CHALMERS

Security Functions for Virtual Machines via Introspection

Master of Science Thesis in the Programme Network and Distributed System

Mazdak Rajabi Nasab

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2012

Security Functions for Virtual Machines via Introspection

MAZDAK RAJABI NASAB

Examiner: ERLAND JONSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2012

Abstract

The recent renaissance of virtualization brought with it the resurgence of ideas for hypervisor based security services. As such, virtual machine introspection (VMI) has been proposed for both passive and active monitoring. While passive monitoring is the method for detecting intrusions, active monitoring allows intervention of a Virtual Machine (VM) behavior, which is proper for intrusion prevention. Several VMI techniques for security purposes have been deployed in different virtualization solutions. XenProbes, XenAccess, and Ether are examples of deployed VMI for Xen.

The goal of this thesis is the design and the implementation of a security function that actively monitors the integrity aspect of guest virtual machines. OS debugging is the method used for active VMI. In this method, Xen built-in capability for OS debugging is used, to control, and to intervene in the behavior of guest virtual machines.

A well-known drawback of VMI in "high-rate" applications is the cost of context switches between the trusted monitor and the virtual machine being monitored. As a result, “low-rate” security functions are probably more suitable candidates for VMI applications. The proposed security functions are low-rate solutions for systems' integrity property. In the attempt to define proper low-rate security functions different available filesystem integrity solutions like DigSig and IMA are surveyed.

As DigSig is limited to ELF files and IMA is not developed completely and is not immune against rootkits, a new security function is developed in this thesis. In this process, IMA is used as the basis of the designed security function. The security function validates the RSA signature of accessed files in guest virtual machines. It prevents file access in case of violation. This security function starts early in the boot process of a guest VM to properly ensure its integrity property. Having implemented the security function, its security strength, performance, and limitations are analyzed. Finally it is concluded, while this security function imposes negligible performance penalty, it improves the security attributes of a virtual machine.

Keywords: Virtual Machine Introspection, VMI, OS Debugging, Kernel Debugging, Filesystem Integrity

Acknowledgments

This report constitutes my Master of Science thesis at Chalmers University of Technology. The work has been done at Ericsson Research.

First, I would like to thank my supervisor at Ericsson Research, **Andras Mehes** for helping me to get started and for his advices that made problems easier for me.

I also would like to acknowledge my supervisor and examiner at Chalmers University of Technology, **Erland Jonsson** for helping me to manage my thesis, and for helping me to prepare this report.

In addition, I wish to thank **Fredric Morenius** at Ericsson Research, and **Lars Rasmusson**, at SICS. Thank you for all ideas, and discussions during this time.

I am grateful to my wonderful family and my friends. Their love and patience were the best support during my master studies.

Lastly, and most importantly, I wish to give special thanks to my sister, **Nina Rajabi Nasab** who has always supported her little brother. To her I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Description

Integrity is one aspect of security. Wikipedia defines integrity:

"Integrity is a concept of consistency of actions, values, methods, measures, principles, expectations, and outcomes. In ethics, integrity is regarded as the honesty and truthfulness or accuracy of one's actions. Integrity can be regarded as the opposite of hypocrisy, in that it regards internal consistency as a virtue, and suggests that parties holding apparently conflicting values should account for the discrepancy or alter their beliefs."[1]

Filesystem integrity is the property of keeping filesystem states, such as filesystem structure, content of files, in a known valid state. Although there can be many definitions of a valid state, in my opinion it is up to the system administrator to define.

Filesystem integrity is an important part of system security. Having compromised a system, the attacker tries to maintain its access by maliciously altering files or by installing malware on filesystem which is usually a non-volatile memory. In addition violating filesystem integrity may lead to system compromise in first place. For these reasons, it is vital for system security to guarantee its integrity including filesystem integrity.

Some solutions are available for filesystem integrity, but they all have flaws or limitations. **Tripwire**[2] is a well-known security function that calculates hash of files and stores them locally; then tripwire re-calculates file hashes on regular time intervals to find changes in filesystem[1]. However Tripwire is a user space solution, which makes it vulnerable to kernel rootkits. Kernel rootkits, which

[1] http://en.wikipedia.org/wiki/Integrity, April 2012
[2] http://sourceforge.net/projects/tripwire

are common today, disable security functions like tripwire from privileged space, i.e. kernel space. In addition tripwire is a passive security mechanism; therefore it cannot prevent malicious modifications in filesystem. **DigSig**[2] is another attempt to solve the filesystem integrity problem. DigSig signs executable files and verifies their signatures whenever those executable files are "mmap"ed into memory for execution. Although it is more effective than tripwire, still it is vulnerable to kernel rootkits. DigSig works only with ELF executable files; which means it cannot monitor other important files like scripts. In addition, as DigSig keys are stored in the kernel space, they are accessible by kernel rootkits. There are other solutions for filesystem integrity, which are designed for either user space or kernel space. As a result they are vulnerable to kernel exploits; and not all of them are able to prevent filesystem compromise.

Since the highest privilege level in an OS is the kernel space, any solution works in the kernel space is considered ineffective against kernel rootkits. Recently, and mostly after the advent of the processor virtualization technology, which provided the basis for native virtualization, virtualization has been used for security purposes. **Advanced Intrusion Detection Environment (AIDE)**[3] is an equivalent of tripwire that takes advantage of virtualization. AIDE uses its agents in guest virtual machines to gather information about the guest VMs filesystem. It then repeats this information gathering about the target file systems to find modifications[3]. Even though AIDE is more effective than tripwire, it is still a passive security function. While detection is beneficial, prevention is desired.

Finally, most security functions for integrity property check file contents in filesystem. However a malware can alter a file when it is loaded in memory. As a result, to address integrity aspect of a system, it is important to verify file integrity when it is loaded in memory. **Integrity Measurement Architecture (IMA)**[4] is a new security function in the Linux kernel, which checks files integrity when they are mapped into memory[4]. However, not all IMA modules are developed yet, and it is still a kernel space solution.

1.2 Goal

The goal of this thesis is the design and the implementation of a security function that actively monitors the integrity aspect of guest virtual machines. The security function designed in this thesis, is RSA signature verification of files when they are loaded in guest VM memory. All important files, generally files owned by root user, in a guest virtual machine are signed with a private key in a trusted machine. When the target VM is created, the signature of important files in that VM is verified by the proper public key. In case of violation, access

[3] http://aide.sourceforge.net
[4] http://linux-ima.sourceforge.net/

to that file is denied, otherwise permitted. The signature verification has to be performed in a trusted domain in a virtualized environment, and preferably only the required public key should be known to the trusted domain. The private key is secret and only used for virtual machine preparation. This key policy is designed to improve the security attributes of guest virtual machines and virtual environment as will be described in section3.1. To properly guarantee the integrity requirement of guest virtual machines, the security function, must start early enough in boot process. It has to start either when the guest virtual machine is created or early in its boot process. This approach ensures that integrity of all accessed files in the target VM is checked.

1.3 Challenges

To fulfill the project goals, some problems have to be solved. Semantic gap is the difference between understanding of an external entity, i.e. VM or hypervisor, about the internal aspect of the target VM, and what is actually happening inside that VM. This gap is a major problem not only for this project but also for any other VMI system. This will be discussed further in section 2.2.2.

Having solved the semantic gap, a method has to be found to retrieve required information and a way to enforce proper policies to the guest VM. In addition, combining the methods chosen for VMI and the tool used for policy enforcement is the point where the novel idea of designing this security function resides. Section 3.1 focuses on these issues.

Another important limitation is performance penalty. Since the virtual machine introspection causes context switching, system performance is reduced by the number of context switching that happens in the system. So it is important to design security function in a way that the least number of context switching occurs. Performance penalty is discussed in section 2.2.1.

Finally, it is desirable to introspect a virtual machine as soon as possible; therefore virtual machine introspection should start with the creation of a virtual machine, or early in the boot process of virtual machines. This will be explained in section 3.6. Solving these problems are complex procedures. The solutions used in these security functions are described in this report.

1.4 Document Organization

In chapter 2 the background information in the area of virtualization and virtual machine introspection are given. In the last part of chapter 2, two available security functions for the integrity of the Linux systems are described. One of those security functions is used in this thesis. Chapter 3 explains system specifications, and design goals. Chapter 4 focuses on the design and implementation

of the security function, while in chapter 5 possible attacks, performance analysis, and limitations of the designed systems are discussed. Finally chapter 6 talks about the future work, and summarizes the project.

Chapter 2

Background

In this chapter, the information that the security functions rely on is presented. In the first section a brief description of virtualization is given. Then detailed information about Virtual Machine Introspection (VMI), its methods, and available libraries are described. Finally two security functions for the integrity of the Linux systems are reviewed. One of them is used in this thesis.

2.1 Virtualization

In a legacy Operating System (OS), kernel is the entity which controls hardware, and it has the highest privileges. To virtualize a VM, OS should be tricked to think that it is controlling hardware, while actually another entity named Virtual Machine Monitor (VMM) or hypervisor is controlling it. VMM is responsible for handling hardware. It should give the guest OS the illusion of running on the top of hardware. There are two types of VMM[5]:

1. Type I**:** VMM resides exactly on the top of hardware, and controls it directly, depicted in Figure 2.1.

2. Type II**:** VMM resides on the top on another OS and uses its services to interface with hardware, depicted in Figure 2.2.

Since Type I VMMs interface hardware directly they have better performance, and they are more reliable. Therefore type I VMM with native virtualization is used in this project. Native virtualization was not possible until the advent of processor virtualization technology. Before that, paravirtualization was used for type I VMMs. To illustrate native virtualization a concept named protection ring need to be explained.

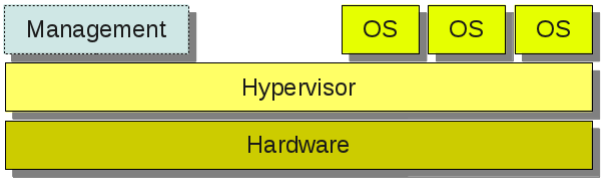

Figure 2.1: VMM type I

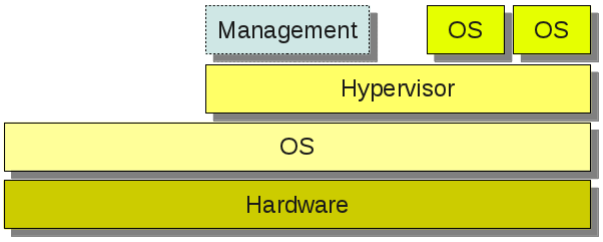

Figure 2.2: VMM type II

2.1.1 Protection Rings

Each CPU has a set of privilege levels which are called "protection rings"[6]. In each ring only a set of defined instructions is permitted to be executed. Executing an instruction which needs higher privilege causes an exception interrupts to be generated. Figure 2.3 depicts the concept of protection rings.

This picture shows x86 architecture of protection rings in Intel and AMD CPUs. In x86 architecture, there are 4 rings and ring 0 is the highest privileged ring. Only small number of instructions need ring 0 privileges, while most instructions only need ring 3 privileges. In operating systems, the kernel runs in the ring 0. So it has full access to all resources, and hardware. User space applications run in ring 3. Ring 1 and 2 are designed for drivers, but in practice they have never been used. The reason is, the number of protection rings differs between different CPU architecture. If an OS is designed to use all protection rings in a specific CPU architecture, it loses its portability to other CPU architectures. However, most processors have at least two protection rings, so designing an OS for two protection rings does not affect its portability. That is the reason almost all major OSs are designed for two protection rings. Kernel runs in the most privileged ring, and user space applications runs in least privileged ring[6].

