	dd52a483	1C000000	3,96
3.14.2	O2/NR	c1000000	c152a6c3	5,17	c100000f	c1000000	c152a6c3	0	1,24
3.14.2	O2/R	dd000000	c152a6c3	5,17	dd000000	dd000000	dd52a6c3	1C000000	4,17
3.14.3	O1/R	d9000000	d94c3c49	4,76	d9000000	d9000000	d94c3c49	18000000	0,96
3.14.3	O2/R	c6000000	c652aac3	5,17	c6000000	c6000000	c652aac3	5000000	4,00
3.14.4	O3/R	c3000000	c35fe549	5,99	c3000000	c3000000	c35fe549	2000000	1,13
3.14.4	O2/R	c8000000	c852a9c3	5,17	c8000000	c8000000	c852a9c3	7000000	4,51
3.14.5	O3/R	c6000000	c65fea49	5,99	c6000000	c6000000	c65fea49	5000000	1,17
3.14.5	O2/R	cc000000	cc52ae43	5,17	cc000000	cc000000	cc52ae43	B000000	3,95
3.14.10	O2/R	c7000000	c752bb43	5,17	c7000000	c7000000	c752bb43	6000000	1,03
3.15.1	O2/R	cb000000	cb5376c7	5,22	cb000000	cb000000	cb5376c7	A000000	1,02
3.15.3	O2/R	d5000000	d5537a07	5,22	d5000000	d5000000	d5537a07	14000000	4,02
3.15.5	O2/R	cf000000	cf537cc7	5,22	cf000000	cf000000	cf537cc7	E000000	1,05
3.15.7	O2/R	d1000000	d1538287	5,22	d1000000	d1000000	d1538287	10000000	1,09
3.15.9	O2/R	ca000000	ca5381c7	5,22	ca000000	ca000000	ca5381c7	9000000	3,74
4.0.1	O2/NR	c1000000	c164ebd8	6,31	c100000f	c1000000	c164ebd8	0	1,29
4.0.1	O2/R	d9000000	d9606498	6,02	d900000f	d9000000	d9606498	18000000	6,96
4.0.2	O2/R	d9000000	d9606e98	6,03	d900000f	d9000000	d9606e98	18000000	2,71
4.0.4	O2/R	dc000000	dc607058	6,03	dc00000f	dc000000	dc607058	1B000000	1,89
4.0.5	O2/R	d7000000	d7607698	6,03	d700000f	d7000000	d7607698	16000000	5,79
4.1.5	O2/R	c7000000	c761a02a	6,10	c7000000	c7000000	c761a02a	6000000	1,26
4.1.6	O2/R	c9000000	c961a06a	6,10	c9000000	c9000000	c961a06a	8000000	4,85
4.1.7	O2/R	c8000000	c8619caa	6,10	c800000f	c8000000	c8619caa	7000000	3,52
4.1.8	O2/R	dd000000	dd619caa	6,10	dd00000f	dd000000	dd619caa	1C000000	1,27
4.2.1	O2/R	c2000000	c262e29a	6,18	c200000f	c2000000	c262e29a	1000000	5,23
4.2.2	O2/R	dc000000	dc62e45a	6,18	dc00000f	dc000000	dc62e45a	1B000000	2,17
4.2.3	O2/R	d1000000	d162ea5a	6,18	d100000f	d1000000	d162ea5a	10000000	6,33
4.2.4	O2/R	d8000000	d862f124	6,18	d800000f	d8000000	d862f124	17000000	3,08
4.2.5	O2/R	d5000000	d562f2e4	6,18	d500000f	d5000000	d562f2e4	14000000	5,69
4.4.0	NO2/NR	c1000000	c17a8ed3	7,66	c1000000	c1000000	c17a8ed3	0	1,79

Table 7.2 – Evaluation results for boundaries identification and derandomization of main kernel code

After kernel binary code identification, K-binID considers that start and end addresses of the last identified blocks that precede and succeed I0 as kernel start and end addresses. Table 7.2 shows in column 3 and 4 the start and end addresses for each test kernel (i.e. addresses of startup_32 and _etext symbols) and in column 5 the main kernel code size.

In column 6 we can see the kernel start address identified by K-binID for each test kernel. For some test kernels K-binID identifies correctly their start addresses while for other kernels K-binID identifies a start address that is slightly shifted from the correct one due to a disassembly error. As shown in column 7, these errors can be rectified knowing that Linux kernel code starts always at an address that is aligned with a page frame. So, correct main kernel start address = start address identified by K-binID AND 0xFFFFF000. In

column 8, we can see that K-binID identifies correctly main kernel code end address on all test kernels.

Hence, K-binID approach built on top of our main kernel disassembler is reliable for locating automatically main kernel boundaries from the hypervisor level.

Due to kernel base address randomization, kernel symbol addresses can be shifted by a random offset from actual addresses of the running kernel instance. To identify this random offset needed to adapt kernel symbol addresses to the running kernel instance, K-binID subtracts an address in kernel symbols file from its corresponding one in the running kernel instance (e.g. system call handler address found in sysenter_eip_msr, kernel start or end addresses). The obtained offset (zero if the kernel is not randomized) is then added to kernel symbol addresses in kernel symbols file of the identified kernel to adapt them to the running kernel instance. Randomization offsets of some instances of our test kernels can be seen in column 9 of table 7.2.

K-binID enables hence the hypervisor to deduce easily the randomization offset of an identified kernel binary code in order to use its kernel symbols for introspection.

We measured K-binID execution time for each test kernel from the identification of I0 until identification of main kernel code version. The obtained results are reported in column 10 of table 7.2. As we can see, K-binID takes from 1 to 7 seconds to locate the main kernel and identify its binary code. This execution time is consumed mostly during the backward disassembly step and increases according to the size of the main kernel binary code and mostly to distance between I0 and the kernel start. This can be confirmed by comparing table 7.2 and table 7.1 where we can see that K-binID execution time is small for test kernels where the rate of code blocks located with our backward disassembly is also small. Hence, K-binID execution time for the other kernels can considerably reduced by locating multiple possible I0 instructions (§4.3.2) then choosing the one whose address is the lowest in VM memory.

7.2.3 Kernels signatures and symbol addresses similarities

We measured similarity of 9 test kernels in terms of signatures of located code blocks and symbol addresses. Table 7.3 shows in the upper side of each cell the rate of located code blocks signatures that are common between the line kernel and the ones specified in each column. The bottom side of each cell shows the rate of common symbols that are in the same – non randomized – addresses in the compared kernels.

As we can see in the upper rates, the more compared kernels are close (resp. distant) in version and customization the more (resp. less) signatures of their code blocks are similar. We observe also in this results that the similarity of code block signatures is very close to 100% among some of test kernels that are very close in version and customization. On the other hand, we can see in the bottom rates that symbol addresses change completely across different kernels. Even between kernels for which similarity of their code blocks signatures is close to 100%, only 30% – at most – of their symbols are on the same addresses.

These similarity results confirms again the precision and the reliability of K-binID and proves the necessity of having a complete coverage of VM main kernel code for precise and reliable kernel binary code fingerprinting. Also, misidentifying kernel binary code leads certainly to using incorrect kernel symbol addresses which may crash the VM kernel and also the hypervisor.

Kernels	Debian 6	Debian 7	Debian 7 updated	3.14.2 o2r	3.14.3 o2r	3.14.3 o1r	4.2.1 o2r	4.2.2 o2r	4.4.0 No2Nr
Debian 6	100,00% 100,00%	64,48% 0,02%	64,48% 0,02%	55,79% 0,00%	55,80% 0,00%	52,80% 0,00%	56,47% 0,00%	56,46% 0,00%	49,79% 0,00%
Debian 7	62,95% 0,02%	100,00% 100,00%	99,53% 3,83%	59,38% 0,00%	59,39% 0,00%	55,13% 0,00%	58,13% 0,00%	58,13% 0,00%	50,32% 0,00%
Debian 7 updated	62,93% 0,02%	99,48% 3,83%	100,00% 100,00%	59,39% 0,00%	59,39% 0,00%	55,13% 0,00%	58,15% 0,00%	58,15% 0,00%	50,34% 0,00%
3.14.2 o2r	54,17% 0,00%	56,97% 0,00%	56,97% 0,00%	100,00% 100,00%	99,91% 27,73%	61,51% 0,03%	63,34% 0,00%	63,33% 0,00%	54,62% 0,00%
3.14.3 o2r	42,95% 0,00%	56,97% 0,00%	56,97% 0,00%	99,90% 27,72%	100,00% 100,00%	61,51% 0,03%	63,34% 0,00%	63,34% 0,00%	54,63% 0,00%
3.14.3 o1r	52,18% 0,00%	55,03% 0,00%	55,01% 0,00%	63,91% 0,03%	63,91% 0,03%	100,00% 100,00%	57,84% 0,00%	57,84% 0,00%	51,46% 0,00%
4.2.1 o2r	57,23% 0,00%	59,07% 0,00%	59,08% 0,00%	66,21% 0,00%	66,21% 0,00%	59,80% 0,00%	100,00% 100,00%	99,98% 25,84%	66,01% 0,00%
4.2.2 o2r	57,23% 0,00%	59,07% 0,00%	59,08% 0,00%	66,21% 0,00%	66,21% 0,00%	59,80% 0,00%	99,98% 25,84%	100,00% 100,00%	66,01% 0,00%
4.4.0 No2Nr	52,98% 0,00%	54,17% 0,00%	54,16% 0,00%	60,73% 0,00%	60,74% 0,00%	57,17% 0,00%	69,48% 0,00%	69,48% 0,00%	100,00% 100,00%

Table 7.3 – Code block signatures (upper rate) and symbol addresses (bottom rate) similarities among some test kernels

7.2.4 False positives evaluation

K-binID identifies main kernel binary code by pinpointing a kernel identifier that is shared by descriptors of all located code blocks. We saw in table 7.1 that this approach identifies precisely VM main kernel binary code with no false negatives or positives when the VM kernel is present in the multi-OS signatures database. We evaluate hereafter false positives of K-binID approach when the VM kernel is not present in the multi-OS signatures database. In this next evaluation, we ignore the theoretical case of having only one kernel in the database which leads always to false positives as the identified code blocks share necessarily one kernel identifier.

Kernel version	Number of located code blocks	Number of matches	%	Number of non matches	%	Output
Debian 7 standard	13521	13489	99,76	32	0,24	Unknown
3.14.1	323776	170474	52,65	153302	47,35	Unknown
3.14.1	12493	6563	52,53	5930	47,47	Unknown
3.14.2	318145	185843	58,41	132302	41,59	Unknown
3.14.2	318135	185842	58,42	132293	41,58	Unknown
3.14.3	319419	180669	56,56	138750	43,44	Unknown
3.14.3	318196	185880	58,42	132316	41,58	Unknown
3.14.4	14318	7313	51,08	7005	48,92	Unknown
3.14.4	318191	185890	58,42	132301	41,58	Unknown
3.14.5	366737	212584	57,97	154153	42,03	Unknown
3.14.5	318247	185915	58,42	132332	41,58	Unknown
3.14.10	12493	6563	52,53	5930	47,47	Unknown
3.15.1	321058	187356	58,36	133702	41,64	Unknown
3.15.3	321068	187342	58,35	133726	41,65	Unknown
3.15.5	321127	187375	58,35	133752	41,65	Unknown
3.15.7	321224	187449	58,35	133775	41,65	Unknown
3.15.9	321233	187455	58,35	133778	41,65	Unknown

Table 7.4 – False positives evaluation with a database of 2 kernels (Debian6 and an updated version of Debian7)

Kernel version	Number of located code blocks	Number of matches	%	Number of non matches	%	Output
3.14.1	323703	170535	52,68	153168	47,32	Unknown
3.14.1	318118	227142	71,40	90976	28,60	Unknown
3.14.2	318134	227138	71,40	90996	28,60	Unknown
3.14.2	318157	227146	71,39	91011	28,61	Unknown
3.14.3	319419	197293	61,77	122126	38,23	Unknown
3.14.3	318196	227165	71,39	91031	28,61	Unknown
3.14.4	366657	252843	68,96	113814	31,04	Unknown
3.14.4	318191	227158	71,39	91033	28,61	Unknown
3.14.5	366737	252862	68,95	113875	31,05	Unknown
3.14.5	318247	227139	71,37	91108	28,63	Unknown
3.14.10	318418	227101	71,32	91317	28,68	Unknown
3.15.1	12590	8577	68,13	4013	31,87	Unknown
3.15.3	321341	227616	70,83	93725	29,17	Unknown
3.15.5	321121	227538	70,86	93583	29,14	Unknown
3.15.7	321223	227537	70,83	93686	29,17	Unknown
3.15.9	321233	227548	70,84	93685	29,16	Unknown

Table 7.5 – False positives evaluation with a database of 3 kernels (Debian6, Debian7 and an updated version of Debian7)

Kernel version	Number of located code blocks	Number of matches	%	Number of non matches	%	Output
3.14.1	318118	227142	71,40	90976	28,60	Unknown
3.14.2	12497	8595	68,78	3902	31,22	Unknown
3.14.2	318135	227139	71,40	90996	28,60	Unknown
3.14.3	319419	197293	61,77	122126	38,23	Unknown
3.14.3	318196	227165	71,39	91031	28,61	Unknown
3.14.4	366657	252843	68,96	113814	31,04	Unknown
3.14.4	318191	227158	71,39	91033	28,61	Unknown
3.14.5	366737	252862	68,95	113875	31,05	Unknown
3.14.5	366737	252862	68,95	113875	31,05	Unknown
3.14.10	318418	227101	71,32	91317	28,68	Unknown
3.15.1	12590	8577	68,13	4013	31,87	Unknown
3.15.3	321058	227570	70,88	93488	29,12	Unknown
3.15.5	321121	227538	70,86	93583	29,14	Unknown
3.15.7	321224	227538	70,83	93686	29,17	Unknown
3.15.9	321233	227548	70,84	93685	29,16	Unknown

Table 7.6 – False positives evaluation with a database of 4 kernels

Kernel version	Number of located code blocks	Number of matches	%	Number of non matches	%	Output
3.14.2	320917	320778	99,96	139	0,04	Unknown
3.14.2	320918	320779	99,96	139	0,04	Unknown
3.14.3	319448	216723	67,84	102725	32,16	Unknown
3.14.3	320979	320546	99,87	433	0,13	Unknown
3.14.4	366657	319036	87,01	47621	12,99	Unknown
3.14.4	320974	320454	99,84	520	0,16	Unknown
3.14.5	366737	318914	86,96	47823	13,04	Unknown
3.14.5	318277	317482	99,75	795	0,25	Unknown
3.14.10	318448	316848	99,50	1600	0,50	Unknown
3.15.1	321058	313413	97,62	7645	2,38	Unknown
3.15.3	321121	313153	97,52	7968	2,48	Unknown
3.15.5	321121	313153	97,52	7968	2,48	Unknown
3.15.7	321224	312990	97,44	8234	2,56	Unknown
3.15.9	321233	312957	97,42	8276	2,58	Unknown

Table 7.7 – False positives evaluation with a database of 5 kernels

Tables 7.4 7.5 7.6 7.7 shows K-binID evaluation results when launched on unknown kernels using multi-OS databases that contain respectively signatures of 2, 3, 4 and 5 kernels other than the VM kernel. In these

tables we see that K-binID locates always thousands of main kernel code blocks whose signatures match with the considered database thanks to signature similarities between VM kernel and the ones present in the multi-OS database. Because in all test cases K-binID found some identified code blocks that do not share any kernel identifier, K-binID identified successfully that all the tested VMs are running a kernel whose signatures are not present in the multi-OS databases.

