Enforcing virtualized systems security

Hedi Benzina

ENSC-2012n°424

THÈSE DE DOCTORAT
DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Présentée par

Hédi Benzina

Pour obtenir le grade de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Domaine :
Informatique

Sujet de la thèse :

Enforcing Virtualized Systems Security

Thèse présentée et soutenue à Cachan le 17 décembre 2012 devant le jury composé de :

Frédéric Cuppens	Rapporteur
Jean-Yves Marion	Rapporteur
Florent Jacquemard	Examinateur
Frédéric Majorczyk	Examinateur
Romain Feybesse	Membre invité
Jean Goubault-Larrecq	Directeur de thèse

Projet soutenu par l'attribution d'une allocation doctorale Région Ile de France - 0810181818

Laboratoire Spécification et Vérification
ENS de Cachan, UMR 8643 du CNRS
61, avenue du Président Wilson
94235 CACHAN Cedex, France

Abstract

Virtual machine technology is rapidly gaining acceptance as a fundamental building block in enterprise data centers. It is most known for improving efficiency and ease of management. However, the central issue of this technology is security. We propose in this thesis to enforce the security of virtualized systems and introduce new approaches that deal with different security aspects related not only to the technology itself but also to its deployment and maintenance.

We first propose a new architecture that offers real-time supervision of a complete virtualized architecture. The idea is to implement decentralized supervision on one single physical host. We study the advantages and the limits of this architecture and show that it is unable to react according to some new stealthy attacks.

As a remedy, we introduce a new procedure that permits to secure the sensitive resources of a virtualized system and make sure that families of attacks can not be run at all. We introduce a variant of the LTL language with new past operators and show how policies written in this language can be easily translated to attack signatures that we use to detect attacks on the system.

We also analyse the impact that an insecure network communication between virtual machines can have on the global security of the virtualized system. We propose a multilevel security policy model that covers almost all the network operations that can be performed by a virtual machine. We also deal with some management operations and introduce the related constraints that must be satisfied when an operation is performed.

Résumé

La virtualisation est une technologie dont la popularité ne cesse d'augmenter dans le monde de l'entreprise, et ce pour l'efficacité et la facilité de gestion qu'elle apporte. Cependant, le problème majeur de cette technologie est la sécurité. Dans cette thèse, nous proposons de renforcer la sécurité des systèmes virtualisés et nous introduisons de nouvelles approches pour répondre aux différents besoins en sécurité de cette technologie et aussi aux aspects liés à à son fonctionnement et son déploiement.

Nous propososns une nouvelle architecture de supervision qui permet de contrôler la totalité de la plateforme virtualisée en temps réel. L'idée est de simuler une supervision décentralisée (plusieurs postes) sur un seul poste physique. Nous étudions les avantages et les limites de cette approche et nous montrons que cette solution est incapable de réagir é à un certain nombre d'attaques nouvelles.

Comme remède, nous introduisons une nouvelle procédure qui permet de sécuriser les ressources critiques d'un système virtualisé pour s'assurer que des familles d'attaques ne peuvent être exécutées en ayant accès à ces ressources. Nous introduisons une variante de LTL avec de nouveaux opérateurs de passé et nous montrons comment des politiques de sécurité formulées à l'aide de ce langage peuvent être facilement traduites en signatures d'attaques qui peuvent nous être très utiles pour la détection des intrusions dans le système.

Nous analysons aussi l'impact d'une communication réseau non sécurisée entre machines virtuelles sur la sécurité globale du système virtualisé. Nous proposons un modèle d'une politique de sécurité multi-niveaux qui couvre la majorité des opérations liées au réseau et qui peuvent être exécutées par une machine virtuelle. Notre modéle traite aussi certaines opérations de gestion de l'infrastructure virtualisée et les contraintes de sécurité qui doivent être satisfaites.

Remerciements

Je remercie en premier lieu mon directeur de thèse Jean Goubault-Larrecq pour les pistes de recherche qu'il a su me conseiller au cours de ces trois années et pour ses conseils avisés. Je tiens également à remercier Frédéric Cuppens et Jean-Yves Marion pour m'avoir fait l'honneur de relire ma thèse. Je tiens à remercier aussi Frédéric Majorczyk, Romain Feybesse et Florent Jacquemard d'avoir accépté de faire partie du jury de ma thèse.

Un grand merci également à plusieurs personnes qui m'ont, d'une manière ou d'une autre, encouragé à entreprendre une thèse : Adel Bouhoula et Florent Jacquemard pour m'avoir accordé leur confiance et leur soutien pendant mon stage au LSV.

Évidemment, merci au LSV pour la très bonne ambiance qui y règne, et qui permet de réaliser une thèse dans d'excellentes conditions. Aussi un grand merci à Virginie, Catherine, Ahmad, Sandie et Philippe pour leur aide.

Enfin, merci à mes parents sans qui je ne serais pas là, et à ma femme pour son soutien quotidien.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context of the thesis

Today's IT intensive enterprise must always be on the lookout for the latest technologies that allow businesses to run with fewer resources while providing the infrastructure to meet today and future customer needs. Virtualization is the cutting edge of enterprise information technology. In recent years the term virtualization has become the industry's newest buzzword. Virtualization technology is possibly the single most important issue in IT and has started a top to bottom overhaul of the computing industry. The growing awareness of the advantages provided by virtualization technology is brought about by economic factors of scarce resources, government regulation, and more competition.

Virtualization technology is being used by a growing number of organizations to reduce power consumption and air conditioning needs and trim the building space and land requirements that have always been associated with server farm growth. Virtualization also provides high availability for critical applications with a streamlines application deployment and migrations. Furthermore it simplifies IT operations and allow IT organizations to respond faster to changing business demands.

In a few words, this technology is a combination of software and hardware engineering that creates Virtual Machines (VMs) by abstracting the computer hardware and allowing a single machine to act as if it were many machines.

In surveys of senior-level IT managers, security is consistently one of the top five concerns, along, specifically, with security related to the hot technology of the moment. Most recently those worries have included social-networking technologies such as Twitter and Facebook and other outlets through which employees could turn loose company confidential data. But the security of virtual servers and virtualized infrastructures also rank near the top of the list and rightly so, according to analysts.