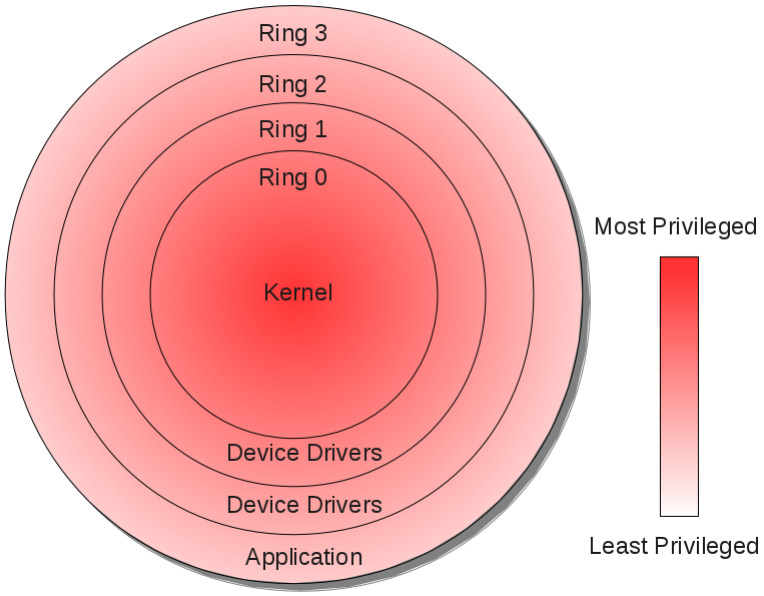

Figure 2.3: protection rings

2.1.2 Paravirtualization

As what mentioned earlier, to virtualize an OS, VMM should control it. However in the design of operating systems, kernel runs in the highest privilege ring, and this opposes the VMM design. For this reason it is not possible to virtualize a legacy OS with a type I VMM, as they both need to run in ring 0. To solve this problem Paravirtualization was used for type I VMMs. In paravirtualization guest OS is modified so that instead of working with hardware directly, the OS uses hypervisor calls to VMM[5]. In this approach OS is aware that it is going to be run in a virtualized environment. Usually OS is modified to make kernel runs in ring 1, and user space applications runs in ring 3. This design lets VMM to run in ring 0 and controls guest OS[5]. This is shown in Figure 2.4. This design has good performance, but the only problem is about OS modification. Modifying an OS is very complicated, and not source code of every OS is available.

2.1.3 Native Virtualization

Fortunately by introduction of virtualization technologies in CPUs, virtualizing an unmodified OS has become possible. Intel introduced Intel VT-x and AMD introduced AMD-sv. Other processor manufactures added the same technology into their CPU architectures. In Fact, they designed two security modes into their CPUs architecture, named root and non-root modes. This is shown

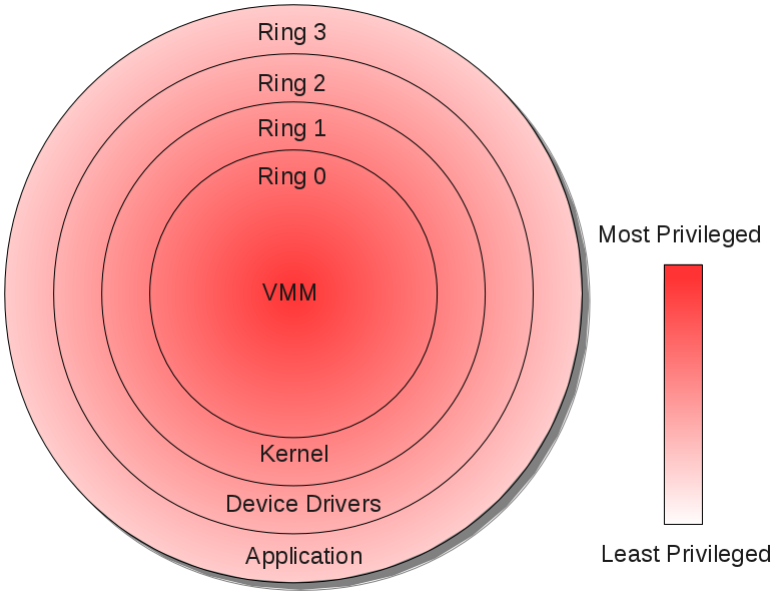

Figure 2.4: Paravirtualization

in Figure 2.5. Using this architecture, unmodified OS runs in the ring 0 of the non-root mode. To virtualize an unmodified OS, VMM has to run in root-mode, which has higher privileges than non-root mode[7]. This allows VMM to control the guest VM kernel. Executing instructions which need higher privileges like accessing hardware causes CPU to generate an exception interrupt, which triggers VMM to take control. VMM then decides how to handle the situation and makes guest OS believe it is controlling hardware. The transition from the non-root mode to the root mode is called VM EXIT, while the transition from the root mode to the non-root mode is called VM Entry[8].

There are many example of type I and type II VMM. Xen, VMware ESX, and Microsoft Hyper-V are type I VMMs, and KVM, VirtualBox, and VMware workstation are examples of type II VMMs. In this project, Xen is chosen as it is a fast open source type I hypervisor.

2.1.4 Xen

Xen[1] is an open source type I hypervisor. It supports both Para-virtualization and native virtualization. In Xen terminology each guest VM is called "domain". There is a special domain called Dom0. Dom0 is a trusted domain and paravirtualized[9]. Using special hypercall named dom0_op, dom0 can manage

[1] http://www.xen.org/

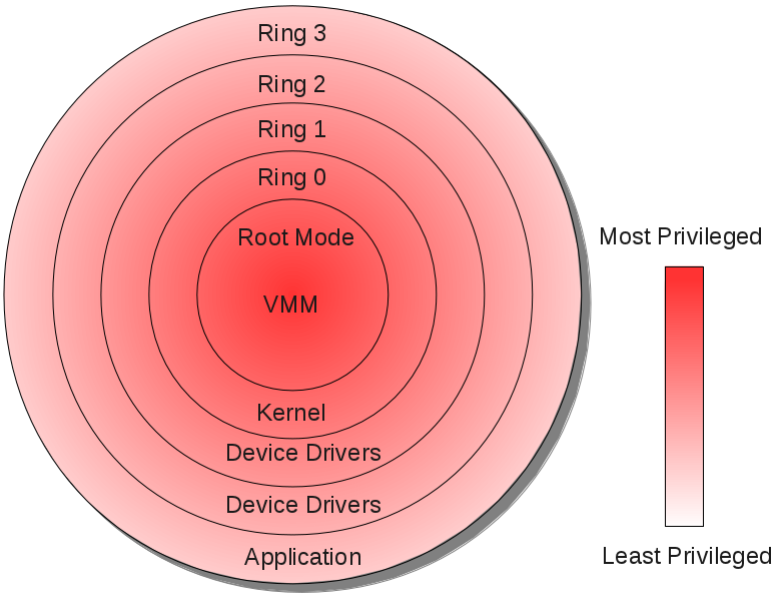

Figure 2.5: Native virtualization

Xen VMM and other virtual machines. For example dom0 can request Xen to map a guest VM memory pages into its own memory address. Other ordinary VMs are called DomU.

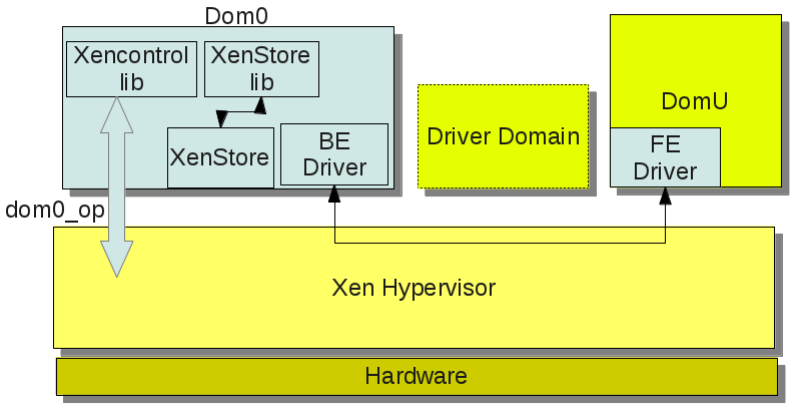

Figure 2.6: Xen architecture

There are different libraries in Dom0 for different purposes. XenControl library allows Dom0 applications to interact with Xen through dom0_op hyper call. In addition, there is a central database, named XenStore which contain a list of guest VMs and their information. Guest VMs can access XenStore through XenBus and Dom0 access it using the XenStore library. It worth mentioning Xen VMM is a very thin layer of software which does not contain hardware drivers. Xen uses drives in the Dom0 kernel to control hardware. It is even possible to delegate the driver role to another DomU which is called the driver domain. Xen introduces a set of generic and widely used hardware to each DomU. OS in a DomU installs drivers, called Front End drivers (FE drivers), for available hardware. Using this method, the FE drivers can communicate with the real hardware in Dom0 by available drivers in Dom0, called Back End (BE) drivers[10]. This is depicted in Figure 2.6. In this scheme DomU believes that it is controlling hardware while VMM can control DomU access.

2.2 Virtual Machine Introspection

Virtual Machine Introspection (VMI) is the act of observing state of a VM from an external entity that can be either VMM or another guest VM[11]. In Xen terminology the external entity is either Xen VMM or Dom0. In addition it is important to define what the state of a VM is. State of a VM is a set of information about internal structure of a VM that can help to understand what is happening inside that VM. This includes content of CPU registers, volatile and nonvolatile memory and I/O data. To gain knowledge about a guest VM it is possible either to use all internal information of that VM or just only important information[12]. Here, the tradeoff is between cost and complexity of retrieving required information. While it may seem better to find as much information as possible about a guest VM, in practice this needs a large amount of time and CPU cycle. Therefore the trend is to retrieve only required information for a specific application. If a designer wants to find the list of installed modules in the kernel, he/she does not need to find information about hard disk content.

In addition, there are two different approaches toward VMI. It is possible either to passively monitor a guest VM or actively control it[9]. Passive systems are usually polling based systems that periodically check the guest VM. However, active systems use event triggered methods to intervene in the guest VM normal behavior. While passive monitoring is helpful in detecting intrusion, active monitoring helps in preventing intrusion. Although the idea of active monitoring is attractive, it is not a trivial task. Not only information has to be retrieved from the guest VM, but also a method has to be found for controlling its behavior. As these tasks are non-trivial, active monitoring methods are rarely used in available libraries and solutions.

In the next sections some limitations and facts about VMI are described.

2.2.1 Switching Performance

As mentioned earlier, transitions from the root mode to the non-root mode, and vice versa are called VM Exit and VM Entry respectively. These transitions are complex operations compared to normal system operations. For this reason they are considered CPU intensive functions. If VMI application is located in VMM, then 2 transitions are needed for VMI: One VM exit from DomU to VMM, and a VM Entry from VMM to DomU. However, if VMI application is located in the Dom0 then 4 transitions are required: a VM Exit from DomU to VMM, a VM Entry from VMM to Dom0, then a VM Exit from Dom0 to VMM, and finally a VM Entry from VMM to DomU.[13]. This process is shown in Figure 2.7.