These results confirm again the precision and the reliability of K-binID approach for main kernel binary code identification.

7.3 NoGap evaluation

we performed an evaluation of NoGap presented in chapter 6 a variety of 17 old and recent Linux kernels. Most of our test kernels were compiled to omit function prologues and epilogues and activate tail calling optimization in addition to kernel base address randomization. Moreover, function code blocks signatures of some of our test kernels are not included in the used signatures database (i.e. unknown kernels) to evaluate identifying known functions on these unknown kernels.

In this section we present NoGap evaluation for localization and identification of known functions code blocks on known and unknown kernels (table 7.8). We present then evaluation results of NoGap for identification of kernel function entry points, boundary addresses and names (table 7.9). We evaluated separately identification of function starts and boundaries as some security applications need only function entry points while other ones may require knowledge of both function entry and exit points. Correctness of function starts, boundaries and names identified by NoGap is checked by comparing them to function addresses and names specified in symbols file of our test kernels.

Similarly to K-binID, reported results in this section were obtained by considering an intercepted ‘mov cr3, eax’ instruction as I0 and by having the multi-OS signatures database in RAM. We present next evaluation results of NoGap for code blocks localization then for function boundaries and names identification on known and unknown kernels.

7.3.1 Code blocks localization

Table 7.8 shows NoGap evaluation results for kernel code blocks localization and identification on our test kernels whose versions and customizations are specified in columns 1 and 2. For known kernels, the size of their mono-OS signatures database is specified in column 3. The fusion of these mono-OS databases produces a multi-OS database of 32,7MB which is 60% of the sum of mono-OS database sizes thanks to non duplication of entries with common signatures and function names.

Column 4 of table 7.8 shows the number of code blocks located by NoGap for each test kernel. These numbers exceeds slightly the ones reported in table 7.1 in the evaluation of K-binID as NoGap stops forward disassembly only when it encounters an invalid instruction. As shown in column 5, NoGap identifies function names associated with almost all located code blocks of test kernels whose signatures are present in NoGap database (we refer to these blocks as identified code blocks). NoGap identifies also almost all located code blocks of unknown kernels thanks to their large similarity with known kernels. The rate of non

Kernel	Option	Database size (MB)	Number of located blocks	Number of matches	Number of non matches	Kernel start address	Kernel end address	Identified start	Identified end	Execution time
Debian 6	O2NR	5,4	182659	182657 (99,999%)	2 (0,001%)	0xc1000000	0xc127379b	0xc1000000	0xc127379b	1,17
Debian 7	O2NR	6,5	218165	218157 (99,996%)	8 (0,004%)	0xc1000000	0xc12cb100	0xc1000000	0xc12cb100	1,38
Debian 7 updated*	O2NR	-	218478	217495 (99,550%)	983 (0,450%)	0xc1000000	0xc12cbe8c	0xc1000000	0xc12cbe8c	1,45
3.14.1	O2R	9,7	320902	320900 (99,999%)	2 (0,001%)	0xda000000	0xda52a483	0xda000000	0xda52a483	1,83
3.14.2*	O2R	-	320918	320861 (99,982%)	57 (0,018%)	0xc8000000	0xc852a6c3	0xc8000000	0xc852a6c3	4,71
3.14.3*	O2R	-	320980	320842 (99,957%)	138 (0,043%)	0xcc000000	0xcc52aac3	0xcc000000	0xcc52aac3	1,86
3.15.1	O2R	9,8	323849	323847 (99,999%)	2 (0,001%)	0xd2000000	0xd25376c7	0xd2000000	0xd25376c7	4,81
3.15.3*	O2R	-	323864	323554 (99,904%)	310 (0,096%)	0xc1000000	0xc1537a07	0xc1000000	0xc1537a07	4,84
3.15.5*	O2R	-	323919	323535 (99,881%)	384 (0,119%)	0xd6000000	0xd6537cc7	0xd6000000	0xd6537cde	1,91
4.0.2	O2R	11,1	386788	386784 (99,999%)	4 (0,001%)	0xc4000000	0xc4606e98	0xc4000000	0xc4606e98	2,21
4.0.3*	O2R	-	386814	386771 (99,989%)	43 (0,011%)	0xd0000000	0xd0606fd8	0xd0000000	0xd0606fd8	7,96
4.0.4*	O2R	-	386879	386817 (99,984%)	62 (0,016%)	0xde000000	0xde607058	0xde000000	0xde60708d	6,04
4.0.5*	O2R	-	386881	386437 (99,885%)	444 (0,115%)	0xd8000000	0xd8607698	0xd8000000	0xd8607698	8,25
4.2.1*	O2R	-	395628	395591 (99,991%)	37 (0,009%)	0xc8000000	0xc862e29a	0xc8000000	0xc862e29a	5,82
4.2.2	O2R	11,3	395660	395658 (99,999%)	2 (0,001%)	0xdf000000	0xdf62e45a	0xdf000000	0xdf62e45a	5,71
4.2.3*	O2R	-	395764	395629 (99,966%)	135 (0,034%)	0xd8000000	0xd862ea5a	0xd8000000	0xd862ea5a	3,24
4.4.0*	NO2NR	-	482091	413113 (85,692%)	68978 (14,308%)	0xc1000000	0xc17a8ed3	0xc1000000	0xc17a9098	21,68

*: Unknown kernel whose signatures have not been explicitly included in NoGap database.

Table 7.8 – NoGap code blocks localization and identification

identified code blocks shown in column 6 is significant only on our test kernel 4.4.0 as the used signatures database does not include signatures of any kernel with close version and customization to this test kernel.

Table 7.8 shows respectively in columns 7,8,9 and 10 start and end addresses of each test kernel in addition the ones identified by NoGap. As we can see in these columns, NoGap identifies precise kernel start and end addresses on most of our test kernels even for the unknown ones. Only for few test kernels (e.g. 3.15.5, 4.0.4 and 4.4.0) NoGap identifies a kernel end address that exceeds slightly the correct one due to the presence of some data bytes after kernel code section that are confused with code blocks and whose signatures match with our database.

These results confirms again the efficiency of our main kernel disassembler on which NoGap is built as most code blocks are located and possible function names in which these code blocks belong are identified even on unknown kernels if their versions and customizations are close to the ones of a known kernel.

The last column of table 7.8 shows NoGap execution time for each test kernel. Similarly to K-binID, NoGap execution time varies mostly between 1 and 8 seconds for most test kernels according to kernel code size and to the distance between I0 and kernel start. NoGap execution time is significant (21 seconds) only

on our test kernel 4.4.0. We believe this is due to the significant number of missing code blocks signatures for this kernel which leads frequently to worst time execution of the signature matching function and hence to an extended execution time of NoGap.

7.3.2 Function boundaries and names identification

Kernel	Option	Number of functions	Constructed functions	Correct function starts	Correct function boundaries	Constructed function / one function name	Correct function start / Correct function name	Correct function boundaries / correct function name	Usable functions	Incorrect function / one name	Correct function boundaries / multiple names
Debian 6	O2NR	17007	17117	16829 (98,32%)	16713 (97,64%)	12637 (73,83%)	63	12496 (73,00%)	73,37%	78 (0,62%)	4217 (24,64%)
Debian 7	O2NR	22338	22322	22019 (98,64%)	21918 (98,19%)	16503 (73,93%)	58	16358 (73,28%)	73,54%	87 (0,53%)	5560 (24,91%)
Debian 7*	O2NR	22364	23396	21905 (93,63%)	21646 (92,52%)	16883 (72,16%)	180	16064 (68,66%)	69,43%	639 (3,78%)	5557 (23,75%)
3.14.1	O2R	25340	25871	25106 (97,04%)	23396 (90,43%)	19666 (76,02%)	1394	17983 (69,51%)	74,90%	289 (1,47%)	5413 (20,92%)
3.14.2*	O2R	25341	25937	25105 (96,79%)	23392 (90,19%)	19694 (75,93%)	1397	17977 (69,31%)	74,70%	320 (1,62%)	5415 (20,88%)
3.14.3*	O2R	25345	26034	25104 (96,43%)	23385 (89,82%)	19739 (75,82%)	1403	17968 (69,02%)	74,41%	368 (1,86%)	5416 (20,80%)
3.15.1	O2R	25545	26044	25300 (97,14%)	23594 (90,59%)	19795 (76,01%)	1394	18116 (69,56%)	74,91%	285 (1,44%)	5478 (21,03%)
3.15.3*	O2R	25545	26499	25272 (95,37%)	23523 (88,77%)	19948 (75,28%)	1428	18042 (68,09%)	73,47%	478 (2,40%)	5478 (20,67%)
3.15.5*	O2R	25548	26634	25266 (94,86%)	23503 (88,24%)	19998 (75,08%)	1437	18015 (67,64%)	73,03%	546 (2,73%)	5478 (20,57%)
4.0.2	O2R	28604	28754	28119 (97,79%)	26572 (92,41%)	21937 (76,29%)	1114	20616 (71,70%)	75,57%	207 (0,94%)	5956 (20,71%)
4.0.3*	O2R	28604	28781	28116 (97,69%)	26568 (92,31%)	21947 (76,26%)	1115	20610 (71,61%)	75,48%	222 (1,01%)	5956 (20,69%)
4.0.4*	O2R	28604	28823	28116 (97,55%)	26562 (92,16%)	21961 (76,19%)	1118	20604 (71,48%)	75,36%	239 (1,09%)	5956 (20,66%)
4.0.5*	O2R	28605	28948	28109 (97,10%)	26543 (91,69%)	22013 (76,04%)	1129	20584 (71,11%)	75,01%	300 (1,36%)	5955 (20,57%)
4.2.1*	O2R	29092	29290	28598 (97,64%)	27013 (92,23%)	22497 (76,81%)	1153	21078 (71,96%)	75,90%	266 (1,18%)	5934 (20,26%)
4.2.2	O2R	29093	29252	28602 (97,78%)	27021 (92,37%)	22483 (76,86%)	1149	21086 (72,08%)	76,01%	248 (1,10%)	5935 (20,29%)
4.2.3*	O2R	29097	29398	28590 (97,25%)	27000 (91,84%)	22521 (76,61%)	1157	21064 (71,65%)	75,59%	300 (1,33%)	5935 (20,19%)
4.4.0*	NO2NR	35035	166783	630 (0,38%)	210 (0,13%)	37723 (22,62%)	53	108 (0,06%)	0,10%	37562 (99,57%)	95 (0,06%)

*: Unknown kernel whose signatures have not been explicitly included in NoGap database.

Table 7.9 – Kernel function boundaries and names identification

Table 7.9 reports NoGap evaluation results for kernel function boundaries and names identification for our test kernels whose version and customization are specified in columns 1 and 2 (unknown kernels are marked with *). Column 3 shows the number of functions in each test kernels which corresponds to the number of text symbols between startup_32 and _etext in system.map file and column 4 reports the number of functions constructed by NoGap after grouping located and identified code blocks into functions. The number of constructed functions by NoGap differs slightly from the number of text symbols in our known test kernels due to some imprecisions during code blocks grouping which causes either incorrect function merging or splitting. On unknown kernels, the number of constructed functions by NoGap always exceeds

the number of kernel text symbols due to the absence of control flow information of non identified code blocks. This leads NoGap to consider some code blocks that belong to a same function as belonging to different functions, hence the additional number of constructed functions.

Column 5 of table 7.9 shows the number and the rate of functions constructed by NoGap whose start addresses (i.e. entry points) are correct while column 6 shows the number and the rate of constructed functions whose boundaries (i.e. both start and end addresses) are correctly identified by NoGap. We precise that the reported rates in these two columns are computed in relation to the number of constructed functions by NoGap.

As we can see in these columns, start address of 93% to 98% of functions constructed by NoGap are valid function entry points on all test kernels that are known or close to known ones. That is, start addresses of these functions appear in symbols file of the test kernels. Also, boundaries of 88% - 98% of functions constructed by NoGap correspond correctly to function boundaries exported in kernel symbols file of kernels whose all or most signatures are present in NoGap database. Only on our test kernel 4.4.0 where just few functions constructed by NoGap have correct start and end addresses due to the absence of most signatures of that kernel in our used database.

The imperfection of NoGap results for kernels whose most signatures are present in the NoGap database is due to three main causes. The first one is related to imprecisions in control flow resolving such as non detecting some tail calls which causes NoGap to consider successive code blocks that belong to different functions as belonging to a same function resulting hence in missing at the same time function end and the start of the following function. The second cause is the absence of some control flow information for unknown kernels due to the non identification of some code blocks which leads to imprecise control flow resolving and hence imprecise function start and boundaries identification. The last cause is due to imprecisions in the Linux kernel symbols file itself where some function information are not exported. This leads to consider correct starts and boundaries of not exported function as incorrect because they do not appear in kernel symbols file which is considered by NoGap as a ground truth. Due to the absence of a precise ground truth, we inspected manually some results considered as incorrect. We found that a considerable number of supposedly incorrect function starts and ends on known kernels seems to correspond to correct starts and ends of non exported functions in kernel symbols file.

So despite some imprecisions, these results show that NoGap approach built on top of our main kernel disassembler enables to identify starts and boundaries of most kernel functions even on some unknown kernels regardless of challenges presented by aggressive compiler optimization and kernel base address randomization.

Table 7.9 reports also NoGap evaluation results for identification of constructed function names. Column 7 shows the number and the rate of – incorrect and correct – constructed functions for which NoGap identified one – correct or incorrect – name. These rates are also computed in relation to the number of constructed functions by NoGap. We can see in this column that the rate of functions for which NoGap identified only one name is on average 75% of the constructed functions for kernels whose most signatures are present in NoGap database. For the test kernel 4.4.0 whose most signatures are missing from NoGap database, this rate is only of 22%. This low rate is due to insufficient identified code blocks for refining names of constructed functions during code blocks grouping. We believe that this rate is a strong indicator for

NoGap to determine automatically if the analyzed kernel is completely unknown or close to a known kernel. That is, a low rate as for this test kernel indicates that the kernel is totally unknown, hence introspection with constructed functions is not safe and should be avoided.