In such technologies, security is very important. For instance, if an attacker succeeds to break the access control mechanism and penetrates one sensitive virtual machine such as the administration one, then, all the rest of machines (sometimes hundreds of machines running virtual servers) are under his control. This is more dangerous than having an isolated machine being attacked. Furthermore, once the system is compromised, all the sensitive data stored in dedicated virtual machines can be compromised.

We address in this thesis the security of virtualized systems *i.e.* systems running under a Virtual Machine Monitor (VMM). We discuss their security issues, present defense mechanisms and introduce new approaches for strongly securing both sensitive resources and communication.

1.2 Contributions

We study in this thesis the security of virtualized systems. We identify security threats and propose new approaches to secure such systems. First, we focus on the the design and implementation of a new intrusion detection architecture dedicated to the supervision of virtual machines running under the control of an hypervisor. This implementation protects both the system resources such as VMs and the defense engine which is in this case our intrusion detection system Orchids [22, 91]. Further, we show that this implementation needs to be improved and complemented by formal methods that help system administrators design security policies, express the security properties that they want to see satisfied and deploy them in order to protect sensitive resources such as the administration VM. Finally we study the security of communication in virtual environments and introduce a new security policy model that fills the security requirements of both network and virtual system management operations. We detail our contributions below.

1.2.1 A Decentralized Supervision System for Securing Virtual Machines

In chapter 3 we focus on the security of virtual machines. We study the existing security threats and propose a new approach for protecting system VMs from outside. We do this by deploying an Intrusion Detection System (IDS) out of the supervised VMs, and equipping all VMs by small sensors reporting at real time all the system actions performed by the users/system. The IDS can react to attacks, stop them and even kill a whole VM or restart it from an early checkpoint. This architecture was designed and implemented in collaboration with Bertin Technologies and was published in 2010 in the SETOP Workshop [5, 4]. This approach is cost-effective and can be adapted easily to different platforms thanks to the modularity in the implementation.

1.2.2 A Temporal Language for Securing Sensitive Resources

In chapter 4, we study the security of sensitive resources in virtual environments. We show that attacking the administration VM for example can lead to the subversion of the whole system. As a defense, we propose to let the administrator write security policies expressing safety properties in a simple language that we qualify as a variant of Linear Temporal Logic with past operators. LTL with past operators has been proved to be more succinct than pure-future temporal logic [18]. Expressing policies in this language is quite intuitive. Then, we propose an algorithm that translates the aforementioned policies into attack signatures that can feed the attack base of the Orchids IDS. This helps automating the generation of new attack rules and simplifies the monitoring of growing security threats. This contribution was published in [5, 3].

1.2.3 A Multi-level Security Policy for Securing Communication

The contributions cited above do not cover the network security aspect, thus we introduce in chapter 5 a Multi-level security policy model in order to secure communication in virtual networks built using virtual machine monitors. Communication is very important in such systems. For instance, the information flows between the supervision VM and other VMs is guaranteed thanks to a virtual network that we build by hand. If an attacker succeeds to capture some flowing data, he will know more about the deployed security mechanism which represents a real security threat. Our policy model covers different aspects of networking and also deals with operations related to the management of the virtual resources [2, 1].

1.3 Research Publications

1.3.1 Conferences and Workshops

The results obtained in this thesis have been partially published:

1. H. Benzina. Towards Designing Secure Virtualized Systems. In Proceedings of The Second International Conference on Digital Information and Communication Technology and its Applications (DICTAP 2012), Bangkok, Thailand. IEEE Computer Society Press, 2012.

2. H. Benzina. A Network Policy Model for Virtualized Systems. In Proceedings of The Seventeenth IEEE Symposium on Computers and Communication (ISCC 2012). Cappadocia, Turkey. IEEE Computer Society Press, 2012.

3. H. Benzina. Logic in Virtualized Systems. In Proceedings of the First International Conference on Computer Applications and Network Security (ICCANS 2011), Malé, Maldives. IEEE Computer Society Press, 2011.

4. H. Benzina. Securing Hypervisors through Temporal Logic and Security Policies. Workshop on Formal methods for specifying and verifying critical systems 2011. Tunis, Tunisia.

5. H. Benzina and J. Goubault-Larrecq. Some Ideas on Virtualized Systems Security, and Monitors. In The third International Workshop on Autonomous and Spontaneous Security (SETOP 2010), Athens, Greece. Springer LNCS 6514.

1.3.2 Research Tools

The implementation of the tool that was built as part of our research is available here :

- H. Benzina. RuleGen, a tool for compiling security policies written in a variant of LTL with past into automata representing attacks signatures (http://www.lsv.ens-cachan.fr /~benzina/rulegen.php).

1.4 Thesis Plan

Chapter 2 introduces concepts standard in the literature and discusses the main contributions in the field of securing virtualized systems. In Chapter 3, we present our contribution in the field of intrusion detection in virtual environments, it consists of a decentralized supervision system implemented on top of the Xen hypervisor. This chapter is rather small and we choose to start by presenting this implementation in order to show its advantages and also its limits against new attacks. Based on thees limits we introduce in chapter 4 a new approach for securing sensitive resources in virtual environments that aims to defend the system against stealthy attacks that cannot be detected by the aforementioned implementation, and we introduce a new temporal language which is a variant of LTL with past that helps system administrators write their own security policies. Furthermore, we show how to translate the written security policies into attacks signatures that can be used by the Orchids IDS. In chapter 5, we discuss the security threats that virtualized systems can face while network primitives in a local virtual network are invoked and present a Multilevel Security Policy dedicated to the enforcement of communication security. This policy covers also all main VM-management operations. Chapter 6 concludes the thesis with a summary of the results obtained and presents perspectives and possible future work.

1.5 The REDPILL project

This thesis has been done in the framework of the Digiteo REDPILL project "Malware Detection On Virtualized Platforms", grant 2009-41D, involving the company Bertin Technologies, and the LSV laboratory (École Normale Supérieure de Cachan).

Chapter 2

State of The Art

2.1 Introduction

In this chapter, we review several standard concepts, definitions and contributions in virtualization technology, intrusion detection and security in general.