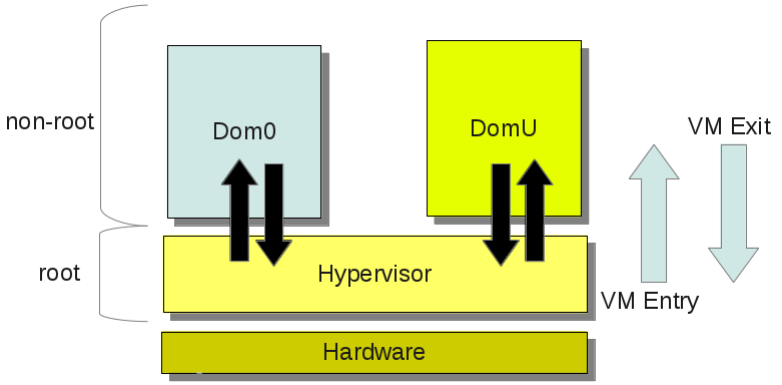

Figure 2.7: context switching

While locating VMI into VMM seems to be a better solution, it makes VMM more complicated. Best practice for VMM design is to implement it as small as possible so that the chance of finding vulnerability in VMM reduces. For this purpose almost all VMI libraries and applications, except a few have implemented VMI in Dom0. However, since VMI in Dom0 needs four transitions, it is not proper for high-rate applications as it would suffer from poor performance. As a result, in all VMI libraries and applications studied in this thesis, high-rate solutions are implemented in VMM and low-rate solutions are implemented in Dom0.

2.2.2 Semantic Gap

Semantic gap is the difference between the external viewpoint about the internal aspect of a guest VM and what actually happens inside it[11]. Whatever exists in a guest VM are all meaningful, data structure, variables, and etc. But from VMM or Dom0 point of view, all those meaningful information are just a bunch of meaningless bits. This means that there is a gap between internal view of a

VM and view of an outside entity about that VM; this gap is called semantic gap, as shown in Figure 2.8.

VMI application should somehow bridge this semantic gap, to be able to retrieve required information. Basically a VMI application can be either semantically aware or semantically unaware[9]. A semantically aware application has built-in information about the target, like memory addresses for specific information or related data structures. However VMI application should have a list of this information for different OSs, OS versions and architectures. On other hand there are unaware VMI applications which do not have initial information about internal data structure but built it overtime. Figure 2.9[9] shows an example of bridging the semantic gap.

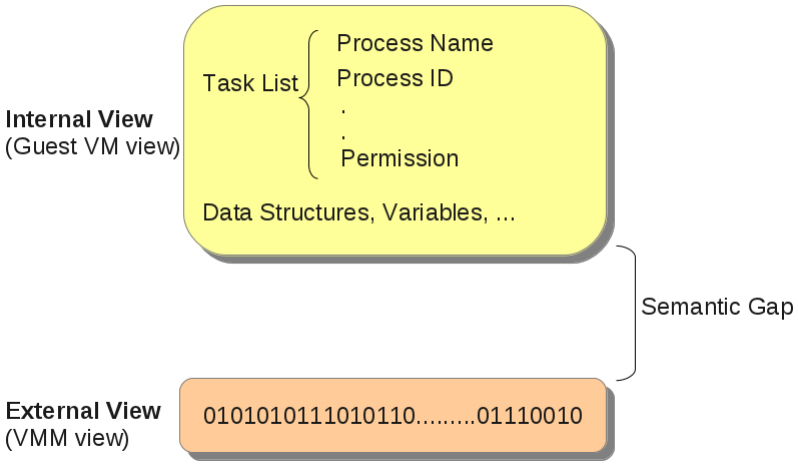

Figure 2.8: semantic gap

As an example, Linux systems use a linked list to keep information of running processes. By knowing the kernel version, and the address of linked list head, one can find its location; then by parsing the linked list the list of running processes can be retrieved. "system.map" is a file which contains the address of symbols in the Linux kernel. Therefore, by searching through that file one can find address of "init_task", the head of mentioned linked list[14][9]. "system.map" is an important file for introspecting Linux virtual machines.

2.2.3 VMM and VMI Detection

In a bare metal computer, if a malware compromises the kernel, it gains the highest privileges. Therefore, it can perform any task while hiding itself from the kernel or any kind of security function. But in a virtualized environment,

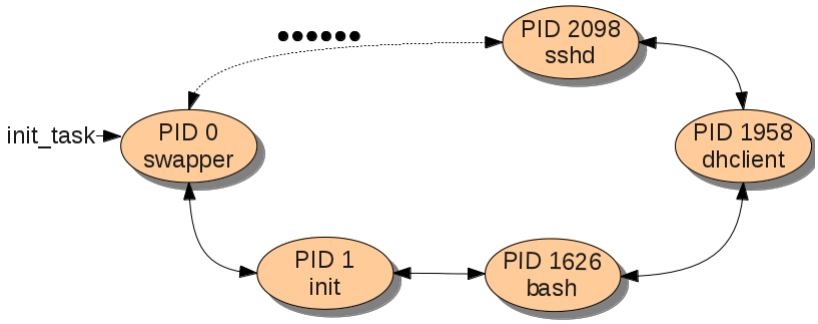

Figure 2.9: semantic gap example

VMM is the highest privileged entity. For this reason it is hard for a malware in a guest VM to hide its existence from VMM. In addition, it cannot perform any arbitrary task. Nowadays, a malware having compromised the kernel tries to find out if it is running in a bare metal computer or a guest VM; so that it can possibly escalates its privileges by compromising VMM or hiding itself from VMM.

The first important question is, whether it is possible for a guest VM to determine VMM existence. And it turns out that in all major virtualization solutions a guest VM can detect VMM. As a matter of fact, for cooperative virtualization between VM and VMM which result in improved performance, hypervisors are intentionally designed to be detectable[14]. There are different ways to detect VMM. Usually a VMM exports virtual hardware, like virtual buses for communications. However, there are other ways for VMM detection. Using timing analysis, a process can find anomalies in execution frequencies to conclude VMM existence.[14, 15]. In addition, the anomaly in page fault analysis can result in VMM detection. In this method, if a process finds out that a memory page is swapped out while it was supposed to be in memory, it can assume VMM exists[14].

The next important question is, whether it is possible to detect VMI security function. Even though this question has not yet been answered, if a malware detects VMM then it can assume a VMI security function exists as well and then attempts to hide its existence[14]. At the time of writing this report, there has not been any malware that can reliably hide itself from VMM or VMI security function[14]. This is the result of the fact that VMM has higher privileges, and has the ability to observe whatever a malware can possibly perform in a guest VM.

2.2.4 Exploits

In a virtualized environment, VMM has full access to all resources, and has the highest privileges. Therefore it is obvious that a compromised VMM leads to compromised VMs. This is the reason why securing VMM is very important. For example, there has been exploits for Xen, like CVE-2007-4993 and CVE-2011-1898 which let a guest VM run arbitrary commands in Dom0 privileges. So it is vital to secure VMM as much as possible. A good practice for VMM is to keep its code as small as possible, to reduce the probability of finding its vulnerabilities[11].

In addition, Dom0 is a special and trusted domain in Xen environment. Dom0 has the ability to access DomUs resources, like memory addresses. Therefore, like VMM, a compromised Dom0 leads to compromised DomUs. All in all the security of Dom0 is as important as security of VMM.

Since the VMI security function is executed with the highest privileges in either VMM or Dom0, it is very important to keep it secure; because vulnerability in VMI effectively leads to whole system compromise.

2.2.5 Methods

In this section, different methods that are available for implementing VMI are described.

2.2.5.1 VM State Access

What VMM provides for VMI is the access to the guest VM state[11]. A VM state is defined by its CPU registers, memory space and I/O access of that VM. However VMM can only provide low level access to the VM state, so VMI basically only can observe a set of bits using VMM. A VMI application should provide a layer of knowledge and intelligence about guest OS to gain the ability of translating low level view of guest VM into meaningful information. As an example a VMI should have knowledge about windows XP internal structure to be able to check its validity. For VMI, it is not necessary to have full knowledge of the guest OS, but only helpful information like IDTR[2], system call, virtual memory I/O. A VMI application can either poll a VM periodically or rely on an event to check a VM state[12]. VM state access is only a method for passive VMI, and compromise detection. Even though accessing a VM state is needed for active introspection, there is no way to actively introspect a VM using only this method. There are other methods that are used for active monitoring, but they all need VM state access as a basis for their functions.

2.2.5.2 Guest OS Hooks

[2]Interrupt Descriptor Table Register (IDTR) is a special register in x86 architecture that points to interrupt vector table for handling interrupts.

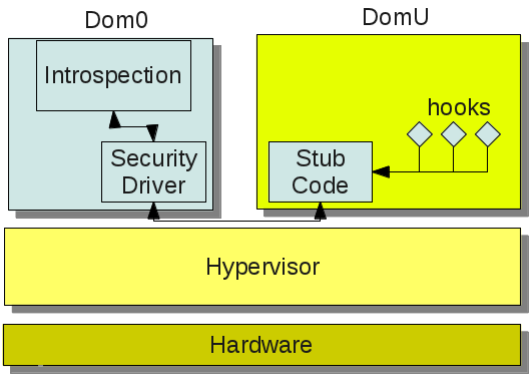

Figure 2.10: hooks

This method is more intrusive, since it needs guest OS modification. In this methods some hooks are inserted into guest OS. They send back required information and events to the security driver in Dom0 through a small code named stub code in DomU. This is shown in Figure 2.10. Usually hooks are kernel modules. This method is used for active monitoring and preventing compromises.

2.2.5.3 Interrupts

In all computer architectures, there are interrupts that facilitates VMI. Examples of those interrupts are context switching interrupts which are available in native virtualization[12]. However, some general system interrupts, such as page-faults; invalid opcode can be useful as well. One interesting interrupt is the debugging exception interrupt, which is a built-in debugging mechanism in modern processors. Breakpoints used in this debugging scheme are called hardware breakpoint in contrast to software breakpoints that will be explained in section 2.2.5.4. Using this interrupts processor generate debug exception when instruction pointer(IP) register hit addresses defined in debug address registers. This is very useful but usually this mechanism is limited to small number of breakpoints, as it is based on registers in CPU. x86 architecture has only 4 debug address registers for this purpose[12].

2.2.5.4 Kernel Debugging

Kernel debugging is like debugging exceptions in CPU architecture, except that instead of being implemented in CPU it is a part of VMM. Kernel debugging basically works by inserting breakpoint opcode in arbitrary memory addresses. This type of breakpoint is called software breakpoint. When processor executes this opcode, it generates a debugging exception interrupt, and then executes its interrupts handler[8]. This Interrupt leads to SIGTRAP signal in the Linux

system which is caught by a debugger. In contrast to debug exception and hardware breakpoints, more functionality can be provided in this method. In addition the number of breakpoints is not limited. These benefits are the results of the fact that this method is software centric. Xen has a built-in mechanism for kernel debugging, which will be used in this thesis[10].

2.2.6 Solutions

In this section a survey of some available VMI libraries and applications are provided.

2.2.6.1 XenProbe

XenProbe[13] is a VMI library for Xen 3.x. However, it is not updated any more. XenProbe works in Dom0 user space, and was developed to be used for active monitoring. This library uses "system.map" file to bridge the semantic gap, and uses kernel debugging techniques by replacing original opcode with breakpoint opcode. XenProbe stores the original opcode in a memory space called out-of-line execution area (OEA). To handle OEA and original opcode XenProbe needs a kernel module in guest OS named XenProbesU.