Column 8 shows the number of constructed functions by NoGap whose only start address is correct and for which NoGap identified a one correct name. In column 9, we report the number and the rate of function whose boundary addresses and their name are correct. We call these two sets of functions as usable functions for security applications as they have one correct name and at least a correct function entry point. The rate of usable functions (in relation to constructed functions) shown in column 10 is of average of 74% for our test kernels whose most signatures are present in NoGap database. This rate is close to 0 for our test kernel 4.4.0 as the absence of its signatures from NoGap database leads to incomplete identification of function names that contain the located code blocks and hence to incorrect code blocks grouping.

The current approach of NoGap does not have a mean to distinguish automatically between usable constructed functions and the rest of them (i.e. false positives). So the current prototype of NoGap considers functions that have one name as usable functions. In column 10, we evaluate the rate of false positives for usable functions. That is, the rate of functions that have one name but whose start or boundary addresses are incorrect in relation to functions that have one name. As we can see in column 10, the rate of false positives for usable functions is at most 4% of constructed functions that have one name on test kernels whose most signatures are available in NoGap database. Again only on our test kernel 4.4.0 were almost all functions that have one name are false positives.

Finally, column 11 reports the number and the rate (in relation to constructed functions) of constructed functions whose boundary addresses are correctly identified by NoGap but for which NoGap associates multiple names. This rate is of average of 21% on kernels whose most signatures are present in NoGap database and close to 0 for the kernel 4.4.0 whose most signatures are not available in the database. The main cause behind associating multiple names to functions with correctly identified boundaries is the way signatures of located code blocks are computed. By ignoring addresses and offsets during code block signatures computation, signatures of non similar code blocks that have similar instruction opcodes and registers become identical which resulting in associating additional function names to these code blocks on top of the ones in which they really belong. This leads the concerned sets of code blocks that constitute functions to share more than one function name as if these functions were duplicated but with different names. We observed that the concerned functions are mostly short functions that contain only few code blocks. We also observed in NoGap results that the more code blocks a function has, the more is the chance of identifying its precise name. We believe that it is possible to refine names of most concerned functions using invariant information about function call and data use relationships shared by kernel functions.

Despite limitations and imperfections of current NoGap approach, presented evaluation results show that NoGap approach built on top of our main kernel disassembler is usable for identifying boundaries and names of most (70% - 74%) known kernel functions even on unknown kernels whose kernel symbols were not provided to NoGap. NoGap approach achieves this despite challenges presented by aggressive compiler optimizations and kernel base address randomization.

7.3.3 Comparison with Nucleus and REV.NG

We compared the rate of kernel function boundaries identification in NoGap (88% - 98%) with the one of function boundaries identification in executable binaries in Nucleus (95% - 98%) and REV.NG (97% - 98%) as reported in their papers. We precise that contrary to NoGap, these systems does not label located functions and were not tested on kernel binaries.

The rate of function boundaries identified correctly by NoGap is slightly less than the ones of REV.NG and Nucleus whose approaches are based on CFG partitioning. We believe that NoGap results – whose correctness is underestimated due to imprecisions in kernel symbols files – can be enhanced to become comparable to REV.NG and Nucleus results by improving control flow resolving in our grouping algorithm.

Also, despite some imprecisions in our results, NoGap is still more convenient for usage then REV.NG and Nucleus as its execution time is only few seconds for locating and labeling functions while REV.NG and Nucleus would require few minutes to only locate functions without labeling them.

8

Applications

Chapter abstract

To illustrate bridging the semantic gap with the three mechanisms for instrumenting kernel functions (i.e. interception, calling and analysis) identified using NoGap or obtained from appropriate kernel symbols file with the help of K-binID, we present in this chapter a VMI-based application for detecting and terminating hidden processes that are invisible in both kernel task list and output of *ps* command.

For the need of this application, we present in the first part of this chapter new learning-based binary analysis mechanisms that enables the hypervisor to obtain automatically global kernel pointers and data field offsets of interest through analysis of instructions of relevant kernel functions.

We detail then in the second part of this chapter the design of an automatic and widely kernel portable hidden process detector that is designed based on instrumentation of Linux kernel functions.

8.1 Global kernel pointers and offsets identification

8.1.1 Introduction

In chapter 6 we presented how NoGap enables the hypervisor to intercept and call located and identified kernel functions to obtain semantic information for VM security monitoring applications. However, semantic information obtained through these mechanisms are limited to the granularity of kernel functions. In other words, the hypervisor can not obtain semantic information that is not available directly during function call or interception. For instance, as in the Linux kernel there is no function that walks through kernel process list and returns at once all available process descriptors, it is then impossible for the hypervisor to obtain at once all existing process descriptors through function call or interception.

To address this limitation and to expand the semantic view that the hypervisor can obtain through kernel function instrumentation, we introduce in the first part of this chapter a preliminary work on a set of new

learning based binary analysis techniques that enables the hypervisor to identify automatically global kernel pointers and field offsets of data structures of interest through analysis of kernel function instructions that use them.

Kernel global pointers are of interest for VMI systems as they point to several important data structures such as process list, kernel modules list in addition to addresses of different sections in the kernel space. Moreover, several dynamic kernel data such as process descriptors are generally attached to a data structure pointed to by a global kernel pointer. Hence if the hypervisor can automatically identify these global kernel pointers in addition to their field locations (i.e. offsets) and semantics, then the hypervisor can obtain automatically highly fine grained and kernel specific semantic information. For instance, by identifying init_task global pointer in addition to next and pid field offsets in a process descriptor, then the hypervisor can walk automatically through process list pointed to by init_task and print PID of each process descriptor in that list. However, it is not obvious how to automatically identify these information directly from kernel binary code without accessing kernel symbols file.

Before we detail our approach for identifying global kernel pointers and field offset of interest, we present hereafter how existing approaches proceed to obtain these information then we highlight their limitations.

8.1.2 Related work

Most existing works on VMI obtain manually addresses of global kernel pointers from kernel symbols file. Moreover, they also obtain manually field offsets of interest from a technical documentation or with the help of an in-VM kernel module that prints needed information. The manual effort in these approaches is very tedious as it must be done for every supported kernel version and customization. So this manual effort is infeasible in real cloud environments that host a big number of different kernels.

Recently, Feng et al. proposed a system named ORIGEN [65] that aims to automate identification of global kernel pointers and field offsets of interest in new versions of a given binary code based on the similarity of its Control Flow Graph (CFG) with the one of a known version of the considered binary code. The idea of ORIGEN is to identify in an offline phase positions of instructions that use – already known – global kernel pointers and field offsets of interest in the CFG of a known binary code version. Then for newer versions of that binary code analyzed in an online phase, ORIGEN locates in their CFGs instructions that are in the same positions as in the CFG of the known version to extract from their operands needed global kernel pointers and field offsets. ORIGEN approach is hence built on a graph-based code searching technique. This implies that ORIGEN approach is less efficient for new binaries whose CFGs are not similar to the one of a known binary version, hence the risk of false positives and negatives for identification of needed global kernel pointers and field offsets. Also, ORIGEN graph-based code searching approach is time consuming and takes several minutes to identify few Linux global kernel pointers and field offsets according to authors evaluation. These limitations make ORIGEN non practical for usage in real world cloud environments.

Compared to ORIGEN, our approach based on static analysis of kernel function instructions that use wanted global kernel pointers and field offsets is more practical thanks to its simple design, reliable efficiency and fast execution time. We detail next our approach for identification of global kernel pointers and

field offsets of interest.

8.1.3 Design of our approach

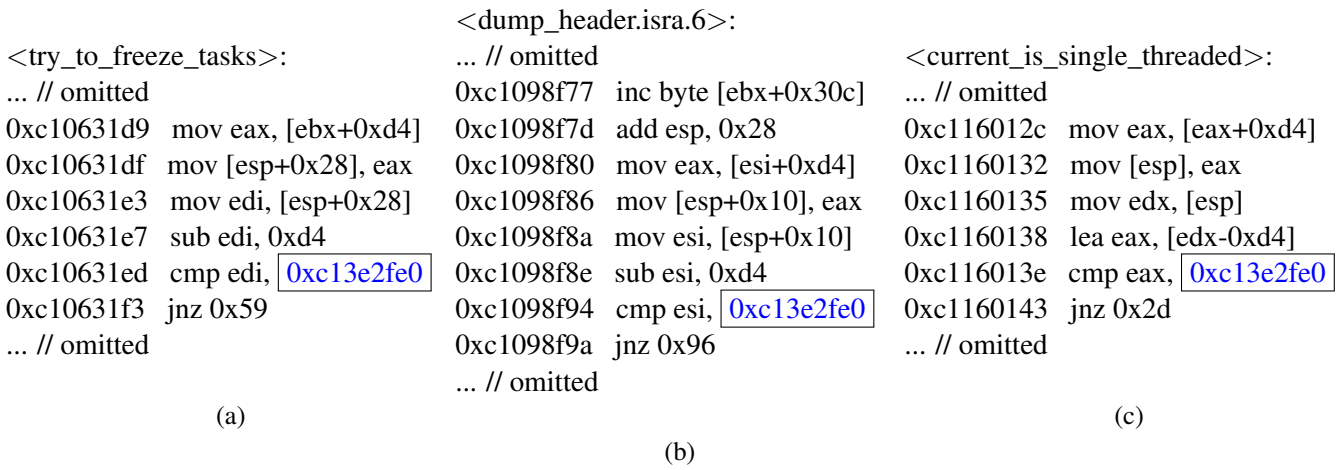


Figure 8.1 – Assembly code snippets that reference init_task in Debian 7

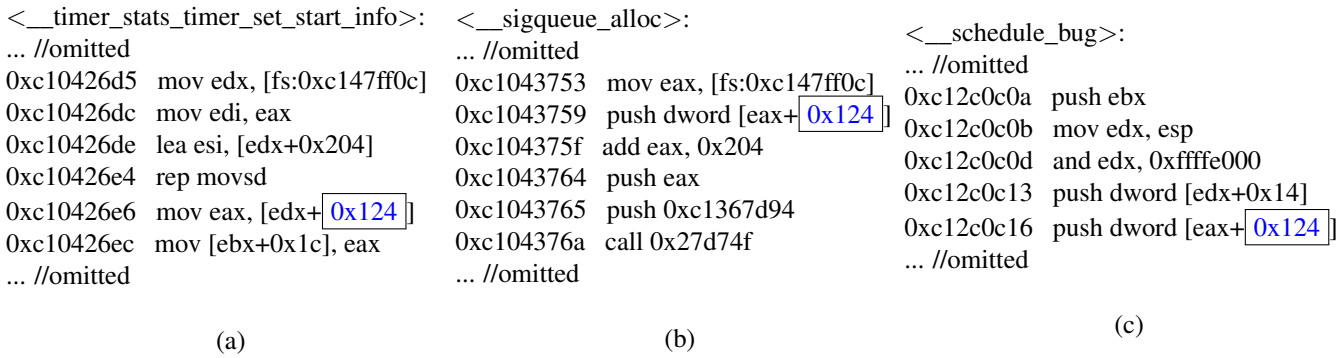


Figure 8.2 – Assembly code snippets that reference task_struct pid field in Debian 7

In the following description of our approach, we assume having boundary addresses and names of kernel functions (e.g. obtained through NoGap). We also assume in the design of our approach code integrity of the instrumented functions.

The key insight behind our approach is that global kernel pointers are produced during kernel compilation and their addresses are integrated by the compiler as immediate value operands in assembly instructions of kernel functions that use them. Figure 8.1 shows assembly code snippets from some Linux Debian 7 kernel functions that use the global kernel pointer init_task which is illustrated in text boxes. Similarly, field offsets in kernel data structures are produced during kernel compilation and integrated as constant values in memory operands of kernel function instructions that use them. Figure 8.2 shows another assembly code snippets from some Linux Debian 7 kernel functions that access the pid field of task_struct whose offset is illustrated in text boxes.

Based on the presented insights and illustrations, global kernel pointers and field offsets are hence integrated in operands of kernel function instructions that use them. To identify those of interest, our approach is composed of an offline learning phase and an online analysis phase.

The offline learning phase consists in generating – from kernel source or binary code – a list of functions that use wanted global kernel pointers and field offsets. In the current implementation of this preliminary work, we manually generate this list from kernel source code with the help of Linux 'grep' command to find a given field usage in kernel functions and Linux Cross Reference website [17] to find kernel functions that use a given global kernel pointer. To generate a robust and kernel portable list of functions that use global kernel pointers and field offsets of interest, we select only kernel functions that are long lived and have a stable behaviour across Linux versions.

To identify a global kernel pointer (resp. field offset) of interest in the online analysis phase, we extract pointer sized immediate values (resp. offsets in memory operands) contained in instructions of the list of kernel functions that use this global kernel pointer (resp. field offset). The immediate value (resp. memory offset) that appears in instructions of each analyzed kernel function corresponds hence to the global kernel pointer (resp. field offset) of interest. For some offsets referenced by kernel functions when accessing fields of global kernel data, it is possible to identify them from the immediate values that are close to a known global kernel pointer as they are in fact the sum of the global kernel pointer and the offset of the accessed filed.

8.1.4 Evaluation

Kernel version	init_task	next (hex)	prev (hex)	pid(hex)	Identification
Debian 7 (3.2.0)	0xc13e2fe0	D4	D8	124	✓
Ubuntu 12 (3.2.0)	0xc180b020	1B4	1B8	204	✓
3.14.1	0xd971b9c0	268	26C	2C4	✓
3.14.2	0xc971b9c0	268	26C	2C4	✓
3.14.3	0xd471b9c0	268	26C	2C4	✓
3.14.4	0xca71b9c0	268	26C	2C4	✓
3.14.5	0xce71b9c0	268	26C	2C4	✓
4.1.5	0xd887fa20	26C	270	300	✓
4.1.6	0xcf87fa20	26C	270	300	✓
4.1.7	0xc287fa20	26C	270	300	✓
4.1.8	0xc687fa20	26C	270	300	✓
4.2.1	0xd5899a80	26C	270	300	✓
4.2.2	0xdd899a80	26C	270	300	✓
4.2.3	0xcd899a80	26C	270	300	✓
4.2.4	0xd489ba80	26C	270	300	✓
4.2.5	0xd789ba80	26C	270	300	✓
Ubuntu 16 (4.4.0)	0xc1abda80	270	274	308	✓

Table 8.1 – Evaluation results for identification of init_task global kernel pointer in addition to pid, prev, and next field offsets

We implemented our presented approach as an extension of NoGap on KVM hypervisor and performed an preliminary evaluation on 17 Linux kernels for identification of the global kernel pointer init_task and offsets of fields pid, prev and next in task_struct needed for the use case of hidden process detection presented in the second part of this chapter. The Linux kernels used in this evaluation include a variety of

versions that range from 3.2.0 (released on 2012) to 4.4.0 (released on 2016). To demonstrate efficiency of our approach, we evaluated it also on kernels that employ kernel base address randomization [15] which causes global kernel pointers to change by a random offset after each VM reboot.