2.2 Virtualization

Virtualization is not a new idea. In fact, it goes back to the early days of computing. We can mention the work of Popek and Goldberg in 1974 [6], which analyzed the different possible types of virtualization solutions, their disadvantages and laid the groundwork for future developments. Virtualization permits to run an operating system inside a virtual machine, which allows running multiple operating systems in the same physical host and sharing costly resources. Historically, virtualization has become fashionable in 2006, when new software running Windows in Mac OS X appeared. Since then, this technology has been integrated into Windows 7, and was built in the heart of computers: first at the processor and then, recently, at the device level. Nevertheless, this remains a rather mysterious technology for the general public.

A *Virtual Machine (VM)* is the set of hardware (CPU, memory, hard disk, peripherals, etc..) emulated by the virtualization software and viewed by the guest operating systems. Specifically, we are talking about HVM (Hardware Virtual Machine). Popek and Goldberg defined a virtual machine as "an efficient, isolated duplicate of a real machine".

A *Virtual Machine Monitor (VMM)*, or virtual machine manager is the virtualization software itself. Two types of *VMM* exist, the first one can be installed as an application on a host (Linux, Mac OS X, Windows, etc..). The second, commonly called a *hypervisor*, is actually a very simple operating system (Linux or Windows) containing the virtualization program. The difference is important in the case of critical applications: using the second type of VMM avoids wasting resources with a host system. Virtual machines can be useful in many areas, often in the professional field where many Applications do not require the power of a server, but where the segmentation of services however requires administrators to dedicate one to each task. The first one is to take advantage of many OSes at the same time, more easily than in a multi-boot. It is thus possible to have Windows on a Macintosh or Windows, Mac OS X and Linux on one machine, etc... Beyond the gimmick, it's an advantage for all software developers who need to test their code under each platform, as each browser, etc (see Figure 2.1)... Note that, for now, probably by the will of Apple, it is impossible to install the client version of Mac OS X on a

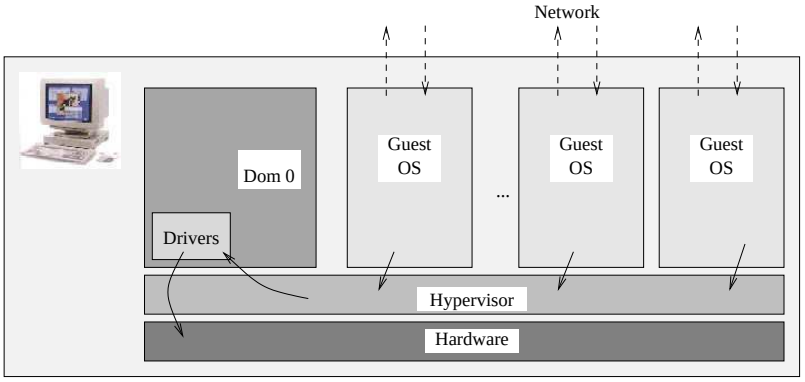

Figure 2.1: A Virtualized System

PC. The opposite is quite possible. Another advantage of using virtual machines is the stability and the increased security of the system : if a VM crashes, the other VMs are not affected. In addition, each VM is encapsulated in a file. It is therefore very easy to make a backup of the system at a given time. This file is also easily transferable from one computer to another. A boon to system administrators and others who regularly change their PC.

2.2.1 Popek and Goldberg Virtualization Requirements

The Popek and Goldberg virtualization requirements [6] are a set of conditions allowing a computer system to implement the *virtualization* technology correctly. They defined the Virtual Machine Monitor (VMM) as a software having some essential characteristics. Programs running under the VMM should find the same conditions as if they were running under ordinary machines, the VMM has to provide an environment which is identical with the original machine. This should not affect the speed of the system. They required also that the VMM has a complete control of system resources. Another characteristic of a VMM is efficiency. It demands that most of the virtual processor's instructions can be executed directly by the real processor, with no software intervention by the VMM. This statement rules out traditional emulators and complete software interpreters (simulators) from the virtual machine umbrella.

The third characteristic, [...resource control, labels as resources the usual items such as memory, peripherals, and the like, although not necessarily processor activity. The VMM is said to have complete control of these resources if (1) it is not possible for a program running under it in the created environment to access any resource not explicitly allocated to it, and (2) it is possible under certain circumstances for the VMM to regain control of resources already allocated...]. [6].

The Virtual Machine Monitor is defined as a particular piece of software called *control program* composed of several modules. These modules fall into three groups : the first one is a *dispatcher* D, that controls the call of other modules. The second one is an *allocator* A that decides whether a resource should be allocated or not. In the case of one single VM, the allocator has only to provide the separation between this VM and the VMM. But when several VMs are running on top of the VMM, the allocator has to handle the access to shared resources. The allocator will be invoked by the dispatcher when a VM tries to execute some privileged instruction that attempts to change the resources associated to this VM. The third set of modules is called *interpreters*. An interpreter is associated to each privileged instruction.

Another interesting part of this work is the specification of the virtual machine properties. The authors have presented three properties of VMs. The first one is *the efficiency property*.

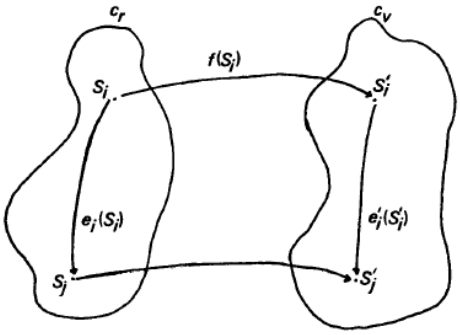

Figure 2.2: The Virtual Machine Map (source : [6])

All unprivileged instructions are executed by the hardware directly, with no intervention of the VMM. The second one is *the resource control property* which ensures that every program should go through the allocator in order to access system resources. The last one is *the equivalence property*.

The availability problem arises from this configuration. It occurs when the allocator fails to satisfy a particular request for a resource. Then the program asking for this resource will be unable to run. Thus the virtual environment is said to be "smaller" than the real system. The authors define the VMM as any *control program* that satisfies the three properties (efficiency, resource control and equivalence). Then functionally, the environment which any program sees when running with a virtual machine monitor present is called a virtual machine. It is composed of the original real machine and the virtual machine monitor.

Theorem. [...For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions...][6].

The theorem provides a condition sufficient to guarantee virtualizability. However, those features which have been assumed are standard ones, so the relationship between the sets of sensitive and privileged instructions is the only new constraint. It is a very modest one, easy to check. Further, it is also a simple matter for hardware designers to use as a design requirement.