2.2.6.2 XenAccess

XenAccess[16] is a VMI library that provides passive monitoring. It has the capability of monitoring guest VM memory states and disk activities. Monitoring memory states is based on mapping guest VM memory to monitoring VM address space. In addition it provides a layer of intelligence to bridge the semantic gap. Up to present moment, at the time of writing this report, only 32-bit guest OS is supported. Monitoring disk activity is based on intercepting disk I/O through the blktap driver. XenAccess monitor disk activity by modifying blktap BE driver in dom0. Figure 2.11[16] shows structure of XenAccess. Unlike some other monitoring libraries, XenAccess does not change guest OS nor install any software or modules in guest VM. XenAccess is under active development, and it is compatible with Xen 3.x to Xen 4.0. As there are some major changes in Xen 4.1, at the time of writing this report, XenAccess cannot work with it. XenAccess is a well-known monitoring library, and it has been used in many monitoring applications. For example, Psyco-virt[17], an intrusion detection tool, uses host and network intrusion detection techniques with VMI methods. Psyco-virt uses XenAccess to retrieve guest VM states, beside other methods it uses.

2.2.6.3 libvmi

Recently developers of XenAccess have released a new VMI library, called libvmi. It supports Xen 3.x to 4.1, and KVM. libvmi is an attempt to develop a general passive monitoring VMI library that works with different virtualization

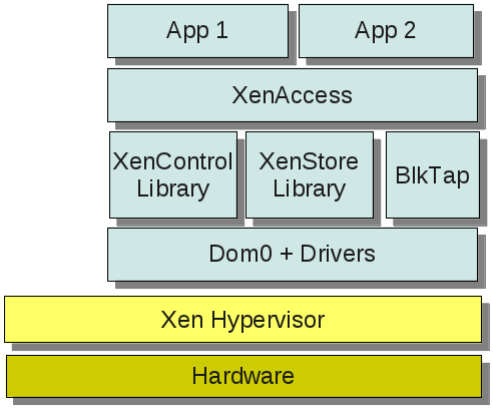

Figure 2.11: XenAccess

solutions[18]. Since libvmi recently has been released, at the time of writing this report, there is not so much information available about it.

2.2.6.4 Lares

Lares[19] is a VMI application, from XenAccess developers, and it was developed to show how to implement active monitoring systems. Larse install hooks, which are kernel modules, inside guest VM to actively monitor a guest VM.

2.2.6.5 Ether

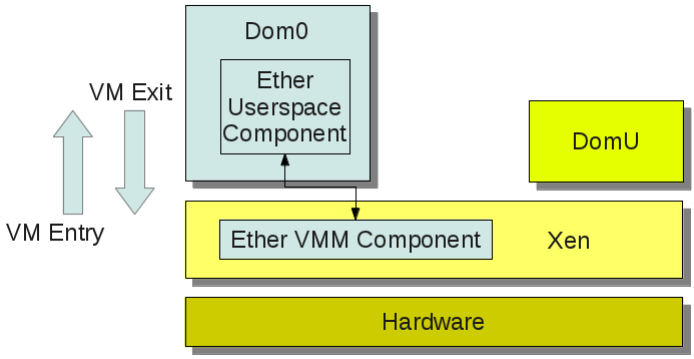

Figure 2.12: Ether

Unlike other VMI libraries, Ether[15] takes a new approach for VMI. It uses two components, one in VMM and the other in Dom0. Ether is an active VMI library and it is very context switching intensive. Ether monitors system calls, memory writes, context switching and other events in a guest VM. For this reason it is placed in VMM to reduce the number of required context switching which improves its performance. While the Ether component in VMM is responsible for VMI, the user space component is responsible for management and bridging semantic gap. Structure of Ether is depicted in Figure 2.12[15]. In addition, Ether needs guest modifications. This library is not active anymore, and has never been updated[15].

2.2.6.6 VIX

This tool suite was developed mainly for digital forensics investigation[14]. Currently digital forensics methods are based on offline examination of a system, which lack information from memory or other volatile information. VIX provides a suite of tools to passively check and examine a live system. Therefore it can access target system memory beside other information to find system intrusions[14].

2.2.6.7 gdbsx

gdbsx is not a VMI library, but an embedded gdb server in Xen VMM[10]. Its main purpose is for kernel debugging; however it can also be used for VMI. gdbsx is basically a gdb server, which can communicate with gdb[10]. In fact, gdb server is a small stub code, designed for embedded systems with limited resources that are unable to run gdb by themselves. Communication between gdb and gdbsx is shown in Figure 2.13. gdbsx is an implementation of gdb server in Xen, and it is only accessible from Dom0. It communicates with gdb using a plain text protocol called Remote Serial Protocol (RSP) over serial or network devices[10].

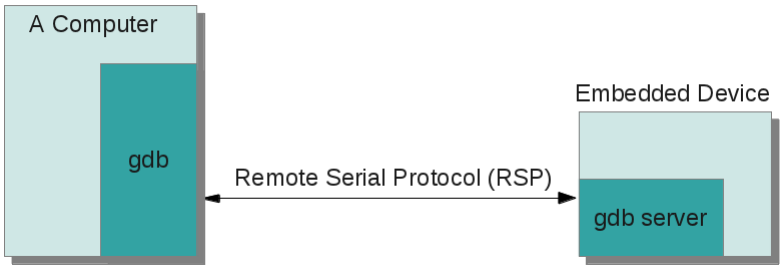

Figure 2.13: gdb server

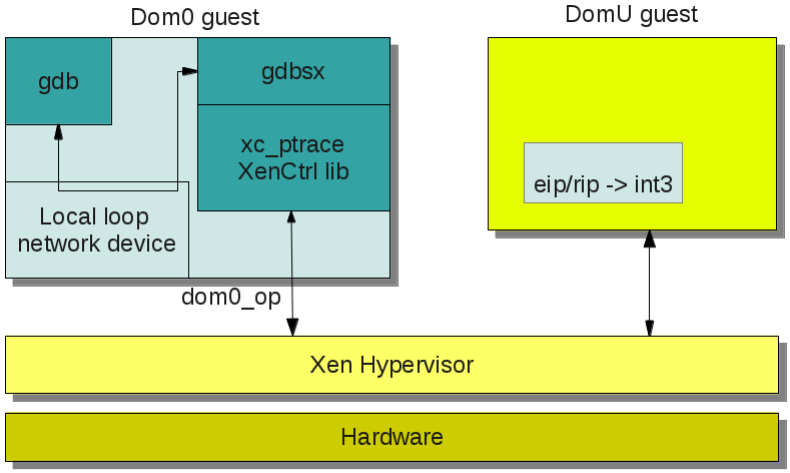

Figure 2.14: Kernel debugging by gdbsx

gdbsx uses "xc_ptrace" function from XenControl library, described in section 2.1.4, to insert software breakpoints into guest VM memory address space[8]. When CPU hits those breakpoints, VMM pauses the target VM and gives control to gdb to debug the target guest OS. This is shown in Figure 2.14[8].

To automate the debugging process, gdb provides three different interfaces[20].

1. Libgdb: It is not a complete and active library, mostly an idea for extending gdb. Up to now two versions of this library has been published. But since this library has been changed in each version of gdb, and also there is no documentation, it has never been used for extending gdb functionalities.

2. Machine Interpreter (MI): It is a text interface to gdb, and is designed for development of systems that use debugger as a small part of them.

3. Python: Since gdb version 7, it is possible to write python scripts, taking advantage of a gdb module, and then ask gdb to execute it. Using this new capability in gdb, one can extend gdb functionalities. This method is newer than MI so it is not widely used like MI; but it has good documentation.

2.2.7 VMI Libraries and Applications Summary

Table 2.1 compares different VMI libraries. Among solutions listed in Table 2.1 only XenAccess, libvmi, and gdbsx are being updated.

	Monitoring	**Method**	VM Modificitaion	**Platform**
XenProbe	Active	Exceptions	Yes	Xen 3
XenAccess	Passive	State Access	No	up to Xen 4.1
libvmi	Passive	State Access	No	Xen 4.1, KVM
Ether	Active	Exceptions	Yes	Xen 3.1
VIX	Passive	State Access	No	N/A
gdbsx	Active	Kernel Debugging	No	Xen

Table 2.1: VMI Libraries

2.3 Filesystem Integrity

Integrity, keeping a system in a known state, is a property of security, and to secure a system, it is vital to satisfy this requirement. For Linux systems solutions have been deployed to address integrity requirements, like tripwire. But they are usually passive solutions that cannot prevent violations. Nevertheless they are user space or kernel space solutions which are ineffective against kernel exploits. The goal of this thesis is to implement a security function for Linux systems using VMI to satisfy this security property. Two available security functions for integrity requirements in Linux systems named DigSig, and IMA are studied. In the following sections a brief description of them is covered.

2.3.1 DigSig

DigSig[2] is an attempt to ensure Linux systems integrity in kernel space. Unlike tripwire and similar solutions, which are user space applications, DigSig is a Linux kernel module. It verifies the RSA signatures stored in the header of ELF executable files. Filesystem labeling is needed before using DigSig. Generally, this is a procedure to add an attribute to files on filesystem. Bsign, a tool developed for this purpose, is used to add the signature segment to ELF headers. Then DigSig uses LSM hooks like "mmap" hooks to control "sys_execve" system call which executes binaries. Although DigSig is an effective solution, it has some limitations. First it only checks ELF binaries, so it cannot check scripts or other important files. Second, kernel rootkits have been developed, and as a result kernel space solutions like DigSig are no longer secure enough against kernel exploits[2].

2.3.2 Integrity Measurement Architecture (IMA)

Integrity Measurement Architecture (IMA)[4] is developed to protect Linux systems against accidental and malicious filesystem modifications. Like DigSig, IMA is a kernel space solution, however as it uses Trusted Platform Module (TPM)[3], IMA can be more secure than DigSig. Another advantage of IMA is that, it does not depend on the ELF header. So it can verify integrity of any

[3] http://www.trustedcomputinggroup.org/

arbitrary file. IMA has some functionality for verifying integrity of a Linux system remotely or locally. Those functions are[4]:

1. Collect: this module calculates a hash value for each file.
2. Store: this module stores collected hash values in a run time table located in the kernel.
3. Attest: this module allows remote verification.
4. Appraise: which prevents access to a maliciously altered file.
5. Protect: which protects filesystem against offline attacks.

Module	**Status**
IMA	Accepted in kernel 2.6.30
EVM	Accepted in kernel 3.2
IMA-Appraisal	Posted
IMA-Appraisal-Directory	N/A
IMA-Appraisal-Digital-Signature	Posted
EVM-Digital-Signature	Posted

Table 2.2: IMA functions

IMA has several different modules; some of them are now available in the Linux kernel mainline; some of them have been posted to be part of the mainline kernel and some parts are under development. Table 2.2 lists IMA modules and their status at the time of writing this report.

2.3.2.1 IMA

The first three functionalities of IMA (collect, store, and attest) are in the mainline kernel since version 2.6.30. These functions are called just IMA. IMA is responsible for computing the hash of files solely based on their content; then it stores computed hash values in a run time measurement list which acts as a cache. IMA uses the inode version of files to detect modifications. It calculates new hash for modified files. Therefore, IMA works on filesystems mounted by "i_version" flag.

2.3.2.2 IMA-Appraisal

While base modules of IMA only compute and store hashes, IMA-Appraisal[4] is the appraise functionality of IMA. It is responsible for local integrity validation and enforcement. It adds an extended attribute to file inode named "security.ima", which contains the valid hash digest of files. IMA-Appraisal compares the current hash of files with the good hash digest stored in "security.ima". If

they do not match which denotes a malicious modification, IMA-Appraisal prevents file access. However since it is based on hash values, IMA-Appraisal is vulnerable to offline attacks; an attacker can modify a file and then update its corresponding good hash digest stored in the file extended attribute. There is another IMA module named "ima-appraisal-digital-signature" that uses digital signature instead of hashes. At the time of writing this report this module is not posted yet. IMA-Appraisal is posted to the kernel mailing list, and it supposed to be accepted in Linux mainline kernel in near future.