Table 8.1 reports evaluation results of our approach for identification of init_task on our test kernels from functions do_coredump, show_state_filter, sys_setpriority, sys_ioprio_set and sys_opprio_get, offsets of prev and next fields from copy_process function and pid field offset from functions copy_process, crash_save_cpu, sigqueue_alloc and kexec_should_crash. Column 1 of table 8.1 shows versions of our test kernels and columns 2,3,4 and 5 show respectively for each test kernel the values of init_task pointer in addition to next, prev and pid offset in task_struct. These values were reported by a kernel module that we have written and executed inside each test kernel to obtain a ground truth for our evaluation.

As we can see in table 8.1, global kernel pointers and field offsets can be very different between kernels with different customizations even if they are of a same version (e.g. Debian 7 and Ubuntu 12). On the other hand, global kernel pointers and field offsets can be similar – by ignoring the effect of randomization on kernel pointers – on kernels with close versions and customizations (e.g. from 3.14.1 to 3.14.5, 4.1.5 - 4.1.8 and 4.2.1 to 4.2.5). We precise that even if these values are stable or do not change frequently across close kernel versions and customizations, it is not possible to know in advance for a given kernel if its global kernel pointers and field offsets of interest are similar to the ones of a close kernel or not. Hence the advantage of an automatic and kernel portable approach like ours over existing manual approaches.

As reported in column 6, our approach identifies precisely with no false positives or negatives init_task global pointer and field offsets of interest in task_struct on all test kernels regardless of their compilation options through analysis of few long lived and stable Linux kernel functions.

These preliminary results shows that our approach is very promising for automatic and portable identification of global kernel pointers and field offsets of interest regardless of kernel compilation options. We intent after fully automating our approach to extensively evaluate it on more global kernel pointers and field offsets to confirm its efficiency and reliability.

8.2 Hidden running process detection

8.2.1 Introduction

Long lived malware employ generally a process hiding attack to conceal their presence from security and administration tools. The idea of process hiding attack is to leave process information accessible to kernel scheduler but remove them from kernel data sources from which security and administration tools obtain process list. This way, the concerned process is not reported in process list by security and administration tools yet continues to run to normally.

Several existing works based on VMI addressed the problem of hidden running process detection through a cross view validation approach that consists in comparing a trusted and untrusted views of kernel process list which reveals the presence of a hidden running process in case there is difference between the two views. However, most of these existing works are not suitable for usage in real world IaaS cloud environments as they generally use too kernel-specific information that are manually provided by a security

expert to construct the trusted and untrusted views of kernel process list.

In this part of the current chapter, we introduce a new Hidden Process Detector (HPD) that is based on a cross view validation approach in which the trusted and untrusted views of kernel process list are constructed automatically through instrumentation of long lived and stable kernel functions.

We discuss hereafter existing approaches for hidden running process detection and their limitations, then we detail the design of HPD and finally we present its evaluation.

8.2.2 Related work

Existing VMI systems for hidden process detection share the idea of cross view validation but differ in how they construct the trusted and untrusted views. Those based on data traversal 3.2.2 such as LiveWire [70] and VMwatcher [78] construct the trusted view from task list whose address and field offsets of interest in its elements are obtained manually and the untrusted view from *ps* list obtained through a shell connection to the VM. These approaches consider the kernel task list as a trusted view despite that it can be subverted through a Direct Kernel Object Manipulation (DKOM) attack [5], hence they can not detect processes that are hidden kernel task list but present in the scheduler list. In addition, they are not convenient for IaaS cloud environments as they require access to the introspected VM through a shell connection.

Lycosid [81] 3.2.3 constructs its trusted view from process switching events detected through trapping writes to CR3 register. Lycosid untrusted view is constructed from *ps* list similarly to LiveWire and VMwatcher. Despite that Lycosid approach is more reliable than the presented ones as its does not trust kernel task list, it is not convenient for IaaS cloud environments as it requires also access to the introspected VM through a shell connection.

VMST [66] 3.2.4 constructs automatically its trusted view from scheduler list pointed to by the global variable runqueue and the untrusted view from kernel task list pointed to by init_task. While VMST approach based on data access redirection is reliable and fully automatic, it is designed on top of Qemu emulator and based on a heavy and complex taint analysis technique. VMST is not practical as its design is not intended for hardware-assisted virtualization which operates most today cloud infrastructures.

LiveDM [109] constructs its trusted view of running processes by tracking process creations and terminations through interception of kernel functions that allocate and free process descriptors (addresses of these functions are obtained from kernel symbols file). Regarding the untrusted view, LiveDM constructs it manually from kernel task list pointed to by init_task. LiveDM enhances automation and kernel portability of trusted view construction through the interception of long lived kernel functions. However, it still non practical for real cloud environment as its untrusted view is constructed using manually provided address of init_task and offsets needed to locate and walk through kernel task list.

Autotap [132] is a system based on dynamic binary code analysis that aims to identify automatically kernel instructions that manipulate kernel data structure of interest. As a security use case, Autotap authors developed a hidden process detector in which they identify addresses of scheduled process descriptors from operands of instructions that use them during kernel scheduler execution (i.e. trusted view). Autotap detects hidden processes by comparing information of identified process descriptor with the ones returned by ps command through a shell connection (i.e. untrusted view). Autotap approach is complex and heavy due to the nature of dynamic binary code analysis. Hence, it is not convenient for usage in real world cloud

environments.

HPD approach surpasses existing systems as the trusted and untrusted views of process list are constructed in an automatic and widely kernel portable way thanks to the instrumentation of long lived kernel functions. We detail hereafter the design of HPD for hidden process detection.

8.2.3 HPD design

Process hiding attack can be done in the Linux kernel on *ps* list obtained from */proc/* file through by modifying *ps* binary file or hooking some file-related kernel functions [130]. This attack can be also performed by removing a process descriptor from the kernel task list through a DKOM attack [5]. In this presented use case, we assume that information of the hidden process are removed from both *ps* list and kernel task list. We believe this scenario is the most representative of today stealthy malware. We assume also in this use case code integrity of the instrumented kernel functions

HPD constructs its trusted view on running processes through the interception of a scheduling function entry point (e.g. identified by NoGap) that manipulates addresses of running process descriptors. Among multiple scheduling-related functions in the Linux kernel that are relevant for constructing a trusted view, we choose in HPD to intercept try_to_wake_up function to enhance HPD portability across Linux kernels. According to Linux Cross Reference web site [17], try_to_wake_up function is long lived as it is available from oldest Linux kernel versions (2.6.11) until the most recent ones (4.11-rc). HPD obtains descriptor address of the to-be scheduled process during interception of try_to_wake_up function by accessing to its parameter 'process descriptor'. Retrieved process descriptor addresses are the trusted view of HPD on running processes.

To obtain the untrusted view on running processes, HPD locates then walks through kernel task list pointed to by init_task global pointer. To do so, HPD identifies init_task global pointer in addition to next and prev field offset in task_struct data structure through our function instructions analysis mechanism presented in the first part of this chapter 8.1.

To detect the existing of a hidden running process, HPD checks if each process manipulated by try_to_wake_up belongs the kernel task list or not. In case HPD detects scheduling of a process that does not appear in kernel process list, it concludes that this process is a hidden running process and its execution should be stopped.

To stop execution of a detected hidden running process, HPD injects a call to sys_kill kernel function with the pid of the hidden process as a parameter. However, killing a running process that does not belong to the task list crashes the Linux kernel. To avoid this, HPD reinserts first descriptor of the hidden process in kernel task – linked – list. Doing so – from the hypervisor level – is straightforward for HPD thanks to the automatic identification of next and prev field offsets in task_struct data structure. At this point, HPD calls VM sys_kill function to safely terminate execution of the detected hidden running process.

8.2.4 Evaluation

We present hereafter evaluation results of HPD prototype implemented on KVM hypervisor as an extension of NoGap for hidden process detection on a Debian 7 and its performance overhead.

Hidden process detection

To evaluate HPD capacity for detecting hidden processes whose information are removed from both *ps* and kernel task lists, we hide a running Firefox process from these lists using adore-ng rootkit (removes the process from /proc/ files) and another rootkit we developed (removes the process from kernel task list). The moment we launch HPD, it detects the hidden Firefox process and outputs information of the its descriptor (e.g. address and pid). The hidden process is then safely terminated through a call injection to sys_kill function and the VM continues to run normally.

Performance overhead

Benchmark Program	KVM	KVM + HPD	#Interceptions/s of scheduling function	Overhead
Apache (request/s)	3042,126	1625,084	10200	46,58%
Kernel downloading (s)	29,734	37,344	3000	19,94%
Kernel decompression (s)	16,628	17,905	340	7,13%

Table 8.2 – HPD performance overhead evaluation

HPD execution causes inevitably a performance loss to the introspected VM due to the interception of scheduling attempts for hidden process detection. We evaluate hereafter this performance loss on the following three test programs: Apache, kernel 3.13.1 source file downloading (wget) and kernel 3.13.1 source file decompression (bunzip2), by measuring their execution time on KVM with and without HPD. For KVM with HPD, we also measured for each test program the approximate number of try_to_wake_up function interceptions per second. Table 8.2 reports the different measurements for each test program.

To measure Apache performance, we sent from a client machine using Apache Bench [2] 10,000 requests with a concurrency level of 32 to the VM introspected by HPD. As we can see in table 8.2, the performance loss of Apache is high (46,58%) because Apache is an IO intensive program that uses a child process to handle each received connection [117]. This behaviour of Apache triggers frequent process scheduling events and hence frequent interceptions of try_to_wake_up functions (10200 interceptions/s). Hence the significant performance loss.

In the second test, we measured with Linux time command the execution time of wget command for downloading Linux kernel 3.13.1 source file (111 MB). For this test, we notice that the performance overhead of HPD is moderate (19,94%). Despite that wget is an IO intensive program, it triggers much less interceptions of try_to_wake_up functions than Apache (3000 interceptions/s). Hence the less important performance loss.

In the last test, we measured the execution time of bunzip2 command for decompressing the downloaded kernel source file. We observe that for this test, the performance overhead of HPD is low (7,13%) because bunzip2 is computation intensive program which does not actively triggers a significant number of process scheduling events (340 interceptions/s).

Hence we conclude that the more a program is concurrent and IO intensive (resp. computation intensive) the more (resp. the less) is the performance overhead induced by HPD.

8.2.5 Limitations and future work

The current prototype of HPD rely on hidden process pid to stop the execution of this latter one. However, a successful process hiding attack means that the attacker is capable of modifying kernel data. Hence, hidden process pid should not be trusted because it can be forged by an attacker to expose a pid of a legitimate process. To address this limitation, we will investigate in a future work terminating hidden process execution using kernel functions that use only on process descriptor address (e.g. do_send_sig_info in the Linux kernel).

In the current prototype of HPD, we addressed the case of processes that are hidden from both kernel task and ps lists and ignored the case where the hidden process is removed only from ps list which we believe non representative of today stealthy malware. Handling this latter case is just a matter of comparing our trusted view with a second untrusted view constructed by reading /proc/ file from the hypervisor through function call injection.

Finally, to reduce VM performance loss caused by HPD, we will investigate in a future work how to obtain a trusted view of running processes with a reduced number of kernel function interceptions.

8.3 Conclusion

We presented in this chapter a security use case that illustrates bridging the semantic gap using the different mechanisms for instrumenting kernel functions identified with the help of our systems NoGap or K-binID. In this use case of detecting running processes that are hidden from Linux task list and ps command output, we described a preliminary work for identifying automatically global kernel pointers and data structures field offsets of interest for security applications developed on top of NoGap or K-binID.

We showed in this chapter how our systems (i.e. NoGap or K-binID) can be used for designing easily automatic and largely kernel portable VMI applications that detect and safely react against malicious activities thanks to the instrumentation of kernel functions.

These properties demonstrate how techniques presented in this thesis are convenient for usage in real world IaaS cloud environments.

9

Conclusion

We present in this chapter a summary of works described in this thesis and their results. We outline after that our future work in addition to perspectives that would become possible thanks to the potential of our results.

9.1 Summary

In this thesis we presented how the increasing adoption of cloud environments operated with virtualization technology opened the way to a promising hypervisor-based security monitoring approach named Virtual Machine Introspection (VMI). Despite that considerable research efforts were done on VMI since its introduction in 2003, the proposed systems were not convenient for usage in real world IaaS cloud environments as they lacked either efficiency (i.e. bridge the semantic gap in a very limited way), automation or guest OS independence. For this reason, the goal of this thesis was to address the problem of bridging the semantic gap for VMI in a way that is convenient for real world IaaS cloud environments.

For this aim, we studied and classified state of the art VMI approaches for bridging the semantic gap in addition to their applications to understand their properties and limitations. We observed in this study that VMI approaches that employ binary code reuse and analysis mechanisms such as interception and call injection on a part of VM kernel binary code, succeed to enhance either their efficiency (i.e. significantly bridge the semantic gap), automation or independence from the guest OS. However, neither of these existing systems enhanced at once the desired properties for IaaS cloud environments.

To address this problem, we investigated in this thesis the application of binary code reuse and analysis mechanisms on all VM kernel binary code, namely all kernel functions, to widely narrow the semantic gap in an automatic and largely OS independent way. For this end, the hypervisor should first be aware of the location of VM kernel binary code which is unknown to the hypervisor due to the semantic gap and variable because of kernel base address randomization mechanism. To solve this challenge, we presented in

this thesis a novel hypervisor-based main kernel binary code disassembler which enables the hypervisor to locate all VM main kernel binary code and divide it into code blocks given only the address of one arbitrary kernel instruction. In this mechanism, we introduced a backward disassembly technique that leverages disassembly self-repairing phenomena to enable a binary analysis system to discover correctly boundaries of instructions that precede a given kernel instruction.

To instrument (i.e. intercept, call and analyze) kernel functions in VM main kernel binary code located and disassembled with our just mentioned mechanism, the hypervisor should first obtain their addresses in the binary code. There is two ways for the hypervisor to obtain automatically addresses of kernel functions for introspection.

The first one is to identify precisely which kernel binary code the introspected VM is running in order to determine appropriate kernel symbols file from which function addresses of that kernel can be obtained. Existing approaches for kernel binary identification are not suitable for usage in IaaS cloud environments as they are either imprecise due to the analysis of a limited part of kernel binary code or usable only on kernels compiled with specific options that facilitate locating sufficient kernel binary code for precise identification. To address limitations of these existing techniques, we proposed in this thesis K-binID, a kernel binary code identification system built on top of our hypervisor-based main kernel disassembler. K-binID approach defines signatures of kernel binary code at the level of code blocks and considers the set of hash signatures of all kernel code blocks as the fingerprint of a main kernel binary code. K-binID approach identifies precisely VM main kernel binary code with no false positives or negatives regardless of kernel compilation options and base address randomization. This enables hence the hypervisor to determine appropriate kernel symbols file from which it can obtain function addresses for VMI.