Virtual Machine Map Figure 2.2 shows the mapping $f : C_r \rightarrow C_v$ between instructions in the virtual environment. That is for any state $S_i \in C_r$ and any instruction sequence e_i, there exists an instruction sequence e_i' such that $f(e_i(S_i)) = e_i'(f(S_i))$. Two related properties exist in the definition of a VM map. The first one is the existence of instruction sequences e_i' on the C_v domain that correspond to the sequences e_i on the C_r domain. The second one is the mathematical existence of a particular mapping from the states of the real machine to the virtual machine system.

2.2.2 Some Challenges

A processor is capable of running a small number of basic instructions. This set, called ISA (Instruction Set Architecture), is encoded in the processor and is not editable. It defines the capabilities of a processor, the hardware architecture which is then optimized to execute the instructions in the ISA as efficiently as possible. The best known ISA in the PC world is the

x86, used from the beginning by Intel and taken over by AMD chips. One can also mention the PowerPC, ARM, MIPS, etc.. Widespread, even ubiquitous, the x86 is not provided free of defects, but it was out of question to replace it by another technology. To avoid this, Intel and AMD developed respectively the VT-x and AMD-V solutions. If the x86 is not well suited to virtualization, it is because of 17 critical non-privileged instructions. The instructions of the x86 ISA are not similar. Some of them can change the configuration of CPU resources and are called critical. To protect the stability of the machine, these instructions can not be executed by all software. From the perspective of the CPU, software belongs to four categories, or levels of abstraction: the rings 0, 1, 2, 3. Each ring defines a decreasing level of privilege. The most critical instructions claim the highest privileges, of order 0. Unfortunately, on an x86 processor, 17 of these critical instructions can be executed by the same software tier 1, 2, or 3. This constitutes a big problem for VMMs. An operating system is actually designed to run in ring 0 and use critical instructions to allocate CPU resources between different applications. But in a situation when it is a guest on a virtual machine, the OS should not even be able to modify the material, otherwise it would crash the entire system. Only the hypervisor must have these rights. It is therefore critical that all instructions are intercepted. It's very easy for all privileged instructions. The OS is then executed in ring 3, as applications, and all requests for privileged instructions trigger an error that is handled by the VMM. This is much more complicated for the 17 hazardous and non-privileged instructions. These do not trigger automatically an error, they must be detected piecemeal by the VMM and then reinterpreted. This enrolls a high overhead, make the hypervisor more complex.

2.2.3 Types of Virtualization

We distinguish two types of virtualization: full virtualization and hardware virtualization. Full virtualization is the primitive virtualization which emulates the physical hardware and its behavior. This is the most costly approach but the easiest one to implement. Hardware virtualization is an extension of the principle of full virtualization. This extension is done by the use of specific processor extensions for virtualization (AMD-V and Intel-VT). These extensions can accelerate the virtualization by different mechanisms. Solutions using this technology are VMWare [8], Sun VirtualBox [9], Microsoft Virtual PC [10], QEMU [11] and many others.

The virtualization of operating systems is called *paravirtualization*. This term tends to be used in many different ways. Paravirtualization is the virtualization of operating systems whose kernels has been modified to communicate with the virtualization layer instead of communicating with the physical hardware. To summarize, the operating system will be aware of being virtualized and will be adapted for this purpose. The simple addition of specific drivers does not necessarily imply paravirtualization. Existing solutions in this area are the products of Citrix XenServer, Sun xVM, XenSource or Microsoft Hyper-V. VMWare starts to get into this technology safely.

Hardware Virtualization

To overcome the problem cited in the previous section, Intel designed VT-x and AMD proposed AMD-V. These two technologies are very similar. It consists of three components, aiming to make the virtualization of the CPU, the memory and devices easier. To facilitate the virtualization of the CPU, Intel and AMD eliminated the need for monitoring and translating the instructions. To do this, new instructions were added. A new control structure is also being introduced, called VMCS (Virtual Machine Control Structure) at Intel. Among the new instructions, one of them (VM entry at Intel) toggles the processor in another execution mode, dedicated to the guest systems. This mode also has four different levels of privilege. With doing so, guest OS can run

Figure 2.3: Mobile Virtualization

in ring 3 of VM mode. If needed, the processor can switche from guest to normal mode. This scale is determined by some conditions set by the VMM using the control bits stored in the VMCS.

Desktop Virtualization

Desktop Virtualization is part of the great family of virtualization technologies with. The first principle of desktop virtualization is to display on one host, tens, hundreds or even thousands of physical hosts, a virtual image of the user station which is actually really executed on a remote server. Behind this great principle, however there are several forms of desktop virtualization. The oldest is Server-Based-Computing, consisting of virtualizing some applications but not the entire operating system. While the user sees (and uses) on his host the applications running on a remote server, the Os is still running on the client side. A variant exists which is application virtualization by isolation. Also called isolation by applicative bubbles, this type of virtualization installs an application with remote streaming on the workstation. It is the least common type but can solve the problems of incompatibility between applications and OSes. Desktop Virtualization may also be related to the operating system streaming. In this configuration, the target system boots from a remote disk on the network and load only the applications that the user wants, this can be done using logical volumes installed on a remote server. Another form of Desktop Virtualization is the VDI architecture (Virtual Desktop Initiative), also known as Hosted Virtual Desktop (HVD). It consists of a total virtualization of the host (applications that operating system), allowing to overcome the problem of incompatibility with the client host.

Mobile Virtualization

The mobile phone is now as important to businesses as desktop computers, and acts as a mobile computer in many cases. Mobile Virtualization is a new technology used mainly for Android phones to separate personal applications from professional ones in order to reduce the risk of compromising data.

This technology was first presented by VMWare [8] in 2009 by their *VMWare Mobile Virtualization platform* which creates a virtual machine for mobile devices, allowing users to move their phone to different handsets (see Figure 2.3. All the data will be stored in a portable file : this solves the problem of loosing the data when the mobile phone is compromised.

Paravirtualization

Paravirtualization is another type of virtualization. Here, the guest operating system is aware of running in a virtualized environment, which of course requires some modifications of the software. In return, it becomes capable of interacting with the hypervisor and to ask it to transmit calls directly to the hardware of the host server. In theory, the virtual performance is then very close to the performance achieved with real hardware. The hypervisor is in direct contact with the physical hardware. It is the exclusive intermediary between the hardware and the operating systems. All operating systems are virtualized in the sense that they have a core adapted to the virtualization layer. Some OSes can have specific rights to access some ressources : this depends on what the administrator wants from his software.