2.3.2.3 EVM

Extended Verification Module (EVM)[4] is another IMA module. This module provides a method to detect offline attacks against files security extended attributes, like "security.selinux", "security.SMACK64", and "security.ima". EVM computes a HMAC using encrypted or trusted keys over those security extended attributes. The computed HMAC is stored in another extended security attribute, "security.evm", in the file inode. Trusted keys are based on TPM but encrypted keys are user defined keys, which are less secure than trusted keys. EVM is now part of the Linux mainline kernel since version 3.2.

Chapter 3

Design and Implementation

In this chapter, design and implementation of the security functions are presented. First specifications and goals of these security functions are described; then design and implementation detail of each security function is given. Finally, it is explained how these two security functions are started early in the boot process of a guest VM.

3.1 Specification

The goal of this project mainly is to design a system that verifies file signatures when they are loaded into a guest VM memory. When a file is accessed, its signature is verified by a public key that is associated to the private key used for initial signing. Xen is chosen as the virtualization environment, because it is a fast open source virtualization solution. Xen already has the VMI tool needed for this project. The signature verification is performed in a trusted domain that is Dom0 in Xen. VMI application runs in Dom0 which has special privileges to access other guest VMs, DomUs.

Kernel debugging is the method used for VMI, and it is already incorporated in the Xen VMM. Figure 3.1 shows the overall system design. The VMI application uses a python program to connect to gdbsx, the gdb server implemented in Xen. This program performs the security function. gdbsx provide facility for VMI application to insert breakpoints in any arbitrary guest VM memory addresses, as described in section 2.2.6.7. Whenever CPU executes a breakpoint in the guest VM, it generates a breakpoint exception interrupt, and executes the related interrupt handler. As Xen controls this interrupt, it finds out about the breakpoint hit in the guest VM. As a consequence, Xen pauses the guest VM, and gives control to Dom0 which in turn generates a SIGTRAP signal. This signal is handled by a debugger, the security function.

By means of VMI, Dom0 is notified whenever a file is accessed in the target DomU; then in Dom0, VMI application performs the security function by verifying the file signature using the proper public key. XenStore is used to retrieve domain ID of a guest VM, which is needed for connecting to gdbsx.

As the nature of the chosen security function is intervening in the behavior of a guest VM, an active monitoring method is needed for this project. As a result, unlike passive monitoring not only information has to be retrieved from guest VM but also a method has to be found to control its behavior, i.e. policy enforcement. In my opinion there are two different viewpoints for enforcing policy to guest VMs:

1. External enforcement: enforcement occurs externally like, pausing, or stopping VM.
2. Internal enforcement: in which the security function forces a guest OS to perform a task, like by changing permissions.

External enforcement is usually used for passive monitoring, and internal enforcement is the proper method for active VMI. The fact that active monitoring and internal enforcement are non-trivial to implement, made them rarely used in available libraries and solutions, as explained in section 2.2.7.

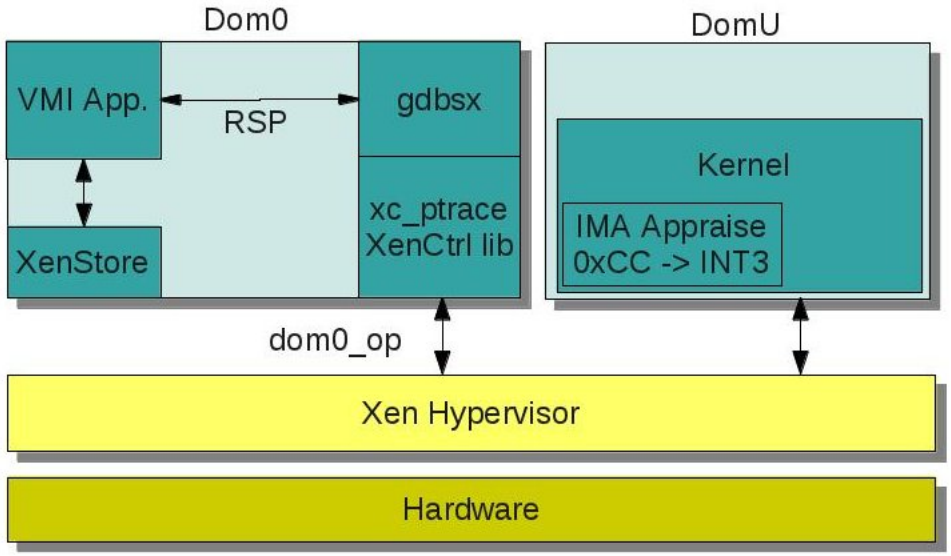

Figure 3.1: general schematic of security functions

To enforce internally, denying file access, either a new module has to be installed or an available mechanism in the Linux kernel has to be used. In this thesis,

IMA-Appraisal is the mechanism controlled in a VM to enforce proper policies. IMA-Appraisal stores the 160-bit SHA1 digest of files in the extended attribute of a file, "security.ima". In contrast to storing good digest values centrally in a database, saving good SHA1 digests in the file inode removes the search delay. This improves performance of the security functions. IMA-Appraisal is used to store valid crypto values in files extended attributes and to enforce internally in a VM. In addition, IMA implements an internal cache, run time table, which improves performance.

While IMA and gdbsx both are available tools, the novelty in this approach is designing a security function that takes advantage of them. The problem here is to find a way to combine IMA and gdbsx capabilities and to implement a security function which is signature verification of loaded files in the guest VM.

To improve the security attributes of the system, VMI application possesses only the public key, and does not have the private key. The private key is only used to prepare the guest VM image on a trusted machine. In this procedure, the private key is used to sign each file on filesystem. This fact has impact on the system design. While only the public key is needed for signature verification, the private key is used for signing. Essentially whenever a file is modified its signature has to be updated as well. The set of files the security function will check depends on the security policy and can be configured. Therefore it could be the case that verification only happens for important files, files owned by root, which are expected to be immutable.

However, regarding this limitation, i.e. updating the file signature, one more security function is designed. This second security function uses only one secret key to verify the HMAC-SHA1 digest of files. HMAC-SHA1 is chosen, since its output digest is the same size as SHA1, which is 160-bit. Since these two crypto functions generate 160-bit digest, no kernel modification is required. While in the RSA signature verification, the Linux kernel should be modified to be able to work with 4096-bit RSA signatures. In this report the RSA verification design is named VMI-RSA and the HMAC-SHA1 validation design is named VMI-HMAC respectively. Here is a brief description of these two security functions:

1. VMI-HMAC: A low-rate context switching security function which validates HMAC-SHA1 digests of files, in a guest VM when they are loaded in memory. The security function runs in Dom0, and uses a secret key.

2. VMI-RSA: A low-rate context switching security function which verifies the RSA signature of files when they are loaded in memory of the guest VM. Verification occurs using a proper public key in Dom0. Dom0 and the applied security function do not possess the private key used for initial signing.

VMI-HMAC and VMI-RSA security functions are described in more detail in sections3.3 and 3.4 respectively.

3.2 VMI

In these VMI systems kernel debugging is used for virtual machine introspection. The general idea is to use gdb to connect to gdbsx and debug the guest VM. But breakpoint interrupt handling process should be automated for the security functions. There are 3 ways to automate, and to control gdb, as they are described in section 2.2.6.7, using libgdb, Machine Interpreter (MI) and python extension. Python extension is used in this thesis, since it allows us to take advantage of python libraries. The following example shows how it is possible to use gdb python extension[20].

To bridge the semantic gap, some initial information is provided to gdb. Primarily, the architecture of guest VM has to be explicitly specified; whether it is a 32-bit OS or 64-bit OS. On the other hand to allow gdb to access variables and structures in the guest VM kernel space, gdb needs to access the compiled directory of the kernel. This allows gdb to know about exact structure of that kernel. Finally, the unzipped version of the kernel file has to be available to the security functions. This file contains symbol information of the kernel, and matches information in the "system.map" file. All these information is given to gdb to let it access the guest VM kernels freely, as you can see in the following code[20].

```
# !/usr/bin/gdb -P

import gdb

# architecture of guest VM
gdb.execute ("set architecture i386:x86-64:intel")

# path to the directory of compiled kernel
gdb.execute("directory kernel_compiled_directory")

# path to the kernel symbol file - unzipped kernel file
gdb.execute ("symbol-file path_to_symbol_file")

# connect to gdbsx
gdb.execute ("target remote 127.0.0.1:localport")

# insert breakpoint
gdb.execute ("break breakpoint_symbol")
```

Since VMI application inserts breakpoints in the kernel locations to control and handle file accesses, it is vital to remove those breakpoints before detaching from the guest VM. If the security functions detach from the guest VM before removing all breakpoints, a breakpoint hit leads to kernel panic. For a normal application, not having a SIGTRAP handler just leads to program termination, but in case of the kernel, the consequence is more severe, a kernel panic. To control this situation, all breakpoints in guest VM are removed before detaching from it. The following code is used for this purpose:

```
def cleanup():
        for br in gdb.breakpoints():
                br.delete()
```

3.3 VMI-HMAC Security Function

IMA-Appraisal extended attribute, "security.ima", is 160-bit, while the RSA signature, the goal of this project, is 4096-bit. For this reason it was decided to design and implement the VMI-HMAC security function that uses the original 160-bit, and does not change the length of "security.ima". In this model, VMI in Dom0 possesses a secret key and using that key, it verifies the integrity of files and updates their extended attributes.

Reading the IMA source, it was noticed that IMA uses the "ima_calc_hash" function for calculating the SHA1 digests. This function is only called by the "ima_collect_measurement" function. To verify the integrity of files, VMI-HMAC changes the IMA knowledge of its crypto function. VMI-HMAC intercepts "ima_calc_hash" function flow. It is possible to replace the calculated SHA1 digest by inserting a breakpoint in either where "ima_calc_hash" returns or where it is called. Function "ima_collect_measurement" is chosen for this purpose. This function calls "ima_calc_hash" function. Following code shows the structure of "ima_collect_measurement":

Security/integrity/ima/ima_api.c:

```
int ima_collect_measurement(iint, ...) {
        ...
        If (...) {
                ...
                // calculate SHA1 digest
                result = ima_calc_hash( ..., iint->
                    ima_xattr.digest );

                // Breakpoint location
                ...
        }
        ...
}
```

By inserting a breakpoint after where the "ima_calc_hash" function is called, the file hash digest, stored in "iint->ima_xattr.digest", can be replaced with HMAC-SHA1 digest. The security function computes the HMAC_SHA1 digest using the secret key. Either an offline breakpoint using "asm("INT3")" or a run time breakpoint using gdb works here. A discussion of different types of breakpoints is provided in section 3.6. When CPU executes a breakpoint it gives control to the breakpoints handler which is gdb/gdbsx, and then finally the following python program is executed:

VMI_HMAC.py:

```
# read SHA1 digest calculated  by IMA
Resp=gdb.execute("p/x iint->ima_xattr.digest",...)
...
# calculate HMAC-SHA1 using secret key
HmacObj = hmac.new ( Key, Resp, hashlib.sha1)
...
# replace SHA1 digest with HMAC-SHA1 digest
gdb.execute ("set iint->ima_xattr.digest="+HmacObj.
    hexdigest(), ... )
...
```

In this code, VMI application first reads the SHA1 digest computed by IMA. It computes HMAC value based on the secret key and the SHA1 digest. Finally

VMI replaces the SHA1 digest with the computed HMAC-SHA1 digest. It worth mentioning, some data processing steps are required between these instructions, since outputs cannot be used directly as input to the other functions. However, the general idea is still as mentioned above.