The main limitation of K-binID is that there is a huge number of unique kernel binary code versions and customizations and hence a huge number of unique and kernel-specific symbols file. So, K-binID is helpless on kernels whose symbols file is unavailable even if their binary code is very similar to a kernel whose symbols file is available for K-binID. To address this limitation of K-binID, we investigated a second way to obtain kernel function addresses which consists in identifying kernel function addresses and their names directly in the memory of a running VM without necessarily having symbols file specific to the VM kernel. We developed NoGap for this end, a system built on our main kernel disassembler which deduces most kernel function addresses and names from hash signatures and control flow information of code blocks that compose these functions. Thanks to the similarity of function code blocks signatures across different kernel versions and customizations, NoGap enables the hypervisor to identify most functions whose code block signatures are known even on kernels whose symbols file is unavailable. This property of NoGap makes it even more suitable than K-binID for usage in IaaS cloud environments.

NoGap enables the hypervisor to intercept, call and analyze identified kernel functions to significantly narrow the semantic gap and obtain automatically highly fine grained and kernel specific information for VMI applications. We demonstrated at the end in thesis through a security use case how kernel functions instrumentation enabled by NoGap can be used to easily and automatically detect and stop execution of processes that are hidden from Linux kernel task list.

9.2 Future work and perspectives

9.2.1 Evaluation and support of other OSes than Linux

In this thesis we presented several techniques based on kernel binary code reuse and analysis in the goal of applying them to all VM kernel binary code to bridge the semantic gap in a convenient way for IaaS cloud environments. Despite that these techniques were designed with OS independence in mind by avoiding usage of any hard coded or OS-specific information, we evaluated them only on Linux kernels because of the availability of their source code and detailed information about their internals which makes possible an extensive evaluation of our techniques. Hence as a future work, we intent to evaluate our techniques on other open source OSes such as FreeBSD [24] and explore how they can be reliably evaluated on closed source but relatively documented OSes such as Windows.

9.2.2 Compiler-agnostic function localization and labelling

We designed in this thesis NoGap, a system that enables to obtain addresses of kernel functions even on some unknown kernels and narrow the semantic gap through the instrumentation of these identified kernel function. To identify kernel functions, NoGap uses hash signatures that are sensitive to the slightest modification in kernel functions code. Knowing that kernel functions code can change with kernel versions and customizations in addition to compiler version, NoGap user must hence continuously update NoGap signatures database for these different parameters in order to maintain NoGap efficiency. To solve this limitation, we observed that even when kernel functions code changes, their semantics and several parameters such as function prototype, call relationships, used global kernel pointers and data fields are generally preserved. Based on these observations, we intent in a future work to explore locating kernel functions and labelling them based only on the mentioned parameters that are compiler agnostic and tolerant to modifications in kernel functions that do not change significantly their semantics.

9.2.3 Fully safe function call injection

We showed in this thesis through the proposed security use case how interception and analysis of kernel functions enables the hypervisor to obtain highly fine-grained and OS specific information such as addresses of scheduled process descriptors and kernel task list location. We also showed in this use case how the hypervisor can safely trigger elementary operations in the kernel such as killing a running process through function call injection mechanism. However, as we detailed in this thesis, NoGap in addition to existing VMI systems support only asynchronous call injection to functions that leave the kernel in a consistent state. This means that NoGap does not enable the hypervisor to call a kernel function at any moment and some advanced kernel operations (e.g. process scheduling) are not yet possible through the current function call injection implemented in NoGap. We intent also in a future work to address this limitation to enable any time and safe call injection to most kernel functions in order to expand the range of possible applications with NoGap.

9.2.4 Conditional function execution interception

We saw in evaluation of the proposed use case how NoGap may lead to a significant VM performance loss as it intercepts – unconditionally – all executions of a monitored kernel function even the ones that may not be of interest. One way to reduce VM performance loss is to intercept only kernel function executions that are of interest to NoGap applications and let the other executions run with no interruption. To do so, we intent to explore in a future work how to intercept kernel functions through breakpoints that are executed only when application-specified conditions are verified.

9.2.5 Global kernel pointers and data structure field offsets identification

At the end of this thesis, we described a preliminary work for identifying automatically global kernel pointers and field offsets of interest through the analysis of some kernel functions that use these information. We showed that the proposed techniques work well in practice for needed information in the proposed hidden process detection use case. We intent to continue this work to automate its manual part and then extensively evaluate it for other global kernel pointers and field offsets to precisely identify its efficiency and limitations.

9.2.6 Support of untrusted kernels

All mechanisms presented in this thesis were designed with the assumption that the VM kernel is not malicious and hence would not try to evade and tamper with our kernel code analysis and instrumentation mechanisms. This assumption can be unverified in real world and hence used to attack our systems and compromise the hypervisor. Hence, we intent to explore in a future work how to preserve efficiency of our techniques on compromised and uncooperative kernels.

9.2.7 Explore VMI applications based on our techniques

Our techniques presented in this thesis makes available at the hypervisor level a rich and highly fine-grained semantic view on introspected VM states and activities thanks to the instrumentation of thousands of identified kernel functions. We believe that this opens the way for an easy development of novel automatic and kernel portable VMI applications that are difficult or even impossible to design with existing VMI techniques. Indeed, the complex part of obtaining needed semantic information in a VMI application is largely simplified just by applying the suitable instrumentation mechanism on the relevant kernel functions. So the effort left to the user of our techniques is to determine the suitable kernel functions and how to instrument them in addition to the logic of the VMI application that uses the obtained semantic information.

We also believe that our techniques can be used to explore a wide range of VMI application types as kernel functions cover all operating system aspects (e.g. memory, processes, files, network, etc.). In particular, the rich semantic view on VM states and activities enabled by our techniques is very useful for checking security policies violations defined by intrusion detection and prevention applications. Also, the possibility of intercepting kernel functions (system calls and internal functions) is also very interesting for both user mode and kernel mode malware analysis. Finally, today hypervisors are not aware of internal

states and activities of the hosted VMs due to the semantic gap. So, these information are not taken into account in resource allocation policies. Moreover, multiple resource management tasks such as memory management and execution scheduling are done both at the level of the hypervisor and the VM kernel but with no collaboration between them. Hence, by making the hypervisor semantically aware of VM states and activities and able to trigger execution of kernel functions thanks to our techniques, we believe that there is an opportunity for a global optimization of VM resource allocation in the cloud. Chapter Template

10

Résumé étendue en français

10.1 Contexte

Le développement rapide d'un matériel informatique (stockage, réseau et calcule) performant et pas cher a conduit à la création d'une nouvelle façon d'utilisation des ressources informatiques appelée le cloud computing ou l'informatique en nuage [95]. Le cloud computing consiste à utiliser des ressources informatiques qui sont accessibles à distance via internet et qui sont généralement partagées par des utilisateurs différents. Le partage des ressources informatique dans le cloud se fait grâce à une couche logicielle de virtualisation qui permet de diviser les ressources physiques (processeur, mémoire, disque, etc.) en ressources virtuelles (processeur virtuel, mémoire virtuelle, disque virtuel, etc.) [76].

Chacun des utilisateurs du cloud se fait attribuer une partie des ressources virtuelles sous la forme d'une machine virtuelle (VM) sur laquelle il peut exécuter un système d'exploitation (OS) et des applications exactement comme sur une machine physique. Grâce à la technologie de virtualisation matérielle, les utilisateurs du cloud peuvent exécuter simultanément leurs OSes et applications dans des machines virtuelles sur une même machine physique. La couche logicielle de virtualisation (appelée hyperviseur) s'interpose entre les VMs et les ressources physiques et gère l'exécution des VMs et l'accès à leurs ressources de telle sorte que les ressources allouées à chaque VM soient isolées de ceux des autres VMs comme si elles étaient des machines physiques séparées.

Aujourd'hui de plus en plus d'entreprises déploient et fournissent leurs services à travers des machines virtuelles dans le cloud à cause de la performance, flexibilité et isolation des ces dernières. Cette adoption massive des environnements cloud les rends une cible privilégiés des attaques informatiques, d'où l'importance de la supervision de la sécurité des VMs. La supervision de la sécurité dans le cloud est essentielle car malgré que la sécurité inter-VMs (isolation des ressources) est assurée par l'hyperviseur, la sécurité intra-VM (de l'OS et des applications) ne l'est pas et les vulnérabilités existantes dans l'OS et les applications restent présentes et exploitables aussi dans les machines virtuelles.

10.2 Supervision de la sécurité dans le cloud

Dans les environnements virtualisés, trois approches de supervision de la sécurité intra-VM sont possibles, dont deux traditionnelles [70]. Tout d'abord l'approche appelée "host-based intrusion detection" qui consiste à exécuter un logiciel de sécurité (ex: antivirus) dans la VM pour superviser l'état et les activités de cette dernière. Le fait que le logiciel de sécurité soit à l'intérieur de la VM lui permet d'avoir une vue complète et riche des différentes activités de la VM supervisée. Par contre, comme le logiciel de sécurité s'exécute sur le même environnement qu'il protège, il n'est donc pas isolé des activités de la VM et il peut lui même être attaqué. L'approche "host-based" permet donc une bonne visibilité des activités de la VM mais une mauvaise isolation de ces dernières.

La deuxième approche de la supervision de la sécurité dans un environnement virtualisé est appelée "network-based". Elle opère au niveau réseau (ex: pare-feu) et consiste à inspecter les activités réseau des VMs pour y détecter des signes ou des comportements représentatifs des attaques et des malwares connus. Le fait d'opérer au niveau réseau permet à cette approche d'avoir une bonne isolation des attaques car l'entité de supervision de sécurité est à l'extérieur des VMs. La principal limitation de cette approche est que l'entité de supervision de sécurité n'a de visibilité que sur le trafic réseau des VMs et donc ne peut pas détecter les malwares et les attaques qui viennent d'une autre source (ex: disque USB). Une deuxième limitation de cette approche est que son efficacité devienne limitée quand le trafic réseau est chiffré car son contenu est incompréhensible pour l'entité de supervision de sécurité.

Grâce à la virtualisation, une troisième approche de la supervision de la sécurité dans le cloud est devenue possible. Elle consiste à superviser les états et activités des VMs depuis l'hyperviseur. Cette approche est appelée dans la littérature "introspection de machine virtuelle" (VMI) et a été introduit en 2003 par Garfinkal et Rosenblum [70]. L'introspection de machine virtuelle combine d'une part la bonne visibilité des approches "host-based" car l'hyperviseur peut inspecter toutes les activités des VMs sur leurs ressources virtuelles (ex: CPU, mémoire, réseau, disque, etc.) grâce à son interposition entre les VMs et les ressources physiques, et d'une part l'isolation des approches "network-based" car l'hyperviseur se trouve à l'extérieur des VMs et sa sécurité est renforcée par des mécanismes processeur d'assistance à la virtualisation. En plus, un système d'introspection peut superviser simultanément plusieurs VMs grâce à l'emplacement de l'hyperviseur.

Ces propriétés rendent l'introspection de machine virtuelle une approche très prometteuse pour la supervision de la sécurité dans le cloud. Cependant, les états et les activités des VMs sont vus par l'hyperviseur comme des zéros et des uns dans la mémoire des VMs en plus des états de leurs ressources virtuelles. Cela constitue un challenge pour l'introspection car pour superviser la sécurité des VMs, l'entité de supervision de sécurité au niveau hyperviseur a besoin d'une vue sémantique (c-a-d compréhensible) sur les processus, les fichiers, les appels systèmes, etc. qui n'est pas reflétée dans la vue dont l'hyperviseur se dispose. Cet écart entre la vue brute des états et des activités des VMs dont l'hyperviseur se dispose et la vue sémantique nécessaire pour la supervision de sécurité depuis l'hyperviseur s'appelle "le gap sémantique" [50].

10.3 État de l'art d'introspection de machine virtuelle

Pour superviser la sécurité des VMs depuis l'hyperviseur, un système d'introspection doit d'abord franchir le gap sémantique, c-a-d obtenir des informations sémantiques sur les états et les activités des VMs à partir de la vue brute disponible au niveau hyperviseur. Dans l'état de l'art d'introspection de machine virtuelle, une variété d'approches a été proposée pour franchir le gap sémantique et explorer des applications innovantes, notamment de sécurité, basées sur l'introspection de machine virtuelle. Ces approches peuvent être classifiées en quatre catégories selon la source et la façon avec laquelle les informations sémantiques sont obtenues.

La première catégorie d'approches d'introspection s'appelle "in-VM" [101][118], elle consiste à déployer à l'avance ou à la volé un agent à l'intérieur de la VM dont le rôle est d'exposer la sémantique des états et des activités des VMs à l'hyperviseur. De son côté, l'hyperviseur se charge de protéger l'intégrité de l'agent in-VM et de vérifier que les états et les activités des VMs exposées par ce dernier respectent les politiques de sécurité définies.

Cette catégorie d'approches d'introspection évite le problème du gap sémantique grâce à l'utilisation de l'agent in-VM. En revanche, l'utilisation d'un agent in-VM rend la conception des approches in-VM dépendante de l'OS de la VM et vulnérable face aux attaques à l'intérieur de la VM comme pour les approches "host-based" à cause de la mauvaise isolation de l'agent in-VM. Pour ces raisons, les approches in-VM n'ont fait l'objet que de peu de travaux dans l'état de l'art d'introspection de machine virtuelle et les approches "out-of-VM" ont été plus considérées.

La deuxième catégorie d'approches d'introspection s'appelle "out-of-VM delivered" [78][100] et elle couvre principalement les premières approches d'introspection proposées dans l'état de l'art. Cette catégorie d'approches opère totalement depuis l'hyperviseur et franchit le gap sémantique grâce à des informations sémantiques sur les détails internes de l'OS de la VM. Ces informations sémantiques sont délivrées manuellement par un administrateur de sécurité ou extraites à partir du code source, d'une documentation technique ou des informations de débogage de l'OS.

Cependant, ces informations sémantiques nécessaires pour franchir le gap sémantique sont très spécifiques au type et à la version de l'OS de la VM et généralement elles nécessitent un effort manuel et humain pour les obtenir. En conséquence, les approches "out-of-VM delivered" ne sont pas adaptées aux infrastructures cloud qui hébergent un grand nombre de types et de versions d'OSes différents car il faut faire l'effort de fournir les informations sémantiques spécifique pour chaque type et version d'OS et après chaque mise à jour de l'OS. Par ailleurs, l'utilisation des informations sémantiques statiques et supposées invariantes pour un OS donné, rend les approches "out-of-VM delivered" vulnérables aux attaques sur les données de l'OS qui rendent les informations sémantiques délivrées obsolètes. Ces attaques faussent donc la vue sémantique obtenue au niveau hyperviseur.

Pour répondre aux faiblesses des approches "out-of-VM delivered", une troisième catégorie d'approches d'introspection appelée "out-of-VM derived" exploite exclusivement l'architecture matérielle et ses fonctionnalités d'assistance aux OSes et hyperviseurs pour dériver (c-a-d déduire) des informations sémantiques sur les états et les activités des VMs [80][105]. Ces approches "out-of-VM derived" déduisent des informations sémantiques en inspectant les évènements observées au niveau processeur ou en utilisant des

mécanismes matériel pour arrêter l'exécution d'une VM lorsque elle effectue une activité assistée par le processeur. Cette façon d'opérer permet aux approches "out-of-VM derived" d'être indépendantes de l'OS de la VM et résistantes aux attaques sur les données de ce dernier.