A *hypervisor*, also called virtual machine manager (VMM), is a virtualization technique that allows to run many OSes on the same physical host. The physical resources are shared between the different OSes using *hypercalls*. The most known hypervisor is *Xen* [12]. A *hypercall* is a software trap from a guest domain (or host) to the hypervisor, just as a syscall is a software trap from an application to the kernel. The hypercall is synchronous, but the return path from the hypervisor to the guest domain uses event channels. An event channel is a queue of asynchronous notifications, and notify of the same sorts of events that interrupts notify on native hardware. When a domain with pending events in its queue is scheduled, the OS's event-callback handler is called to take appropriate action.

2.3 Intrusion Detection

Intrusion Detection aims to detect actions that attempt to compromise the integrity, the availability or confidentiality of a resource. Early work in intrusion detection began with Anderson [13] in 1980 and Denning [14] in 1987. Today there are more than 140 intrusion detection systems [15]. Intrusion Detections Systems (IDS) are designed to reveal, usually through alerts, any activity that may be considered intrusive by analyzing information from various areas within a computer or a network to identify possible security breaches.

Intrusion Detection Systems are generally classified into two broad categories depending on the type of data to analyze [16]: Host-based IDS (HIDS) and Network-based IDS (NIDS). HIDS are characterized by the analysis of events or log messages generated by the system. NIDS analyze the data that travels over the network. An IDS performs a passive scan. The passive analysis is to be contrasted with the active analysis, this is the case for example a firewall that blocks certain packets. Intrusion detection functions include:

- Analyzing both user and system activities
- Checking system and file integrity
- Recognising patterns of attacks (*Pattern Matching*)
- Alerting users when security policies are violated (sending emails, logging...)

The different modules making up an IDS according to standards proposed by the Intrusion Detection Working Group [17]. This architecture consists of three modules common to most

IDS. The *Activity* of the information system provides a source of data to some *Sensors*. These sensors then have the role to extract and process certain information in order to transmit shape events to an *Analyzer*. The analysis module then uses these events to detect a possible intrusion and generates alerts accordingly. These alerts are finally sent to an alert *Manager*. The latter is responsible for processing alerts from the various analyzers and report any suspicious activity on the system to the *administrator*. Finally, note that an intrusion detection system may consist of several sensors dealing with different data sources, multiple analyzers using different methods of analysing and multiple *Managers*.

The performance of an intrusion detection system, including its method of analysis, depends of two important concepts that allow to evaluate the performance [19] :

False Negatives. Ideally, any intrusion must result in a warning. An intrusion that is not detected, that is to say, did not generate alert, then constitutes a false negative. In other terms, false negative is the failure of an IDS to detect an actual attack. The reliability of an analyser depends on the rate of false negatives. This rate must be the lowest possible.

False Posistives. Any alert must correspond to an effective intrusion. When the IDS generates an alert that does not make sense, this alert is qualified false positive. The relevance (or credibility) of an analyzer is related to its rate of false positive which represents the percentage of false alarms.

2.3.1 Misuse Detection

Misuse detection detects a known attack via the definition of a scenario. This approach uses a knowledge base, called attack signature base and a method of pattern matching to recognize the defined signatures. A misuse IDS is then composed of: a set of sensors producing a stream of events, a base for attack signatures and an algorithm for pattern matching.

Attack Signatures

Each signature can be seen as a characteristic sequence of events of an attack differenting it from normal behavior. The construction of this base requires an accurate knowledge of the attacks and their parameters.

Attack Scenario

An attack scenario can be represented by an automaton and also by finite state machines. An automaton represents the sequence of actions needed to achieve the attack [20]. This approach allows one to express complex scenarios of attacks containing different ways to reach the same state. The automaton can also be expressed using specific language as in [22] or [21]. Several approaches [23, 24, 25] use a finite state automaton. The automaton can also be represented as a variant of a colored Petri net as in IDIOT [26] or in the form of state transition diagrams as in the NETSTAT tool [27]. The states of this automaton represent the recent history symbols (system calls) that were observed, a transition from one state to another characterizes the set of traces to be produced after this state.

Pattern Matching

The *Pattern Matching* uses algorithms to recognize a signature in a record corresponding to a sequence of events. This conventional approach is problematic when multiple scenarios give rise

to the same signature. To overcome this problem, some approaches use algorithms recognition based on genetic algorithms [28], bayesian networks [29] or some approaches doing the analysis of system configurations [30]. Other approaches use a *multi-events* correlation system, including pre-conditions, post-contentions and conditons [31, 33] to clarify the definition of scenarios. This approach gives a high performance in terms of analysis, but is generally the source of a high rate of false positives. Indeed, one limitation of this approach is that it is difficult to write a signature covering several variants of the same attack without generating false positives.

2.4 Security Policies

The definition and implementation of a security policy is the heart of systems security. A security policy defines a set of security properties, each property representing a set of conditions that the system must respect to remain in a state considered as *safe*. An incorrect definition or the partial application of a policy can lead the system to a non-safe state, allowing the theft of information or resources, the modification of information or the destruction of the system. In this section we give a general definition of some security properties and mechanisms used to implement a security policy. The security of computer systems is generally limited to ensuring the rights of access to data and system resources by implementing authentication mechanisms and controls to ensure that the users of these resources have only the rights that they were granted. The security mechanisms in place may still cause discomfort to at the user level while the instructions and rules are becoming increasingly complicated. Thus, information security must be studied in such a way that it does not prevent users from the necessary uses of the system. This is why it is necessary to define initially a security policy, that can be implemented according to the following four steps:

- Identify needs in terms of security, IT risks weighing on the company and their possible consequences
- Develop rules and procedures in order to protect the system
- Monitor and detect vulnerabilities of information systems and keep abreast of vulnerabilities on used applications and hardware
- Define the actions to take and who to contact in case of detection of a threat (the administrator of the system in most cases)

If we consider the system as a finite state machine with a set of transitions (operations) that change the system state, then a security policy can be seen as a way that partitions these states into authorized and unauthorized states. Given this simple definition, we can define a secure system as a system that begins in an authorized state and will never enter an unauthorized state.