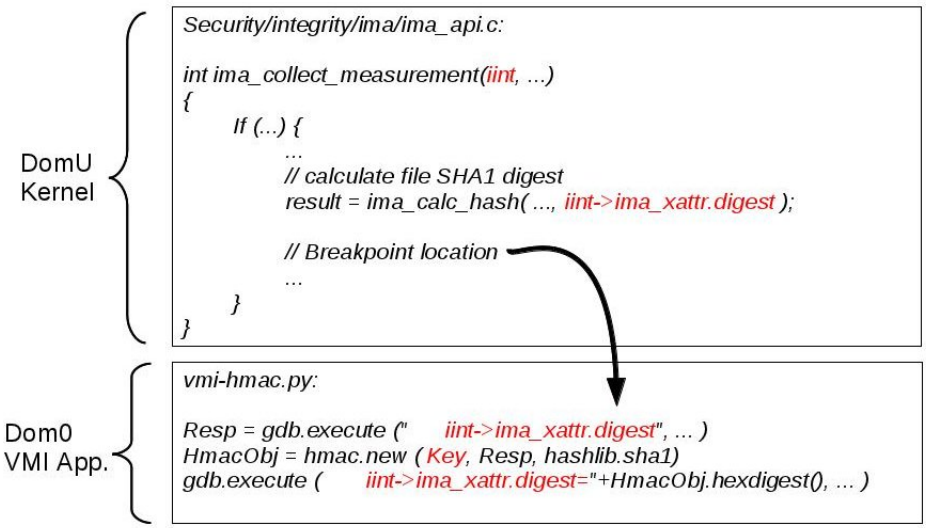

Figure 3.2: VMI-HMAC Security Function

IMA-Appraisal decides whether to allow or to deny file access based on the comparison of the current digest of files and the "security.ima" attribute. By intercepting IMA crypto function, VMI-HMAC changes IMA view of crypt function from SHA1 to HMAC-SHA1. Figure 3.2 shows generally how VMI-HMAC works.

To make this system work filesystem has to be labeled; so each file contains the correct HAMC-SHA1 computed by the secret key instead of the original SHA1 digest. Without having filesystem labeled, it is not possible to run guest VM in the IMA-Appraisal enforcement mode. However, as VMI-HMAC has intercepted IMA crypto function, it is possible to label filesystem using normal IMA labeling method, inside the guest VM. The method is to boot guest VM in the IMA-Appraisal fix mode, and then open all files as following:

1. Load kernel using "ima_appraise=fix" mode
2. Use the following command to label filesystem:
    - Find / -uid 0 -exec head -n 1 '{}' \;

In addition, a BASH script, "setxattr-hmac.sh", is developed to label the filesystem offline. To use it, guest VM filesystem needs to be mounted. The set of steps needed for labeling are:

1. Mount filesystem:

    - Mount <device> <path>

2. To label one specific file:

    - setxattr-hmac.sh <key> <filename>

3. To label filesystem:

    - Find <path> -uid 0 -exec setxattr-hmac.sh <key> '{}' \;

Having filesystem labeled, it is possible to run the guest VM in the IMA-Appraisal enforcement mode, taking advantage of VMI application in Dom0. VMI-HMAC is implemented, and it works correctly.

3.4 VMI-RSA Security Function

The major problem in implementing a security function that uses RSA signatures is the fact that IMA uses 160-bit hash digests. For this reason IMA internal variables and structures use 160-bit fields. To support the 4096-bit RSA signatures, some kernel modifications are needed. The following changes are made in IMA code to let it support the 4096-bit RSA signatures:

security/integrity/ima/ima.h:

```
#define IMA_DIGEST_SIZE          SHA1_DIGEST_SIZE
```

to

```
#define IMA_DIGEST_SIZE          512
```

security/integrity/integrity.h:

```
struct evm_ima_xattr_data {
        u8 type;
        u8 digest [SHA1_DIGEST_SIZE];
}
```

to

```
struct evm_ima_xattr_data {
        u8 type;
        u8 digest [512];
}
```

Having made these changes, IMA works fine with 4096-bit length fields. These changes do not affect IMA normal behavior since all variable and structure fields are initiated with 0.

As one of the design requirements of VMI-RSA, Dom0 and the security function should not possess the private key. Therefore it is not possible to update files extended security attributes. For updating a file's attribute the private key is needed. As a result it is not possible to update, or create new files that meet IMA criteria, by default files owned by root user. However, it is practical to verify the RSA signature stored in extended security attributes, by intercepting IMA-Appraisal behavior in "ima_appraise.c", and in "ima_appraise_measurement" function. The following code shows the structure of this function:

```
Security/integrity/ima/ima_appraise.c:

int ima_appraise_measurement( iint , file ,  ...) {
        ...
        //  get extended attribute, security.ima, from
            file inode
        inode->i_op->getxattr(..., file, xattr_value,
            ...);

        // Calculate current SHA1 digest of file
        status = evm_verifyxattr( file, ..., iint );

        ...
        // compare current SHA1 digest with extended
            attibute, security.ima
        rc = memcmp(xattr_value.digest, iint->ima_xattr.
            digest, IMA_DIGEST_SIZE);

        // breakpoint location

        If (rc) {
                // verification failed
                status = INTEGRITY_FAIL;
                ...
        } else {
                // verification passed
                status = INTEGRITY_PASS;
                ...
        }
        ...
}
```

This function first reads the security extended attribute of the file, and then computes the current SHA1 digest of the file, and finally compares those values. If they match, then the check is passed and IMA-Appraisal allows file access. Otherwise IMA-Appraisal denies file access.

Filesystem is labeled offline using a private key. In the filesystem labeling process "security.ima" attribute of a file is set to its RSA signature. RSA signatures are calculated based on the content of files using the private key. Having made changes to IMA to support the 4096-bit RSA signatures, IMA-Appraisal loads the signature from filesystem and compares it to the current digest of file. The result of comparison is definitely false. However if VMI-RSA intercepts IMA-

Appraisal at this point using a breakpoint, it can verify the signature using the correct public key and the SHA1 digest IMA provides; then VMI-RSA can change the result of comparison to any arbitrary value. In this code, variable "rc" stores the comparison result of current SHA1 digest, and the "security.ima" attribute. If VMI-RSA changes "rc" based on the result of the RSA signature verification, it can change IMA-Appraisal behavior. If the verification fails, VMI-RSA forces the IMA-Appraisal to deny file access. Otherwise, it forces the IMA-Appraisal to allow file access.

A breakpoint is inserted exactly after the "memcmp" call. This can happen by adding an offline breakpoint, using "asm("INT3")" or by using a run time breakpoint using gdb. No matter which type is used, at that point, a debug exception interrupt is generated which is handled by gdb/gdbsx. Finally the VMI-RSA security function is executed. The general code that used is as following code:

VMI–RSA.py:

```
# read current calculated hash digest
hash = gdb.execute ("p/x iint->ima_xattr.digest", ... )

# read signature stored in security.ima extended
    attribute
signature = gdb.execute ("x/128wx xattr_value.digest",
    ... )

if VerifySignaturel (signature, hash, publickey) == True:
        # status = INTEGRITY_PASS;
        resp=gdb.execute ("set rc=0", ... )
else:
        resp=gdb.execute ("set rc=1", ... )
        # status = INTEGRITY_FAIL;
```

Figure 3.3 depicts how generally this security function works.

This RSA signature verification system works fine. However before using it the filesystem has to be labeled so each file has its own signature in the "security.ima" extended attribute. The process of labeling the filesystem is like the VMI HMAC solution except that it is not possible to label using normal IMA-Appraisal method. The only possible method of labeling the filesystem is to label it offline, using the private key. A BASH script, setxattr-rsa.sh, is developed to perform this task.

1. Mount filesystem:

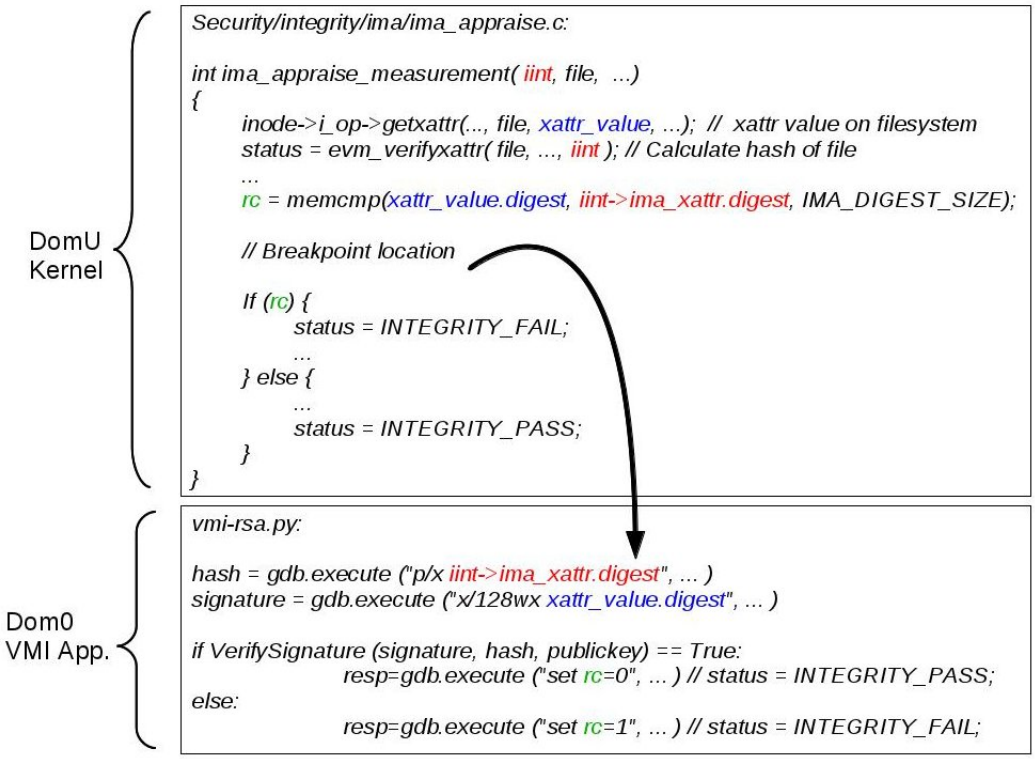

Figure 3.3: VMI-RSA Security Function

- Mount <device> <path>

2. To label one specific file:
   - setxattr-rsa.sh <private key> <filename>
3. To label filesystem:
   - Find <path> -uid 0 -exec setxattr-hmac.sh <private key> '{}' \;

Having filesystem labeled, guest VM can be run in the IMA-Appraisal enforcement mode using the VMI RSA security function.

3.5 Immutable Files Extension

As previously mentioned, by providing incorrect SHA1 digest to VMI application, attacker may compromise the kernel and fool the security functions. To address this flaw an extension is added to the security functions with regard

to immutable files. Some files in guest VM filesystem are immutable and are not supposed to be changed. For this reason the security functions have to retrieve the identical SHA1 digest from the guest VM kernel for the same file. Good SHA1 digests for immutable files are computed, and stored in a local file in Dom0. Whenever a file is accessed and checked by either VMI-HMAC or VMI-RSA solutions, the file name and absolute path is retrieved. Then by comparing the SHA1 digest provided by IMA with the good known SHA1 digest, it can be concluded if IMA in the guest VM kernel has been compromised. Since both VMI-HMAC and VMI-RSA work based on the SHA1 digest that IMA calculates, this extension works with both solutions. The pseudo code for this mechanism is:

```
Compute good digest of immutable files and store in <db>
Start guest VM
Run VMI application
Wait for interrupts
        Read <filename>
        Retreive <current digest> from IMA in guest VM
        Search <filename> in <db> and read <valid digest>
        If <current digest> != <valid digest>
                Pause guset VM
```

To find the filename and the absolute path, information in the stack is parsed when handling interrupts. Using gdb up to 10 functions in the stack is retrieved, and their arguments are parsed to find filename and file path.