Malgré ces avantages, la visibilité sémantique que puissent avoir les approches "out-of-VM derived" est limitée uniquement aux activités qui sont assistées par le processeur. Donc le champ d'applications possibles de ces approches est aussi très limité. En plus, ces approches peuvent induire une perte de performance importante aux VMs à cause de l'interception fréquente des activités assistées par le processeur.

La dernière catégorie d'approches d'introspection comprend les approches dites "hybrid" [69], qui utilisent une combinaison de techniques des autres catégories pour améliorer la visibilité sémantique d'un système d'introspection et étendre le champ d'applications possibles. Pour les propriétés d'une approche d'introspection de cette catégorie, elles dépendent fortement de la combinaison de techniques utilisée.

10.4 Problématique de recherche

Malgré le nombre et la variété des approches d'introspection existantes pour franchir le gap sémantique, aucune d'entre elle est à la fois automatique, indépendante de l'OS de la VM, sécurisée et franchit complètement le gap sémantique sans poser des problèmes de performance à la VM. Toutes ces propriétés sont essentielles pour un système d'introspection conçu pour être utilisé par un opérateur de cloud qui propose son infrastructure comme un service (IaaS), et qui est engagé à garantir une qualité de service optimale tout en hébergeant des milliers de machines virtuelles hétérogènes dans lesquelles les utilisateurs installent le type et la version d'OS de leur choix.

C'est pourquoi l'objectif principal de cette thèse est de proposer une nouvelle approche d'introspection qui est adaptée aux environnements cloud de type IaaS qui s'exécutent sur l'architecture x86. Une fois une telle approche est conçu, un deuxième objectif de cette thèse et de montrer à travers une application de sécurité comment l'approche proposée permet de franchir le gap sémantique puis détecter une attaque et réagir face à cette dernière avec des propriétés meilleures de celle de l'état de l'art.

10.5 Contributions de la thèse

Les principales contributions de cette thèse dans le but d'atteindre ces objectifs sont les suivantes:

10.5.1 Étude technique des approches d'introspection majeures et de leurs propriétés

Dans cette thèse nous étudions techniquement et de façon détaillée les systèmes d'introspection existants et nous classifions leurs approches pour franchir le gap sémantique. Pour chacun des systèmes considérés, nous présentons techniquement comment il procède pour franchir le gap sémantique pour une application spécifique, puis nous détaillions les avantages et les inconvénients de son approche.

À la fin de cette présentation technique, nous comparons les propriétés des systèmes d'introspection considérés. Suite à cette comparaison, nous concluons que lorsque les techniques d'analyse et d'instrumentation

de code binaire sont intégrées dans un système d'introspection et appliquée sur une partie du code noyau de l'OS (ex: appels systèmes) de la VM, elles améliorent considérablement une ou plusieurs propriétés du système d'introspection concerné (automaticité, indépendance de l'OS, efficacité du franchissement du gap sémantique, etc.) [66][69].

Grâce à cette conclusion, nous explorons dans cette thèse l'idée d'appliquer les techniques d'analyse et d'instrumentation de code binaire sur tout le code binaire du noyau de l'OS de la VM (c.-à-d. sur toutes les fonctions noyau) afin d'obtenir une sémantique très fine et riche des état et des activités des VMs et améliorer à la fois toutes les propriétés recherchées dans un système d'introspection destiné à être utilisé dans un cloud de type IaaS.

10.5.2 Désassemblage de code noyau basé sur l'introspection

Notre idée d'intégrer dans un système d'introspection les techniques d'analyse et d'instrumentation de code binaire et de les appliquer sur tout le code noyau de la VM nécessite d'abord que l'hyperviseur connaisse l'emplacement et la taille du code noyau de la VM. Ces informations sont indispensables pour désassembler et interpréter correctement le code noyau de la VM à des fins d'introspection. Cependant, l'emplacement et la taille du code noyau de la VM sont des détails très spécifiques à l'OS de la VM et totalement inconnus pour l'hyperviseur car ils changent selon le type, la version et les options de compilation de l'OS de la VM. En plus, l'emplacement du code noyau de la VM peut changer de manière aléatoire après chaque redémarrage de la VM à cause des mécanismes de randomisation (ex: ASLR [15]). Tout cela constitue un challenge pour notre idée qui nécessite de localiser et désassembler correctement le code noyau de la VM.

Dans cette thèse nous proposons un mécanisme d'introspection capable de localiser tout le code noyau de la VM, fragment par fragment (appelé bloc de code), à partir d'une seule instruction noyau identifiée automatiquement depuis l'hyperviseur dans la mémoire de la VM. Ce mécanisme procède d'abord par la localisation d'une instruction qui appartient sûrement au code noyau de la VM grâce aux fonctionnalités du processeur telles que le registre qui pointe vers la première instruction du gestionnaire des appels systèmes dans le noyau de la VM. Nous appelons cette instruction I0. Une fois une telle instruction est localisée, notre mécanisme découvre le code noyau qui succède I0 en désassemblant bloc par bloc le code binaire à partir de l'adresse de cette dernière jusqu'à ce qu'il rencontre un signe de fin du code (ex: une instruction invalide).

Contrairement à cette deuxième étape, découvrir et interpréter correctement le code noyau qui précède I0 est beaucoup plus difficile car l'octet qui la précède désigne uniquement la fin d'une instruction précédente et ne donne aucune indication sur l'adresse du début de l'instruction précédente car les instructions de l'architecture x86 sont de longueurs variables. Pour résoudre cette difficulté, nous avons proposé dans cette thèse un mécanisme de désassemblage en marche arrière qui permets de localiser correctement les débuts d'instructions qui précèdent une adresse donnée. L'idée clé derrière ce mécanisme de désassemblage en marche arrière est que dans une séquence d'instructions obtenue lorsqu'un code binaire est désassemblé à partir d'une position incorrecte (c.-à-d. ne désigne pas un début d'instruction), les instructions correctes réapparaissent généralement juste après quelques-unes incorrectes. Ce phénomène s'appelle l'autoréparation du désassemblage [90].

Grâce à ce phénomène, l'idée générale de notre mécanisme de désassemblage en marche arrière est de désassembler le code binaire à partir de plusieurs positions différentes qui précèdent une adresse donnée, c'est qui donne plusieurs séquences qui contiennent chacune des d'instructions correctes grâce au phénomène d'autoréparation. Ensuite, notre mécanisme considère sous certaines conditions que les instructions qui apparaissent aux mêmes adresses dans les séquences obtenues, sont correctes (c.-à-d. ses instructions ont été désassemblées à partir de leurs adresses de débuts correctes). Cette approche permet donc découvrir et interpréter correctement une séqeunce d'instructions qui se trouve avant une adresse donnée. Notre mécanisme itère ces étapes pour découvrir et interpréter correctement bloc par bloc le code noyau de la VM jusqu'à ce qu'il rencontre un signe de fin de code.

L'ensemble des étapes présentées permet donc à l'hyperviseur de localiser automatiquement le code noyau de la VM à partir de I0.

10.5.3 Identification précise de code binaire du noyau d'une VM

Afin d'appliquer les techniques d'analyse et d'instrumentation binaire sur tout le code noyau de la VM, c.-à-d. sur toutes les fonctions noyau, l'hyperviseur doit d'abord connaitre les adresses de ces dernières. Un moyen possible pour faire cela est d'identifier quel code binaire s'exécute dans le noyau de la VM puis d'extraire les adresses des fonctions noyau à partir du fichier de symboles associé au noyau identifié. L'identification de code binaire consiste à trouver la version et les options de compilation (customisation) du noyau pour savoir à partir de quel fichier de symboles l'hyperviseur peut obtenir les adresses des fonctions noyau de la VM.

Un fichier de symboles est unique à une version et customisation spécifiques d'un code noyau. Donc même si deux noyaux sont très similaires en termes de version ou de customisation, leurs fichiers de symboles et donc leurs adresses de fonctions peuvent être très différentes. Cela implique que le système d'identification de code binaire du noyau doit être très précis afin d'éviter la confusion des noyaux proches et donc l'utilisation d'un fichier de symboles inapproprié pendant l'introspection ce qui peut cracher la VM et l'hyperviseur. Les techniques existantes d'identification de code binaire du noyau ont une précision limitée car soit ils déduisent l'identité du noyau en se basant uniquement sur une partie de son code binaire [54], soit ils exigent que le code noyau soit compilé de façon spécifique afin de pouvoir localiser une grande partie du code binaire du noyau [71]. Les techniques existantes d'identification de code binaire du noyau ne sont pas donc adaptées à une utilisation dans un cloud IaaS où les noyaux des VMs peuvent être de n'importe quelle version et customisation.

Dans cette thèse nous proposons K-binID, un nouveau système d'identification de code binaire du noyau basée sur notre mécanisme de désassemblage de code noyau et qui permet d'identifier précisément depuis l'hyperviseur le code binaire du noyau de la VM (parmi un ensemble connu) indépendamment de sa version et de sa customisation. La principale contribution de K-binID est qu'il définit des signatures (hashes) au niveau des blocs de code du noyau pour permettre une identification précise du code binaire de ce dernier malgré les difficultés posées par certaines options de compilation et le mécanisme de randomisation ASLR.

K-binID procède en deux phases: une phase hors ligne d'apprentissage et une autre phase en ligne d'identification de code binaire du noyau. Dans la phase hors ligne, K-binID génère pour chaque noyau

connu à l'aide du fichier de symboles de ce dernier l'ensemble de signatures de ces blocs de code noyau puis produit une base de signatures multiOS. Chaque entrée de cette base contient une signature à laquelle associé une liste d'identifiants des noyaux connus qui comportent un bloc de code avec une telle signature.

Dans la deuxième phase, le noyau de la VM n'est pas connu et l'hyperviseur cherche à l'identifier s'il appartient à l'ensemble des noyaux connus grâce à la base de signatures générée en amont dans la première phase. Pour cela, K-binID localise l'ensemble des blocs de code noyau grâce à notre mécanisme de désassemblage puis grâce à la base multiOS, K-binID associe à chaque bloc localisé une liste des noyaux connus qui contiennent un bloc avec la même signature que le bloc localisé. En fin, K-binID détermine s'il y a un noyau connu dont l'identifiant est associé à tous les blocs localisés. Dans ce cas, l'hyperviseur conclut que cet identifiant correspond à celui du noyau de la VM et que le fichier de symboles associé à cet identifiant peut être utilisé pour obtenir les adresses des fonctions noyau de la VM. Sinon, s'il y a au moins deux blocs auxquels n'est associé aucun identifiant commun de noyau, l'hyperviseur conclut que le noyau de le VM ne fait pas partie de l'ensemble des noyaux connus.

Un prototype de l'approche de K-binID a été implémenté sur l'hyperviseur KVM. Son évaluation sur une variété de noyaux Linux différents montre que K-binID permet d'identifier précisément le noyau de la VM et donc aussi le fichier de symboles associé à partir duquel les adresses des fonctions noyau peuvent être obtenues à des fins d'introspection de machine virtuelle.

L'inconvénient principal des approches d'introspection qui nécessitent l'utilisation d'un fichier de symboles, comme K-binID, est que le contenu de ce dernier est très sensible aux modifications dans le code binaire du noyau. En effet, le moindre patch appliqué au code noyau peut changer les adresses de la plupart des fonctions et rend le fichier de symboles de l'ancien noyau inutilisable sur le nouveau noyau même si la plupart du code de ce dernier reste similaire au code de l'ancien noyau. Ce cas est représentatif de l'application des mises à jour sur la VM et la personnalisation puis la récompilation des noyaux. Il peut donc rendre un système d'introspection tel que K-binID inutilisable à cause du manque du fichier de symboles approprié.

10.5.4 Localisation et identification des fonctions noyau

Pour adresser les faiblesses de l'approche de K-binID et pour permettre l'introspection des VMs dont les noyaux sont inconnus mais proches des noyaux connus, nous proposons dans cette thèse NoGap. NoGap est un système qui permet d'identifier la plupart des adresses et des noms des fonctions noyau directement dans la mémoire d'une VM sans avoir nécessairement le fichier de symboles associé au noyau de la VM.

Pour faire cela, plusieurs challenges doivent être relevés. Le premier challenge est que dans le contexte de l'introspection, l'emplacement du code noyau est inconnu pour l'hyperviseur. Le deuxième challenge est que certaines optimisations du compilateur font qu'il y a plus dans un code binaire des instructions représentatives du début ou de fin d'une fonction ce qui rend très difficile l'identification des adresses de début et de fin des fonctions noyau depuis l'hyperviseur.

Pour répondre au premier challenge, NoGap est implémenté grâce à notre désassembleur qui permet de localiser automatiquement les blocs de code du noyau de la VM. Pour le deuxième challenge, NoGap déduit les adresses de début, de fin et les noms des fonctions noyau grâce aux propriétés et aux informations

contenues dans les blocs de code qui constituent ces fonctions, et c'est la contribution principale de NoGap. Une fois les adresses et les noms des fonctions noyau sont obtenues, NoGap permet à l'hyperviseur de les instrumenter (c.-à-d. les intercepter, appeler et analyser) afin de franchir automatiquement le gap sémantique et obtenir une compréhension très fine des états et des activités d'une VM.

D'une façon similaire à K-binID, NoGap procède aussi en deux phases, une phase hors ligne d'apprentissage et une autre phase en ligne d'analyse d'identification des adresses et des noms des fonctions noyau. Dans la phase d'apprentissage, NoGap génère pour chaque noyau connu à l'aide du fichier de symboles de ce dernier les signatures des blocs de code qui constituent chaque fonction noyau. NoGap produit dans cette phase une base multiOS dont chaque entrée contient une signature d'un bloc de code à laquelle associée une liste de noms des fonctions noyau qui contiennent un bloc avec une telle signature.

Dans la deuxième phase, l'hyperviseur cherche à identifier les adresses et les noms des fonctions noyau dont les signatures sont présentes dans la base multiOS générée en amont dans la phase précédente. Cette phase est composée de deux étapes. Dans la première, NoGap localise les blocs de code du noyau à travers notre mécanisme de désassemblage puis grâce à la base multiOS, il identifie pour chaque bloc localisé une liste de fonctions noyau qui contiennent un bloc avec une telle signature. Dans une deuxième étape, NoGap parcourt la liste des blocs localisés et considère chaque ensemble de blocs successives comme une fonction noyau s'il y a un même nom de fonction noyau associé à l'ensemble de ces blocs et s'ils sont liés par le flux d'exécution. L'adresse du début du premier bloc de cet ensemble ainsi que l'adresse de fin du dernier bloc sont considérées par NoGap comme respectivement l'adresse de début et de fin d'une fonction noyau dont le nom est celui qui est associé à tous les blocs de cet ensemble.