2.4.1 Security Properties

Systems security is based on three fundamental properties: confidentiality, integrity and availability. The interpretation of these three areas varies depending on the context in which they are used. This representation is related to user needs, services and laws in force. The definition and application of these properties are part of the evaluation criterias of security. *Confidentiality* is based on the prevention of unauthorized access to information. The need of this property has emerged after the integration of critical information systems, such as government organizations or industries, in sensitive areas.

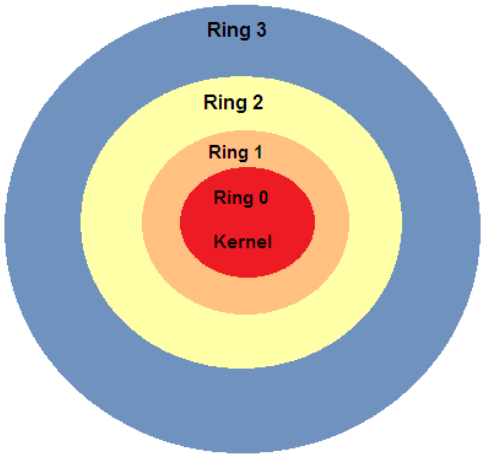

Figure 2.4: Protection Rings in x86-32 Systems

The *integrity* property refers to the state of data which, when processed, saved or transmitted, remains unaltered. In the case of a resource, the integrity means that the resource works correctly. The property of data integrity is to prevent unauthorized modification of information. Ensuring the fidelity of information with respect to their container is known as data integrity. The warranty information related to the creation or owners shall be known as the integrity of the original, more commonly called authenticity.

Availability refers to the ability to use a desired information or resource. This property should be accompanied with the reliability of the system, because having a system that is no longer available is a system. As part of the security, availability of property refers if an individual may deliberately deny access to certain information or resources of a system.

2.5 Virtualization and Security

2.5.1 Overview

In x86-32 systems, there are four rings of protection from 0 to 3 (see Figure 2.4). In almost all operating systems without virtualization, only the rings 0 and 3 are used (except in the GNU Hurd system [7]. The most privileged one is ring 0 which contains the OS kernel. The least privileged one is ring 3 which contains the applications and the dynamic data. The other two rings are not used. The diagram below shows the distribution of application components in a modern operating system. In paravirtualization, the OS does not have a direct access to the hardware. Only the hypervisor can access it directly. For security reasons, it will be necessary to totally separate the operating system and the hypervisor. In this case, ring 1 will be used. Thus the hypervisor will be placed in ring 0 and the OS will take ring 1. Applications remain in ring 3. Now that we have explained the utility of the protection rings. We have to understand how these rings are implemented in real? And where do they appear in nature? The protection rings are implemented in the memory. A RAM area is assigned to a particular ring by the OS. A program running in a memory area assigned to the ring 3 can not change a memory area assigned to the ring 0. When AMD and Intel redesigned the x86 architecture to move to the 64-bit architecture, they decided to remove the rings 1 and 2 because they were not used (see

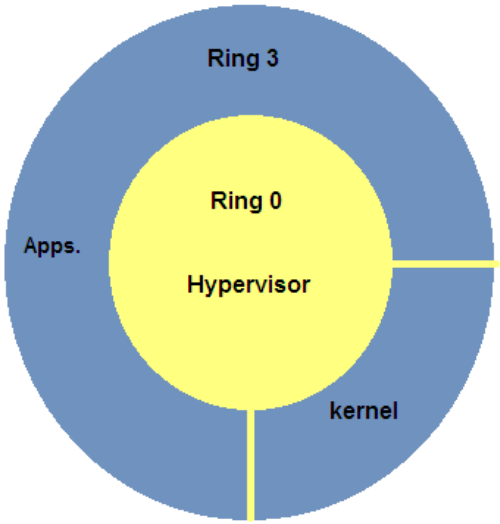

Figure 2.5: Protection Rings in Xen

Figure 2.5). This does not create a particular problem because these rings were not used in operating systems. Virtualization has come relatively soon after and had the habit of using an additional ring in order to partition the hypervisor, the operating system and the applications. Virtualization solutions has therefore been left with only two rings rather then three. To solve this problem, the Xen project comes with the idea that ring 3 will be shared by the OS and the applications. The hypervisor will run in a separate ring (ring 0).

Then, AMD and Intel quickly realized the importance that virtualization is taking. So, they decided to include the virtualization instructions in their processors to make the opearations related to this technique easier. These extensions have enabled hardware virtualization. At the same time, a ring "-1" was added to make the paravirtualization avoid sharing the ring 3 between the OS and applications, Figure 2.6 explains this new arcitecture. The partitioning of virtual machines is of course a basic characteristic of a virtualization platform. In fact, the hypervisor does not have the total control of virtual machines (VMs). It can simply turn them off, start or pause them. The partitioning is managed by restricting the access to the memory. Hypervisors have been specifically designed to prevent memory overflows. The only way to exploit these overflows is hypercalls in Xen for example. Sor far, this kind of vulnerability was not detected. Also in the case of full virtualization and hardware virtualization, virtual machines do not even have a specific interface with the hypervisor making this type of vulnerability impracticable. The real security risks in virtualization platforms are, mainly, at the management interface. The management interfaces are not specific to virtualization but their use in this particular case is generalized. Access to management interfaces must be secured by traditional network security mechanisms such as authentication methods. Once an interface is compromised, the data and the access to virtual machines remain safe. The interfaces do not generally have access to particular data, they only have a global view of the system in order to be able to configure the storage media. The access to a VM is protected by conventional protection mechanisms such as login and password.