It worth mentioning that, the security functions pause the guest VM when they find out that IMA in guest VM is compromised. This behavior is logical as if IMA is compromised VMI application cannot trust IMA, and using security functions is useless.

3.6 Boot Process

To increase the integrity protection of a guest VM, it is reasonable to start the security functions early in the boot process. This should happen as soon as possible so integrity of all files accessed in that guest VM can be checked. The method used in this project for VMI is kernel debugging, which is based on breakpoints. A solution should be figured out to insert breakpoints into proper locations in the kernel space as soon as the kernel is loaded into memory. The problem here arises from the fact that gdbsx uses software breakpoints, which are based on replacing an opcode located in the intended memory address with the breakpoint opcode. When CPU executes the breakpoint opcode, it generates a debugging exception interrupt, which is caught by debugger. So basically a

method has to be figured out to find when the kernel is loaded into memory by boot loader. Then breakpoints can be safely inserted into the kernel memory space. There are several ways to solve this problem:

1. Pending breakpoints[20]: gdb supports a set of breakpoints called pending breakpoints, which are basically breakpoints for addresses that are not loaded into memory. gdb handles this type of breakpoints by keeping their criteria, and whenever gdb loads a library, it checks if that library meets the pending breakpoint criteria. However this cannot be used for this project which debugs the Linux kernel, not a normal application.

2. Hardware (HW) breakpoints[20]: unlike software breakpoints, HW breakpoints use CPU registers to generate a debugging exception interrupt. Desired addresses are loaded into special CPU registers. Whenever Instruction Pointer (IP) register points to those addresses, CPU generates an INT3 interrupt. This interrupt is handled by the debugger. Using HW breakpoints, CPU debugging registers are set with proper addresses when guest VM is being created. This method eliminates the need of knowing when kernel is loaded into memory. Although this is a favorite solution, gdbsx has not yet supports HW breakpoints.

3. Boot loader introspection: As another solution, it is possible to first insert breakpoints into boot loader space, where the boot loader gives control to the kernel. At that point it is certain that kernel is in memory and breakpoints can be inserted. However, this method is boot loader dependent, which is not preferable.

4. Delayed VMI execution: By delaying execution of the security functions, the kernel may happen to be loaded in memory. An effort to insert a breakpoint in memory repeats until it succeeds. Delay time should be reasonable. It should not be too long so a part of the guest VM process passes without VMI. Nevertheless it should not be too short causing that the guest VM spends a lot of time in loop.

5. Offline breakpoints: debugging is based on INT3 interrupt. The source code of Linux kernel can be modified to generate this interrupt when IP is in the proper locations. Assembly INT3 instruction generates a debug exception interrupt. So if this instruction is added to the source code, the compiled kernel will generate this interrupt at proper locations. The only problem with this solution is the kernel modification. In addition when offline breakpoints are used, cautions should be taken to not panic kernel. If CPU executes INT3 instruction in the kernel while there is no interrupt handler, kernel crashes. This makes using modified kernel completely dependent on the security functions. However, for these systems, offline breakpoints are used in special way that does not affect normal kernel behavior at all.

In these systems an offline breakpoint is used carefully to eliminate most of its disadvantages. Instead of adding breakpoint instruction in favorite locations, it is only added in the function that IMA-Appraisal arguments are parsed. This makes it possible to load the kernel normally without the security functions if the "ima_appraise=" argument is not given to the kernel as an argument. But by giving "ima_appraise=" arguments, a SIGTRAP is generated, which is caught by VMI application. In this design, at this point software breakpoints are inserted into the kernel memory at the proper location to accommodate the security functions. This is depicted in Figure 3.4. Using offline breakpoints in all needed locations results in losing flexibility, and the modified kernel cannot be used independently at all when IMA is enabled.

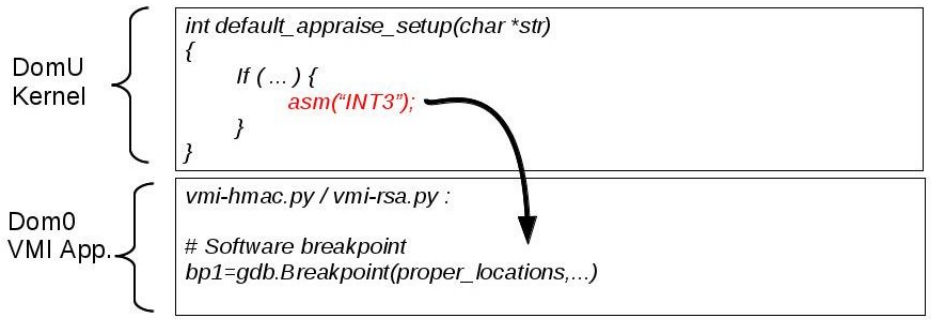

Figure 3.4: Boot Process Breakpoint

Having inserted breakpoints in the kernel, as soon as it is loaded into memory is not enough to ensure the integrity of a guest VM. Basically the boot process has to be checked completely. If not checked, an attacker can load another optional kernel without IMA-Appraisal and VMI. For this reason in these systems integrity of MBR and boot loader, GRUB in my configuration, are checked. To check MBR, the hash of the first 512 bytes of guest VM image is computed, and is kept in the VMI application configuration file. While creating the guest VM, its current MBR is compared against the good known value stored in the configuration file. This guarantee that MBR is not changed. In addition to verifying the integrity of the boot loader, the hash of all files in the "/boot" path including kernel files are compared with the good known values that are calculated in the trusted machine. It is assumed, all files in the "/boot" are immutable. They are not supposed to be changed by a normal user or even root. Any unexpected modified file makes the security functions to pause the guest VM.

Totally, this system ensures the integrity of a guest VM filesystem by verifying MBR and boot loader integrity, and by starting VMI as soon as the kernel is loaded in memory.

Chapter 4

Results

Having designed and implemented the desired security functions, they are analyzed in terms of performance, constraints and robustness against attacks. First, some possible attacks and solutions against them are explained. Performance analysis of the security functions comes in the next section. In the end, some limitations of these systems are listed.

4.1 Attacks

It is important to figure out the possibility of any attack against implemented security functions. Attacks which lead to VMI detection, like timing anomaly and page fault anomaly works here as well, as described in section 2.2.3. However, they are all general attacks, not specific to these systems. One attack that is specific to these security functions is based on the fact that IMA in the guest VM kernel is used. Having compromised the kernel, an attacker can disable IMA, which effectively disables these security functions. On the other hand, using the compromised kernel, an attacker can trick these systems by providing incorrect hash digests instead of the current updated ones. The possibility of disabling IMA in the kernel is not answered in this report. However to address the second possibility, providing incorrect hash digests using the compromised IMA to trick security functions, the immutable files extension is added to both the VMI-HMAC and the VMI-RSA designs. This extension is explained in section 3.5.

As some files are immutable, it is expected to receive the same hash digest from IMA for the same file. By comparing the digest provided by IMA and the good known digest, it can be concluded if IMA is working correctly. Although an attacker can provide the correct hash for those specific files, since it is not apparent to the attacker which files are being checked, he/she cannot predict it and consequently cannot trick this extension.

Another attacking point against these security functions is IMA policies. It is possible either to change IMA policies, or to change attributes of files, so that an important file does not meet the IMA policy. Setting IMA policy happens early in the boot process by the initram file. Therefore it is hard to compromise it. By default IMA policy is to check files owned by the root user, and this does not include special files like the ones in "proc" and "sys" filesystem. On the other hand, if an attacker gains root access in the guest VM then he/she can change the file owner to read or modify the content of files. However, this is not an effective attack, as privilege is reduced from root access to other user's access. One way to mitigate this attack is to change IMA policies to check all files and not just files owned by root. Even though, this increases system load, the guest VM is more secure. This attack point is not addressed in these systems, but by adding new functionalities it can be solved.

4.2 Performance

In this section the performance of the designed security functions is analyzed. A set of test cases is used to measure the performance of different system configurations, like bare metal, Dom0, DomU with or without IMA-Appraisal. Four tests are used in which a task is repeated 6 times. The task is the process of reading the first 10KB of all files in a defined set. Linux "dd" tool is used for reading 10KB from files. Description of used sets is listed below.

1. Set 1: A set of 100 10MB file with total size of 1GB
2. Set 2: A set of 20000 52KB files with total size of 1GB
3. Set 3: A set of 100 100MB files with total size of 10GB
4. Set 4: A set of 20000 520KB files with total size of 10GB

To consider IMA internal cache, the run time table, in this analysis each test repeats its task 6 times. If IMA-Appraisal is enabled the first iteration shows its effect in system performance. Other 5 iterations show normal behavior of the Linux system. Since IMA stores the hash digest of files in an internal run time table, it appraises a file only the first time that file is accessed. As long as the file content is not changed, IMA-Appraisal would not appraise it again. As a result, after the first iteration, execution time of later iterations is basically like in systems without IMA-Appraisal functionality. Except that there is an extra IMA run time table lookup step. To verify the result of tests, each test is repeated 3 times. In system configurations that IMA-Appraisal is enabled, for each three runs of tests, the system is restarted to flush out the IMA run time table. But in the system configurations without IMA-Appraisal functionality, 3 runs of tests are executed in a loop. The list of system configurations used for the system measurement is as the following:

1. Bare Metal: a system configuration which neither virtualization nor IMA-Appraisal is enabled.

2. Bare Metal-IMA: a bare metal configuration, but IMA-Appraisal is enabled. This configuration is used to measure impact of IMA-Appraisal in the bare metal system.

3. Dom0: Dom0 in the Xen environment without IMA-Appraisal functionality.

4. Dom0-IMA: Dom0 in the Xen environment with IMA-Appraisal enabled. This configuration is used to measure impact of IMA-Appraisal in Dom0.

5. DomU: DomU in the Xen environment without IMA-Appraisal functionality.

6. DomU-IMA: DomU in the Xen environment with IMA-Appraisal enabled. This configuration is used to measure impact of IMA-Appraisal in DomU.

7. DomU-BR: DomU in the Xen environment with IMA-Appraisal and VMI. But VMI only returns control to Dom0, and no special task is performed. This system configuration is used to measure impact of context switching.

8. DomU-HMAC: DomU in the Xen environment with IMA-Appraisal enabled. This configuration is used to measure the performance of the VMI-HMAC solution.

9. DomU-RSA: DomU in the Xen environment with IMA-Appraisal enabled. Performance of the VMI-RSA solution is measured in this configuration.

These tests are run on a computer with an Intel Core i7-2829QM, 2.30GHz processor, which has Intel VT support, 16GB memory size, and a 7200 RPM hard drive. The DomU used for testing is installed on a physical hard drive, and has 12GB memory capacity. Fedora 16, 64-bit, is used as OS for all system configurations, with default packages and setup. The Linux kernel used is 3.2.0-rc1 with IMA-Appraisal patched.

To measure the execution time of tests, NTP service is run on all system configurations. NTP servers are synchronized with a set of stratum 1 and 2 NTP servers. Precision of NTP servers are nanoseconds, but after arithmetic calculations precision is reduced to microseconds. In addition, to reduce the timing error, execution time of all the 6 iterations of tests are not recorded. But only the first iteration and the average of all other 5 iterations are recorded. The result of tests with milliseconds precision are stated in Table 4.1.