Un prototype de l'approche de NoGap a été implémenté sur l'hyperviseur KVM et son évaluation sur une variété de noyaux Linux montre que NoGap permet d'identifier correctement les adresses et les noms de la majorité (environ 75%) des fonctions d'un noyau s'il est connu ou au moins son code binaire est similaire à celui d'un noyau connu. NoGap permet à l'hyperviseur d'intercepter l'exécution de ces fonctions noyau pour avoir une compréhension très fine de l'activité de la VM, de les appeler pour lire ou modifier et les données et les états de la VM ou de les analyser pour découvrir des informations utiles à la supervision de la sécurité, par exemple des adresses de structures de données pertinentes telles que la liste des processus actifs.

Malgré que NoGap réduit la dépendance des fichiers de symboles en identifiant des fonctions sur des noyau inconnus, le grand nombre de versions et de customisations d'OS possibles dans la pratique constitue une limitation principale pour l'approche actuelle de NoGap. Cette limitation exige que la base de signatures multiOS doive être constamment mise à jour afin de couvrir le maximum possible de signatures de blocs de code de fonctions et maintenir l'efficacité de NoGap.

10.5.5 Identification des adresses globales et des offsets des champs de données

Certaines informations très spécifiques à un noyau, telles que l'adresse de la liste de processus actifs, sont indispensables pour la supervision de sécurité d'une VM. Cependant, elles ne peuvent pas être obtenues via l'appel ou l'interception des fonctions noyau. Dans cette thèse, nous proposons dans un travail préliminaire une nouvelle approche basée sur l'analyse binaire et qui permet à l'hyperviseur d'identifier

automatiquement les adresses globales (ex: liste de processus) et les offsets des champs d'un type de structure de données (ex: le champs pid dans la structure task_struct) à travers l'analyse des instructions des fonctions qui accèdent à ces adresses et offsets.

Cette approche procède aussi en deux phases, une phase hors ligne d'apprentissage et une autre phase on ligne d'analyse. Dans la première phase, l'hyperviseur génère grâce à un noyau connu une liste de fonctions noyau qui accèdent à une adresse globale ou une offset donnée. Actuellement cette phase est effectuée manuellement par l'utilisateur de notre approche.

Dans la deuxième phase, l'hyperviseur cherche à identifier une adresse globale ou une offset d'un champ de donnée sur un noyau qui n'est pas forcément pas connu mais dont les adresses et les noms des fonctions sont connus. Pour cela, l'hyperviseur analyse les instructions de chaque fonction dans la liste générée dans la première phase et extrait les adresses et les offsets référencées par un chaque fonction. Ensuite, l'hyperviseur considère l'adresse ou l'offset partagée par toutes les fonctions analysées comme l'adresse ou l'offset recherchée.

Un prototype de cette approche a été implémenté comme une extension de NoGap sur l'hyperviseur KVM, et son évaluation montre qu'à travers l'analyse de quelques fonctions stables dans le noyau Linux, l'hyperviseur peut déduire automatiquement sur un grand nombre de noyau Linux l'adresse de la liste de processus actifs ainsi que les offsets de plusieurs champs de données dans un descripteur de processus (ex: pid, processus suivant, précédant, etc.)

10.5.6 Détection de processus caché à travers l'instrumentation de fonctions noyau

Pour illustrer comment une application de supervision de sécurité franchit le gap sémantique grâce à NoGap via les possibilités d'interception, d'appel et d'analyse des fonctions noyau depuis l'hyperviseur, nous proposons dans cette thèse une application, appelé HPD, de détection et d'arrêt de l'exécution d'un processus caché.

Un processus caché est un processus malveillant qui supprime son descripteur de la liste de processus actifs afin de devenir invisible pour les outils de sécurité et d'administration système tout en continuant de s'exécuter normalement [132]. Plusieurs système d'introspection ont adressé le problème de détection de processus caché via le croisement d'une vue fiable de la liste de processus qui n'est impactée par le processus caché avec une deuxième vue non fiable dans laquelle n'apparait pas le processus caché. La différence entre ces deux vues censées être équivalentes implique donc la présence d'un processus caché [78][109]. Cependant, ces systèmes existants ne conviennent pas à une utilisation dans la pratique dans un cloud IaaS car généralement ils utilisent des informations fournies manuellement par un administrateur de sécurité pour construire les deux vues nécessaires pour la détection de processus caché. L'approche de HPD est aussi basée sur le croisement de deux vues sur les processus actifs dans une VM, mais contrairement aux travaux existants, HPD obtient ces vues de façon automatique et portable sur beaucoup de noyaux grâce à l'instrumentation des fonctions stables sur l'ensemble des noyaux Linux.

Pour détecter l'existence d'un processus actif et caché dans une VM, HPD exploite le fait que tout processus actif est forcément ordonnancé périodiquement par l'OS de la VM à travers une fonction noyau. Dans une première étape, HPD obtient une vue fiable sur les processus actifs dans une VM à travers l'interception

d'une fonction d'ordonnancement qui manipule les descripteurs des processus actifs. Pour chaque interception, HPD vérifie dans une deuxième étape si le processus que l'OS tente d'ordonnancer est visible dans la vue non fiable obtenue de la liste de processus actifs déclarée par l'OS. HPD identifier automatiquement l'adresse de la liste de processus ainsi que les offsets des différents champs de données intéressants pour la détection de processus caché (pid, processus suivant et précédant, etc.) grâce à l'analyse des instructions des fonctions qui utilisent ces informations. Si un processus en cours d'ordonnancement n'est pas visible dans la liste des processus actifs, HPD conclut que ce processus et un processus caché. Dans ce cas, HPD réinsère ce processus dans la liste de processus actifs de l'OS et appelle depuis l'hyperviseur une fonction noyau de la VM pour arrêter l'exécution du processus caché détecté.

Un prototype de HPD a été implémenté comme une extension de NoGap et testé sur une VM qui contient un processus caché. HPD a correctement détecté et arrêter l'exécution du processus caché. Nous avons mesuré l'impact de HPD sur la performance de la VM, HPD cause une perte de performance qui augmente plus la VM exécute des applications parallélisées d'entrées/sorties et diminue plus la VM exécute des applications de calcules non parallélisées.

10.6 Conclusion

Dans cette thèse nous avons présenté comment la virtualisation matérielle a permis la création d'une nouvelle approche de supervision de sécurité dans le cloud qui opère depuis l'hyperviseur. Cette approche appelé introspection de machine virtuelle, était le sujet de nombreux travaux de recherche qui ont adressé le problème du gap sémantique entre la vue sémantique in-VM et la vue brute de l'hyperviseur des états et des activités des VMs. Cependant, les systèmes d'introspection proposés dans l'état de l'art ne conviennent pas à une utilisation dans la pratique dans des enivrements cloud de type IaaS. Ces systèmes existants franchissent une partie limitée du gap sémantique, nécessitent une assistance manuelle par un administrateur, utilisent un agent intrusif dans la VM, doivent être réadaptés pour chaque type et version d'OS ou posent des problèmes de performance. L'objectif principal de cette thèse était de proposer une approche d'introspection qui franchit le gap sémantique de façon applicable dans la pratique aux environnements cloud de typer IaaS.

Pour atteindre cet objectif, nous avons étudié puis classifié les approches et les applications des travaux existants d'introspection afin de comprendre leurs propriétés et limitations. Dans cette étude, nous avons observé que les approches d'introspection qui appliquent des techniques d'analyse et d'instrumentation de code binaire sur une partie du code noyau d'une VM (ex: appels systèmes) arrivent à franchir une grande partie du gap sémantique ou améliorer leur automaticité, indépendance de l'OS de la VM ou l'impact sur la performance de la VM. Cependant, aucune de ces approches regroupe à la fois toutes les propriétés nécessaires pour être utilisée dans la pratique dans des environnements cloud de type IaaS.

Dans cette thèse nous avons exploré l'application depuis l'hyperviseur des techniques d'analyse et d'instrumentation de code binaire sur le code noyau d'une VM, c.-à-d. sur toutes les fonctions noyau, afin de réduire largement le gap sémantique de façon non intrusive, automatique et largement indépendante de l'OS. Pour cela, l'hyperviseur doit d'abord localiser le code noyau dans la mémoire de la VM. L'emplacement du code noyau dans la mémoire d'une VM est inconnu pour l'hyperviseur car, il est d'une

part très dépendant du type et de la version de l'OS de la VM et d'autre part il change de manière aléatoire au redémarrage de la VM pour des raisons de sécurité. Dans cette thèse, nous avons proposé un mécanisme de désassemblage de code binaire qui permet à l'hyperviseur de découvrir le code binaire du noyau d'une VM, fragment par fragment (appelé bloc de code), à partir d'une seule instruction noyau arbitraire identifiée automatiquement par l'hyperviseur dans la mémoire de la VM. Dans cette mécanisme, nous avons introduit une nouvelle technique de désassemblage en marche arrière qui exploite le phénomène d'autoréparation sur l'architecture x86 afin d'identifier correctement des frontières d'instructions qui précèdent une adresse donnée.

Une fois l'emplacement du code noyau dans la mémoire est identifié, l'hyperviseur doit ensuite obtenir les adresses et les noms des fonctions noyau dans la mémoire d'une VM. Un moyen possible pour les obtenir est d'identifier quel code binaire s'exécute dans le noyau de la VM puis d'extraire les adresses et les noms des fonctions noyau à partir du fichier de symboles associé au noyau identifié. Les techniques existantes d'identification de noyau ne conviennent pas à une utilisation dans un cloud IaaS car soit elles ne sont pas précises ou supportent uniquement des noyaux compilés avec des options spécifiques qui facilitent la localisation de fragments de code noyau dans la mémoire d'une VM. Nous avons présenté dans cette thèse K-binID, un système d'introspection basée sur notre désassembleur de code noyau, qui permet une identification précise du code noyau d'une VM. K-binID définit des signatures d'un noyau au niveau des blocs de code de ce dernier et considère l'ensemble des signatures des blocs de code d'un noyau comme l'empreinte du code binaire de ce noyau. Cette approche de K-binID permet une identification précise de code binaire du noyau d'une VM indépendamment des difficultés posées par les options de compilation ou randomisation d'un noyau, et cela permet à l'hyperviseur d'identifier précisément de quel fichier de symboles les adresses et les noms des fonctions noyau peuvent être obtenus.

La principale limitation de K-binID est qu'il existe dans la pratique un grand nombre de versions et de customisations uniques de code binaire de noyau. Cela implique qu'il a y dans la pratique autant de fichiers de symboles uniques que de versions et de customisations de code noyau. K-binID ne permet pas d'obtenir les adresses et les noms des fonctions d'un noyau inconnu dont le fichier de symboles est indisponible même si le code binaire de ce noyau inconnu est très proche de celui d'un noyau connu. Pour répondre à cette limitation de K-binID, nous avons exploré dans cette thèse la possibilité d'identifier les adresses et les noms des fonctions noyau directement dans la mémoire d'une VM sans avoir forcément son fichier de symboles.

Le principal challenge de cette approche est que certaines options de compilation de noyau font qu'il n'y a plus d'instructions qui marquent le début ou la fin d'une fonction dans un code binaire. Pour répondre à ce challenge, nous avons développé NoGap qui est un système d'introspection basé sur notre désassembleur de code noyau et qui permet de déduire la majorité (75% en moyenne) des adresses et des noms des fonctions noyau grâce aux adresses, signatures binaires et le flux d'exécution des blocs de code localisés dans le code noyau à travers notre désassembleur. Grâce à la grande similarité du code binaire des noyaux proches en termes de version et de customisation, NoGap permet à l'hyperviseur d'identifier la majorité des adresses et des noms des fonctions d'un noyau inconnu dont le code binaire est proche de celui d'un noyau connu. Cette propriété de NoGap le rend encore plus adapté à une utilisation dans un cloud IaaS que K-binID.

NoGap permet à l'hyperviseur d'intercepter, appeler et analyser les fonctions noyau localisées dans la

mémoire d'une VM pour réduire largement le gap sémantique et obtenir automatiquement une compréhension très fine des états et des activités de la VM. Nous avons présenté à la fin de cette thèse comment les fonctions noyau peuvent peut être instrumentées par NoGap afin de détecter et arrêter automatiquement l'exécution d'un processus caché.

Les contributions de cette thèse avancent significativement l'état de l'art d'introspection de machine virtuelle et permettent à un opérateur cloud de disposer d'une infrastructure intelligente qui soit consciente sur le plan sémantique de l'activité des VMs hébergées et capable de fournir des services de sécurité à valeur ajoutée.

Bibliography

[1] Amazon Elastic Cloud. aws.amazon.com/ec2. [Online; accessed 31-January-2017].

[2] Apache bench - apache http server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html. [Online; accessed 31-January-2017].

[3] "Clang" C Language Family Frontend for LLVM. https://clang.llvm.org/. [Online; accessed 31-January-2017].

[4] Compute Engine - IaaS. https://cloud.google.com/compute/. [Online; accessed 31-January-2017].

[5] Direct Kernel Object Manipulation. http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf.

[6] Disk encryption software. https://www.truecrypt71a.com/. [Online; accessed 31-january-2017].

[7] GCC, the GNU Compiler Collection. https://gcc.gnu.org/. [Online; accessed 31-January-2017].

[8] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/. [Online; accessed 31-January-2017].

[9] Gmail - Google. https://www.google.com/gmail/. [Online; accessed 31-January-2017].

[10] Google app engine. https://appengine.google.com/. [Online; accessed 31-January-2017].

[11] Google drive. https://www.google.com/intl/fr_ALL/drive/. [Online; accessed 31-January-2017].

[12] IBM Smartcloud Application. https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki [Online; accessed 31-January-2017].

[13] IDA F.L.I.R.T. Technology: In-Depth. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml. [Online; accessed 31-January-2017].

[14] IDA Pro Disassembly. https://www.hex-rays.com/products/ida/. [Online; accessed 31-January-2017].

[15] Kernel address space layout randomization. https://lwn.net/Articles/569635/.

[16] Linux Crash Utility. http://people.redhat.com/anderson/.

[17] Linux Cross Reference. http://lxr.free-electrons.com/. [Online; accessed 31-January-2017].

[18] Microsoft Azure. azure.microsoft.com. [Online; accessed 31-January-2017].

[19] Microsoft Hyper-V. https://msdn.microsoft.com/fr-fr/library/mt169373 [Online; accessed 31-January-2017].

[20] Microsoft office 365. https://login.microsoftonline.com/. [Online; accessed 31-January-2017].

[21] Microsoft Visual Studio. https://www.visualstudio.com/. [Online; accessed 31-January-2017].

[22] objdump - GNU Binary Utilities -. https://sourceware.org/binutils/docs/binutils/objdump.html. [Online; accessed 31-January-2017].

[23] Oracle VirtualBox. https://www.virtualbox.org/. [Online; accessed 31-January-2017].