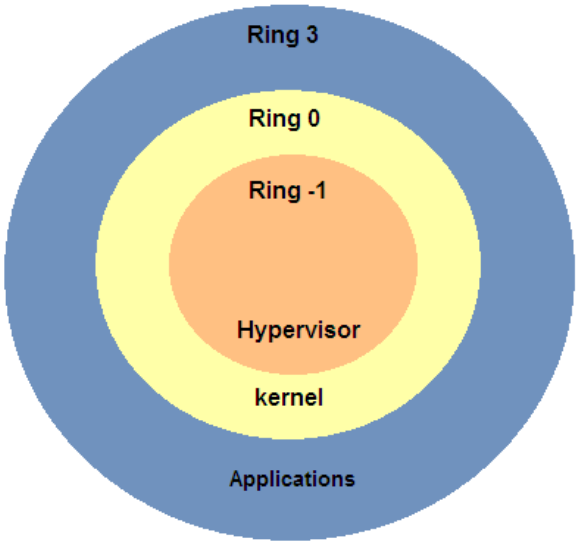

Figure 2.6: Adding a New Ring

2.5.2 Security Benefits

If the primary reason for the popularity of virtualization is server consolidation and the optimization of resources, security officers, also, may find some benefits of adopting this technology. The key benefit of virtualization is certainly *isolation*. Every VM is running in a separate sandbox which reduces the risk of information leakage and unauthorized access. Each VM has its own memory resources, I/O and processors. The sandboxing isolates VMs from each other and from other virtualized servers. This helps keeping the data safe and ensures their integrity, and allows also hosting different types of servers, dedicate them for a specefic application and optimize the system layer for this latter.

Isolation can be considered as the most important security-relevant property of hypervisors. If properly used, it guarantees that a malicious code in one VM does not affect the remaining VMs. Besides, resource usage of a VM can not affect the performance of other VMs. Isolation can also be used to separate applications : one can place vulnerable applications in a dedicated VM without caring about the security of the rest of VMs. If this VM is affected, the rest of the platform remains safe. Another security-related feature of hypervisors is their small codebase, compared to a modern OS, it is much more easier to ensure that hypervisor's code does not contain any bugs or flaws. This can be very useful for building TCBs (Trusted Computing Base) [34]. Moreover, in traditional OSes, the security mechanisms (IDS, anti-virus, firewall...) can be circumvented as soon as the OS is compromised. But in virtual environments, these security mechanisms can be moved out of the VM (in a dedicated VM) which makes them more resistant to attacks [35].

Companies usually demand to have several types of partitions: one for the production service, another one for testing, one for validation, and another for development. As sandboxing is total, a problem with one of the VMs will not have an effect on an other one. If a VM is compromised, one can kill it and restart it later from the last checkpoint. The security officer can also dedicate a virtual server for testing new updates before their installation. In terms of patch management, the company *Blue Lane* [36] has even developed a virtual patch system. The patches are not

VMware Vulns. by Year	Total Vulns	High Risk Vulns	Remote Vulns	Vulns in 1st Party Code	Vulns in 3rd Party Code
1999	1	1	0	1	0
2000	1	1	0	1	0
2001	2	0	0	2	0
2002	1	1	1	1	0
2003	9	5	5	5	4
2004	4	2	0	2	2
2005	10	5	5	4	6
2006	38	13	27	10	28
2007	34	18	19	22	12
Totals	100	46	57	48	52

Figure 2.7: Temporal Course of VMware Vulnerabilities Since 1999

directly applied on the physical server, but tested on the VM before its installation. This company has also recently developed a new software solution running on a VMWare virtual machine to secure servers located in other VMs. Virtual machines can also be used as virtual *honeypots* allowing the collection of information coming from hackers. This is called *crash-and-burn*. This technique offers the ability to keep an eye on malicious behaviors, test some codes and restore to a previous state of the system in case the VM crashes. Virtualization can also provide a greater security when surfing Internet. A Windows user with the VMware Player can start an instance of Linux equiped with the Firefox navigator and surf without being exposed to ewploits and vulnerabilities related to Windows or Internet Explorer navigator. Another advantage of virtualization is also the ability to have a remote access to a specific network without deploying a VPN (Virtual Private Network).

In the next section we will see that virtualization does not represent a perfect mean for securing systems and applications : many risks can be arised by using this technology.

2.5.3 Security Risks

With the growth of virtualization technologies, the security alerts related to this technology are increasing. Figure 2.7 shows the increasing number of vulnerabilities appearing in the VMWare VMM since 1999. Virtualization introduces new software layers that represent new areas that are exposed to attacks and which are quite complicated to manage. The direct access to hardware by these layers can also cause a lot of damage.

Three parts of the virtual architecture must be supervised as they provide a new playground for hackers. The hypervisor is certainly the most exposed one, because it makes the link between the hardware and VMs. The second sensitive part is the administrative platform, as it gives privileged access to all the virtual instances of the infrastructure, this platform is called *Dom 0* in the case of the Xen hypervisor or *Management Virtual Machine* in the case of *VMWare*. Finally, the dedicated chips to the virtual infrastructure (as Intel VT or AMD SVM) is also a great danger. They use a set of specific instructions that facilitate the implementation of multiple operating systems on one machine. These platforms can be exploited to get unauthorized access to system resources, through a rootkit for example. These attacks are particularly difficult to detect because they use lower software layers. Blue Pill [37], is one of these attacks, it was made public in 2006. In this attack the whole machine is virtualized by running a small hypervisor under it. The system can loose the references of the devices, the hardware interrupts and even the system time : every thing is intercepted and processed by the hypervisor. This gives the attacker

the opportunity to do his work without being detected since any detection system can be turned off by the hypervisor! Another type of attacks appeared in 2007 against the Xen hypervisor. A user in *domU* can execute commands on *dom0* while using the *pygrub* tool. *Pygrub* dedicates a bootloader to *domU* as in physical hosts. This vulnerability is very dangerous since *dom0* is a very priviliged domain and have a direct access to hardware. Some other vulnerabilities was also found in Xen, one of them allows a *domU* to break its isolation and can cause a local DoS (denial of Service). *VMWare* is much more exposed to attacks then *Xen*. Many vulnerabilities was discovered in this hypervisor. This is the result of the big number of associated products (virtual center, vSphere, workstation...). Almost all these vulnerabilities are about privilege escalation. The most critical ones can be exploited by an unauthenticated attacker from the Internet and can cause a complete compromise of data integrity and service availability.
Another threat in virtual environments is covert channels. It is a way to exploit a channel that was not dedicated for communication in order to communicate information [38]. In most cases, covert channels exist when two entities have access to a shared variable, the first one by reading from this variable and the second one by writing to this same variable. There is two kinds of covert channels : storage channels and timing channels. The first one modifies a stored object while the second type uses timed events in order to send information. One can reduce the number of covert channels in the system. Mandatory Access Control (MAC) [39] is very efficient against covert channels. It is the case when security classes will be assigned to users in order to limit the access of certain resources to some specific security classes. In virtual environments, the risk arising from covert channels is that users can use these channels to exchange information with each other wihtout using network connections [40]. Furthermore, Denial of Service attacks (DoS) can be more devastating when executed in a virtual environment then in any another one since subverting the hypervisor would lead to a complete subversion of the whole architecture and would give the attacker an unlimited control of all the VMs and their data. This is the reason why the hypervisor must be as secure as possible [42, 43]. To summarize, virtualization products are clearly not free from vulnerabilities [44, 45]. The impact of a vulnerability on a virtualized platform will be more devastating compared to a conventional architecture. Many countermeasures can be taken to prevent these vulnerabilities from being exploited and to reduce the attacks surface. On the other hand, the frequnecy of these vulnerablities is relatively low which gives time to security officers to design strong defence methods. In the next section, we will present some existing security solutions for these risks and discuss their efficiency against some security threats.