	Test 1		Test 2		Test 3		Test 4	
	100 files 10 MB each Total 1GB		20,000 files 52KB each Total 1GB		100 files 100 MB each Total 10GB		20,000 files 520KB each Total 10GB	
	1st	Ave.	1st	Ave.	1st	Ave.	1st	Ave.
Bare Metal	*0.260*	*0.046*	*22.033*	*11.123*	0.332	0.046	34.504	11.123
Bare Metal with IMA	*18.398*	*0.047*	*1034.534*	*11.124*	106.386	0.047	1110.298	12.186
Dom0	*0.252*	*0.059*	*28.827*	*13.460*	0.347	0.060	39.143	13.467
Dom0 with IMA	*18.643*	*0.144*	*1098.977*	*46.854*	106.872	0.144	1175.865	47.495
DomU	*0.348*	*0.059*	*31.873*	*13.450*	0.506	0.059	49.075	13.462
DomU With IMA	*15.158*	*0.060*	*86.159*	*13.601*	98.002	0.066	199.328	30.183
VMI Breakpoint	*15.522*	*0.064*	*911.183*	*14.338*	117.502	0.097	931.106	14.384
VMI HMAC	*20.980*	*0.135*	*1985.688*	*30.713*	120.916	0.206	2010.153	30.977
VMI RSA	*23.995*	*0.109*	*2402.922*	*24.750*	124.306	0.109	2449.640	25.085

Table 4.1: Test results

From Table 4.1, following patterns can be deduced:

1. The performance of DomU-IMA is better than bare metal-IMA and Dom0-IMA, while initially it was expected to observe different results. It is not yet figured out what is the reason behind this pattern; but as this happens only when IMA is enabled, it is believed that it is due to the internal structure of IMA.

2. In system configurations which IMA-Appraisal is not used, such as bare metal, Dom0 and DomU, initially it was expected to observe almost identical values for the first execution time and the average execution time of other iterations. But first execution time is longer than the average value. This happens as for the first iteration, Linux has to load files into memory. For other iterations, files are already loaded into memory. Test results, in these system configurations indicates the impact of Linux cache on performance.

3. In cases where IMA-Appraisal is enabled, the first execution time is considerably longer than the average time value of the rest iterations. This shows the impact of IMA-Appraisal in the systems. During first iteration

of tests, IMA has to calculate hash digests of files, while in other iterations their hash digests are read from IMA run time table. Since files content has not changed, the hash digest need not to be re-computed.

4. In cases that IMA is enabled the average execution time after the first iteration is longer compared to systems that IMA is not used. The reason is IMA has to look up its run time table. This induces a little performance overhead on the system.

5. In systems that IMA-Appraisal is enabled, first execution time of test 1 is almost 6 times longer than first execution time of test 3. In test 1, 10MB files are read while in test 3 100MB files are read. Although more tests are needed here, but It demonstrates IMA performance decreases dramatically as the file size grows.

6. By considering first execution time of tests in DomU or systems that are implemented in DomU, it can be observed that for big files, i.e. 10MB and 100MB files, the overhead is mostly due to the use of IMA. However, for small files, i.e. 52KB and 520KB files, over head is mostly the result of context switching. In test 1 first execution time grows from 0.348 seconds in DomU to 15.158 seconds in DomU with IMA enabled. In test 3, the same pattern holds. But in test 2, first execution time suddenly grows from 86.159 seconds in DomU-IMA to 911.183 seconds in DomU-BR. This pattern holds for test 4 as well.

7. The first execution time in DomU-IMA, VMI-BR, VMI-HMAC, and VMI-RSA are considerably longer than the first execution time in DomU. However, the average values of those systems are almost the same. IMA-Appraisal and VMI context switching are CPU intensive functionalities, but as an internal cache, IMA run time table, is used, the overall system performance is acceptable.

8. By comparing results of tests, it appears that the performance penalty induced by VMI-RSA is almost the same as the performance penalty in VMI-HMAC. As a result it can be concluded these two security functions are almost identical regarding system performance.

All in all these tests have shown that even both IMA-Appraisal and VMI context switching are CPU intensive tasks, using an internal cache, IMA run time table in these designs, improves the system performance dramatically.

4.3 Limitations

According to the system specification and the implementation, these two security functions have some limitation:

1. Since VMI-RSA is designed not to possess the private key it cannot update files extended security attributes. This means files which meet IMA policy, by default files owned by root user, cannot be updated in this system. All changes and updates have to happen using other interfaces. To add or update a file, its signature should be provided by a trusted party.

2. To start VMI early in boot process, an offline breakpoint is inserted in the kernel. This modification made the kernel dependent on the security functions. Even though this happens only when IMA-Appraisal is enabled by the "ima_appraise=" argument. Effectively this means in the mentioned condition it is not possible to boot the kernel without VMI. The Best solution to overcome this limitation is to use hardware breakpoints instead of software ones.

3. These security functions need the compiled kernel directory for bridging the semantic gap. Since VMI-HMAC and VMI-RSA security functions, need to access variables and structure, gdb needs compiled kernel directory to access its variables.

4. Performance penalty is the result of context switching. A number of context switching happens for handling the breakpoint interrupts. Context switching is considered CPU intensive tasks. These switching reduces system performance. To mitigate this penalty, the least number of breakpoints have to be inserted. For this reason in each of these systems only one breakpoint is used.

Limitation 1 is the result of system specification, while limitations 2 and 3 are the result of the implementation. However, performance penalty in limitation number 4 is related to the nature of breakpoints and context switching.

Chapter 5

Conclusion

5.1 Contribution

In this thesis, Virtual Machine Introspection (VMI) and its application in security area are studied. After deep survey of VMI and available VMI libraries, two security functions are designed. They use an active VMI method to satisfy the integrity requirement of a guest virtual machine. Although the main goal was to design a system that verifies the RSA signature of files in the target guest VM, as a result of limitation about file modification, one more security function with a slightly different approach was designed.

In the main design the security function verifies the RSA signature of files, when they are loaded in the guest VM. This verification happens in Dom0 by VMI application, using the proper public key. But since the system specifications prohibit Dom0 and the security function from possessing the private key, this design loses the ability to modify files in normal ways. This is not a major problem since the security policy only meets the files that are important and are meant to be immutable. However, another security function capable of modifying files and their security attributes was designed. In the second design, the VMI application in Dom0 checks the HMAC-SHA1 digest of files, when that file is accessed in the guest VM. HMAC-SHA1 crypto function uses a secret key that is known to Dom0 and the security function. While the first design is more secure, the second one has no problem with file modification. Both solutions use Integrity Measurement Architecture (IMA) to retrieve the needed information, and to enforce the security policy. IMA is a new security function in the Linux kernel, but as it is a kernel space solution it is vulnerable to rootkits. These security functions take advantage of IMA-Appraisal and VMI to raise security properties of a guest VM.

Both solutions use the kernel debugging techniques, which are implemented in the Xen hypervisor. The Xen hypervisor has a built-in gdb server, which

can communicate with gdb. By using the python extension in gdb, one can automate the debugging process. These security functions take advantage of this possibility to control the behavior of a guest VM. By inserting a breakpoint in a guest VM kernel, these systems get control of the target VM whenever that breakpoint is hit. While most VMI libraries and applications are passive solutions, the VMI method used in this thesis, is able to actively monitor a guest VM for prevention purposes. In addition, it is already incorporated in the Xen hypervisor.

As it is vital for guest VM integrity to be checked constantly, the security functions starts as soon as the guest VM kernel loads in the memory. To guarantee the integrity of the boot process, the hash of MBR, and boot loader files including kernel files, are validated against good known hash digests. These checks guarantee that the correct and known boot loader loads a valid kernel.

Having implemented these two security functions, a number of possible attacks, and solutions for some of them, are enumerated. To answer the possibility of kernel compromise in guest VM, which can lead to IMA misbehavior, the immutable files extension is added to each of these systems. The idea behind this extension is to compare the good known hash digest of immutable files with the values provided by IMA. For immutable files it is expected to retrieve identical known digests from IMA.

One major drawback of VMI is its performance penalty for high rate applications. Some context switches are required to handle a breakpoint interrupt. These context switches reduce the system performance. For this reason, VMI has been suggested for low rate application. In this thesis low rate VMI applications are designed to mitigate this limitation. In each design only one breakpoint is used. After implementing a prototype of the security functions, their performances are measured. Four different tests are used to measure performance of 9 different system configurations, such as bare metal, Dom0. By comparing measured results, it is realized that the nature of these security functions reduces system performance. However, as IMA uses an internal cache, the wholes security function does not have significant performance penalty. This means that for normal applications, the performance penalty is negligible while the security characteristics of the guest VMs are clearly improved.

5.2 Future Work

Having completed the system implementation, the project reached to a point that, further system development required complete IMA modification. Hence one obvious way to improve these systems is to not rely on IMA and to develop a new kernel module. Using a new kernel module provides the possibility of adding new functionalities to these security functions. Furthermore the RSA design issue about file modification is another area that needs more research. Finally,

to start VMI application early in the boot process minute kernel modification had to be made. Although this modification does not affect kernel normal behavior, it is desirable to have a solution which does not need any kernel modification. One possible and clean solution is using the hardware breakpoints instead of software ones. However, currently hardware breakpoints are not implementing in gdbsx in Xen. A path to improve the system is, to develop hardware breakpoints capability or to find another solution which does not require any kernel modification.

Bibliography

[1] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *2ACMConfSec*, 1994.

[2] David Gordon Ericsson, Serge Hallyn, Ibm Ltc, Makan Pourz, I Ericsson, and Vincent Roy Ericsson. Digsig: Run-time authentication of binaries at kernel level axelle apvrille – trusted logic, December 29 2008.

[3] Advanced Intrusion Detection Environment. http://aide.sourceforge.net/. April 2012.

[4] IMA Community. Linux integrity subsystem. http://linux-ima.sourceforge.net, April 2012.

[5] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[6] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programmings; Chapter 5 Protection; Section 5.5 Privilege levels. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[7] VMware. White paper: Understanding full virtualization, paravirtualization, and hardware assist.

[8] Nitin A. Kamble, Jun Nakajima, and Asit K. Mallick. Evolution in kernel debugging using hardware virtualization with xen. Linux Symposium, July 2006.

[9] K. Nance, M. Bishop, and B. Hay. Virtual machine introspection: Observation or interference? *Security Privacy, IEEE*, 6(5):32 –37, sept.-oct. 2008.

[10] Xen. Xen.org.

[11] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*. The Internet Society, 2003.

[12] J. Pfoh, C. Schneider, and C. Eckert. Exploiting the x86 architecture to derive virtual machine state information. In *Emerging Security Information Systems and Technologies (SECURWARE), 2010 Fourth International Conference on*, pages 166 –175, july 2010.

[13] Nguyen Anh Quynh and Kuniyasu Suzaki. Abstract xenprobes, A lightweight user-space probing framework for xen virtual machine, August 14 2008.

[14] Brian Hay and Kara Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, April 2008.

[15] 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.

[16] B.D. Payne, M.D.P. 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, dec. 2007.

[17] F. Baiardi and D. Sgandurra. Building trustworthy intrusion detection through vm introspection. In *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on*, pages 209 –214, aug. 2007.

[18] libvmi. http://code.google.com/p/vmitools/.

[19] B.D. Payne, M. Carbone, M. 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, may 2008.

[20] Richard Stallman, Roland Pesch, Stan Shebs, and et al. Debugging with gdb - the gnu source-level debugger. In *gdb*. Free Software Foundation, 2011.