[24] The FreeBSD Project. https://www.freebsd.org/.

[25] The LLVM Compiler Infrastructure. http://www.llvm.org. [Online; accessed 31-January-2017].

[26] Udis86 Disassembler Library for x86 / x86-64, howpublished =.

[27] VMware ESXi. http://www.vmware.com/fr/products/esxi-and-esx.html. [Online; accessed 31-January-2017].

[28] VMware Workstation. http://www.vmware.com/products/workstation.

[29] Volatility Framework. http://www.volatilityfoundation.org/.

[30] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGOPS Operating Systems Review*, 40(5):2–13, 2006.

[31] Ferrol Aderholdt, Fang Han, Stephen L Scott, and Thomas Naughton. Efficient checkpointing of virtual machines using virtual machine introspection. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 414–423. IEEE, 2014.

[32] Irfan Ahmed, Vassil Roussev, and Aisha Ali Gombe. Robust fingerprinting for relocatable code. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 219–229. ACM, 2015.

[33] AMD. Secure Virtual Machine Architecture Reference Manual.

[34] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium*, 2016.

[35] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-Agnostic Function Detection in Binaries.

[36] S. Bahram, Xuxian Jiang, Zhi Wang, M. Grace, Jinku Li, D. Srinivasan, Junghwan Rhee, and Dongyan Xu. DKSM: Subverting Virtual Machine Introspection for Fun and Profit. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, pages 82–91, Oct 2010.

[37] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860, 2014.

[38] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[39] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A Survey on Hypervisor-Based Monitoring: Approaches, Applications, and Evolutions. *ACM Computing Surveys (CSUR)*, 48(1):10, 2015.

[40] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[41] Chris Benninger, Stephen W Neville, Yagiz Onat Yazir, Chris Matthews, and Yvonne Coady. Maitland: Lighter-weight VM introspection to support cyber-security in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 471–478. IEEE, 2012.

[42] Andrew R Bernat and Barton P Miller. Incremental call-path profiling. *Concurrency and Computation: Practice and Experience*, 19(11):1533–1547, 2007.

[43] Andrew R Bernat and Barton P Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16. ACM, 2011.

[44] Sebastian Biedermann, Stefan Katzenbeisser, and Jakub Szefer. Hot-hardening: getting more out of your security settings. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 6–15. ACM, 2014.

[45] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[46] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Transactions on Computer Systems (TOCS)*, 30(4):12, 2012.

[47] K. Burns, A. Barbalace, V. Legout, and B. Ravindran. KairosVM: Deterministic introspection for real-time virtual machine hierarchical scheduling. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–8, Sept 2014.

[48] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. Technical report, DTIC Document, 2009.

[49] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Research in Attacks, Intrusions, and Defenses*, pages 22–41. Springer, 2012.

[50] P.M. Chen and B.D. Noble. When virtual is better than real [operating system relocation to virtual machines]. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 133–138, May 2001.

[51] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries. In *NDSS*, 2015.

[52] Yingxin Cheng, Xiao Fu, Bin Luo, Rui Yang, and Hao Ruan. Investigating the Hooking Behavior: A Page-Level Memory Monitoring Method for Live Forensics. In ShermanS.M. Chow, Jan Camenisch, LucasC.K. Hui, and SiuMing Yiu, editors, *Information Security*, volume 8783 of *Lecture Notes in Computer Science*, pages 255–272. Springer International Publishing, 2014.

[53] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. Introspection-based memory de-duplication and migration. In *ACM SIGPLAN Notices*, volume 48, pages 51–62. ACM, 2013.

[54] Mihai Christodorescu, Reiner Sailer, Douglas Lee Schales, Daniele Sgandurra, and Diego Zamboni. Cloud security is not (just) virtualization security: a short paper. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 97–102. ACM, 2009.

[55] Cristina Cifuentes and K John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[56] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[57] Intel Corp. Intel 64 and IA-32 Architectures Software Developer's Manuals. http://www.intel.com/products/processor/manuals/.

[58] Timothy W. Curry. Profiling and Tracing Dynamic Library Usage via Interposition. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 18–18, Berkeley, CA, USA, 1994. USENIX Association.

[59] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.

[60] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. REV.NG: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141. ACM, 2017.

[61] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[62] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 839–850. ACM, 2013.

[63] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2011.

[64] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 23th Symposium on Network and Distributed System Security (NDSS)*, 2016.

[65] Qian Feng, Aravind Prakash, Minghua Wang, Curtis Carmony, and Heng Yin. Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 11–22. ACM, 2016.

[66] Yangchun Fu and Zhiqiang Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 586–600, May 2012.

[67] Yangchun Fu and Zhiqiang Lin. EXTERIOR: Using a dual-VM Based External Shell for guest-OS Introspection, Configuration, and Recovery. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 97–110, New York, NY, USA, 2013. ACM.

[68] Yangchun Fu, Zhiqiang Lin, and Kevin W Hamlen. Subverting system authentication with context-aware, reactive virtual machine introspection. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 229–238. ACM, 2013.

[69] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. HyperShell: A Practical Hypervisor Layer Guest OS Shell for Automated In-VM Management. In *Proceedings of the 2014 USENIX Annual Technical Conference*, Philadephia, PA, June 2014.

[70] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[71] Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, and Heng Yin. OS-Sommelier: Memory-only Operating System Fingerprinting in the Cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 5:1–5:13, New York, NY, USA, 2012. ACM.

[72] Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, and Heng Yin. Multi-Aspect, Robust, and Memory Exclusive Guest OS Fingerprinting. *IEEE Transactions on Cloud Computing*, 2014.

[73] Yufei Gu and Zhiqiang Lin. Derandomizing Kernel Address Space Layout for Introspection and Forensics. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, New Orelans, LA, 2016. ACM.

[74] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process Implanting: A New Active Introspection Framework for Virtualization. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, SRDS '11, pages 147–156, Washington, DC, USA, 2011. IEEE Computer Society.

[75] Yacine Hebbal, Sylvie Laniepce, and Jean-Marc Menaud. Virtual Machine Introspection: Techniques and Applications. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 676–685. IEEE, 2015.

[76] Chris Horne. Understanding full virtualization, paravirtualization and hardware assist. *White paper, VMware Inc*, 2007.

[77] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E Porter, and Radu Sion. Sok: Introspections on trust and the semantic gap. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 605–620. IEEE, 2014.

[78] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 128–138, New York, NY, USA, 2007. ACM.

[79] RW Jones and M Booth. Virt-inspector-display operating system version and other information about a virtual machine.

[80] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 1–1, Berkeley, CA, USA, 2006. USENIX Association.

[81] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based Hidden Process Detection and Identification Using Lycosid. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 91–100, New York, NY, USA, 2008. ACM.

[82] 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 Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 91–104, New York, NY, USA, 2005. ACM.

[83] Thomas Kittel, Sebastian Vogl, Tamas K Lengyel, Jonas Pfoh, and Claudia Eckert. Code validation for modern OS kernels. In *Workshop on Malware Memory Forensics (MMF)*, 2014.

[84] 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.

[85] R Krishnakumar. Kernel korner: kprobes-a kernel debugger. *Linux Journal*, 2005(133):11, 2005.

[86] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183. IEEE, 2010.

[87] Tamas K Lengyel, Steve Maresca, Bryan D Payne, George D Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 386–395. ACM, 2014.

[88] Shengying Li. A survey on tools for binary code analysis. *URL http://www. ecsl. cs. sunysb. edu/tr/BinaryAnalysis. doc*, 2004.

[89] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.

[90] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[91] Lionel Litty, H Andrés Lagar-Cavilla, and David Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *USENIX Security Symposium*, pages 243–258, 2008.

[92] Lionel Litty and David Lie. Manitou: A Layer-below Approach to Fighting Malware. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 6–11, New York, NY, USA, 2006. ACM.

[93] Andrei Luțaș, Adrian Coleșa, Sándor Lukács, and Dan Luțaș. U-HIPE: hypervisor-based protection of user-mode processes in windows. *Journal of Computer Virology and Hacking Techniques*, 12(1):23–36, 2016.

[94] Gordon Fyodor Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, 2009.

[95] Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. 2011.

[96] Xiaozhu Meng and B Miller. Binary code is not easy. Technical report, Tech. rep., Computer Sciences Department, University of Wisconsin, Madison, 2015.

[97] James Newsome. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. 2005.

[98] James Newsome, David Brumley, and Dawn Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. 2006.

[99] Eunbyung Park, Bernhard Egger, and Jaejin Lee. Fast and space-efficient virtual machine checkpointing. In *ACM SIGPLAN Notices*, volume 46, pages 75–86. ACM, 2011.

[100] Bryan D Payne. Simplifying virtual machine introspection using libVMI. *Sandia Report*, 2012.

[101] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 233–247. IEEE, 2008.

[102] Bryan D Payne, MDP De Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 385–397. IEEE, 2007.

[103] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 709–724. IEEE, 2015.

[104] 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.

[105] 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.

[106] Cuong Pham, Z. Estrada, Phuong Cao, Z. Kalbarczyk, and R.K. Iyer. Reliability and Security Monitoring of Virtual Machines Using Hardware Architectural Invariants. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 13–24, June 2014.

[107] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[108] Nguyen Anh Quynh. Operating system fingerprinting for virtual machines. *Proc. DEFCON*, 18, 2010.

[109] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Kernel malware analysis with untampered and temporal views of dynamic kernel memory. In *Recent Advances in Intrusion Detection*, pages 178–197. Springer, 2010.

[110] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[111] Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):34, 2004.

[112] Nathan E Rosenblum, Xiaojin Zhu, Barton P Miller, and Karen Hunt. Learning to Analyze Binary Computer Code. In *AAAI*, pages 798–804, 2008.

[113] Mark Russinovich, David Solomon, and Alex Ionescu. *Windows internals*. Pearson Education, 2012.

[114] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. Hybrid-Bridge: Efficiently Bridging the Semantic-Gap in Virtual Machine Introspecrtion via Decoupled Execution and Training Memoization. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA*, 2014.

[115] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Reverse engineering, 2002. Proceedings. Ninth working conference on*, pages 45–54. IEEE, 2002.

[116] 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 Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[117] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[118] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM Monitoring Using Hardware Virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 477–487, New York, NY, USA, 2009. ACM.

[119] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 611–626, 2015.

[120] K Skarfone and Peter Mell. Guide to intrusion detection and prevention systems.

[121] 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*, pages 363–374. ACM, 2011.

[122] Abhinav Srivastava and Jonathon Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *Recent Advances in Intrusion Detection*, pages 39–58. Springer, 2008.

[123] Sahil Suneja, Canturk Isci, Eyal de Lara, and Vasanth Bala. Exploring VM Introspection: Techniques and Trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 133–146. ACM, 2015.

[124] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2002.

[125] Donghai Tian, Xi Xiong, Changzhen Hu, and Peng Liu. Integrating Offline Analysis and Online Protection to Defeat Buffer Overflow Attacks. In Mike Burmester, Gene Tsudik, Spyros Magliveras, and Ivana Ilić, editors, *Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 409–415. Springer Berlin Heidelberg, 2011.

[126] David Urbina, Yufei Gu, Juan Caballero, and Zhiqiang Lin. SigPath: A Memory Graph Based Approach for Program Data Introspection and Modification. In *Computer Security-ESORICS 2014*, pages 237–256. Springer, 2014.

[127] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 927–940. ACM, 2015.

[128] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[129] Gary Wang, Zachary John Estrada, Cuong Manh Pham, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring. In *WOOT*, 2015.

[130] 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*, pages 545–554. ACM, 2009.

[131] Rui Wu, Ping Chen, Peng Liu, and Bing Mao. System Call Redirection: A Practical Approach to Meeting Real-World Virtual Machine Introspection Needs. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 574–585, June 2014.

[132] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. Automatic Uncovering of Tap Points from Kernel Executions. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 49–70. Springer, 2016.

[133] Junyuan Zeng and Zhiqiang Lin. Towards Automatic Inference of Kernel Object Semantics from Binary Code. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15)*, Kyoto, Japan, November 2015.

[134] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[135] Hao Zhang, Lei Zhao, Lai Xu, Lina Wang, and Deming Wu. vPatcher: VMI-Based Transparent Data Patching to Secure Software in the Cloud. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 943–948, Sept 2014.

[136] Lingchen Zhang, Sachin Shetty, Peng Liu, and Jiwu Jing. RootkitDet: Practical End-to-End Defense against Kernel Rootkits in a Cloud Environment. In *Computer Security-ESORICS 2014*, pages 475–493. Springer, 2014.

UNIVERSITE BRETAGNE LOIRE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de Doctorat

Yacine HEBBAL

Mécanismes de monitoring sémantique dédiés à la sécurité des infrastructures cloud IaaS

Semantic monitoring mechanisms dedicated to security monitoring in IaaS cloud

Résumé
L'introspection de machine virtuelle (VM) consiste à superviser les états et les activités de celles-ci depuis la couche de virtualisation, tirant ainsi avantage de son emplacement qui offre à la fois une bonne visibilité des états et des activités des VMs ainsi qu'une bonne isolation de ces dernières. Cependant, les états et les activités des VMs à superviser sont vus par la couche de virtualisation comme une suite binaire de bits et d'octets en plus des états des ressources virtuelles. L'écart entre la vue brute disponible à la couche de virtualisation et celle nécessaire pour la supervision de sécurité des VMs constitue un challenge pour l'introspection appelé « le fossé sémantique ».
Pour obtenir des informations sémantiques sur les états et les activités des VMs à fin de superviser leur sécurité, nous présentons dans cette thèse un ensemble de techniques basé sur l'analyse binaire et la réutilisation du code binaire du noyau d'une VM. Ces techniques permettent d'identifier les adresses et les noms de la plupart des fonctions noyau d'une VM puis de les instrumenter (intercepter, appeler et analyser) pour franchir le fossé sémantique de manière automatique et efficiente même dans les cas des optimisations du compilateur et de la randomisation de l'emplacement du code noyau dans la mémoire de la VM.

Mots clés
Introspection de Machine Virtuelle, Le Fossé Sémantique, Analyse Binaire Statique, Réutilisation du Code Binaire.

Abstract
Virtual Machine Introspection (VMI) consists in monitoring VMs security from the hypervisor layer which offers thanks to its location a strong visibility on their activities in addition to a strong isolation from them. However, hypervisor view of VMs is just raw bits and bytes in addition to hardware states. The semantic difference between this raw view and the one needed for VM security monitoring presents a significant challenge for VMI called "the semantic gap". In order to obtain semantic information about VM states and activities for monitoring their security from the hypervisor layer, we present in this thesis a set of techniques based on analysis and reuse of VM kernel binary code. These techniques enable to identify addresses and names of most VM kernel functions then instrument (call, intercept and analyze) them to automatically bridge the semantic gap regardless of challenges presented by compiler optimizations and kernel base address randomization.

Key Words
Virtual Machine Introspection, Semantic Gap, Static Binary Analysis, Binary Code Reuse.