2.6 Some Existing Approaches

Much work have been done around securing virtualized systems. In this section, we will present some of the most important contributions in this field and conclude with a discussion of the pros. and cons. of every cited work. This section will not only include the presentation of some interesting papers but also a summary of some big projects around securing virtualized platforms.

2.6.1 XSM/FLASK for Xen

The Xen Security Modules (XSM) framework is a direct application of the Flask architecture [46, 47, 48] to the Xen hypervisor. This project was started by NSA (National Security Agency) (Flask is an OS security architecture that provides flexible support for security policies, the Flask architecture is now implemented in SELinux).

XSM is a generalized security framework for Xen, it creates general security interfaces and allows custom security functionality in modules. This makes the hypervisor able to support many security policy models at the same time. XSM comes also with the idea of decomposing the *domain 0 i.e* minimizing the importance of this domain by reducing its privileges and separating the hardware priviliges from domain ones. In addition, XSM gives the ability to partition resource allocation and control between domains. Some other modules in XSM implement media encryption, IP-filtering/routing and measurement and attestation functionalities. Besides, all the modules can be registered and linked in at boot, they may also register a security hypercall and a policy magic number to identify and load a policy from boot.

2.6.2 sHype

sHype is an implementation of the XSM modules. This project [98] is one of the most famous contributions in the field of secure hypervisors. It was developed by IBM research for the Xen hypervisor. This project consists of a security architecture that controls the sharing of resources among VMs according to formal security policies. The primary goal of sHype is to control of information flows between VMs. The architecture was designed to achieve medium assurance (Common Criteria EAL4 [50]) for hypervisor implementations. sHype supports a set of security functions: secure services, resource monitoring, access control between VMs, isolation of virtual resources, and TPM-based attestation. The mandatory access control enforces a formal security policy on information flow between VMs. sHype leverages existing isolation between virtual resources and extends it with MAC features. TPM-based attestation [51] provides the ability to generate and report runtime integrity measurements on the hypervisor and VMs. This enables remote systems to infer the integrity properties of the running system.

Besides, sHype uses a *reference monitor* that enforces, mandatory access control policies on inter-VM operations. A reference monitor is designed to ensure mediation of all security-sensitive operations, which enables a policy to authorize all such operations [53]. However, the reference monitor usually does not decide whether a subject can access an object. It only enforces the decision, which is often made elsewhere in the system.

The architecture of sHype consists of: (1) the policy manager maintaining the security policy; (2) the access control module (ACM) delivering authorization decisions according to the policy; and (3) and mediation hooks controlling access of VMs to shared virtual resources based on decisions returned by the ACM. The *policy manager* interacts with the ACM in order to establish a security policy or to help the ACM re-evaluate access control decisions. The *Access Control Module* (ACM) stores all security policy information locally in the hypervisor, and supports policy management through a privileged hypervisor call interface. This interface is access-controlled by a specialized hook and will only be accessible by policy-management-privileged domains. *Mediation hooks* are specialized access enforcement functions that guards access to a virtual resource by VMs. They enforce information flow constraints between VMs according to the security policy. These hooks determine access decisions with the ACM, enforce access control decisions and can determine VMs labels, access operation type etc, these information are useful for the access control. With these hooks, sHype minimizes the interaction with the core hypervisor.

Discussion

The main goal of sHype was to control all explicit information flows between VMs. So far, sHype has fulfilled this objective and has shown its efficiency in this area. In addition, it has shown promising results in ensuring the integrity of the system and preventing information leakage. On the other hand all these results was obtained with the Xen hypervisor, as sHype was originally

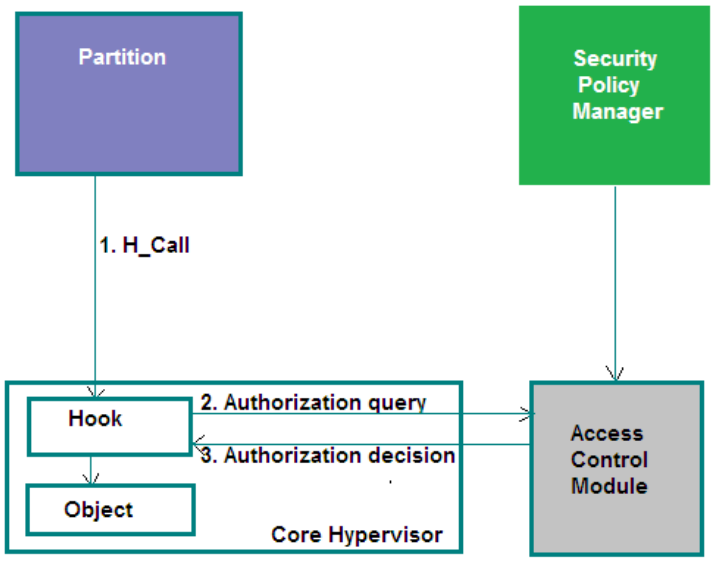

Figure 2.8: sHype Architecture

developed under Xen. This makes it unable to secure other hypervisors and becomes software and OS-dependant. For instance, VMWare and KVM [54] are gaining success, and it will be a pity that such a software does not support these virtual machine monitors. Besides, sHype can not be run under Windows or MAC OS X which makes it loose a huge umber of users. Another disadvantage of sHype is its deployment and its administration : this software is not adapted to simple users and needs some training before starting using it. In order to bypass these problems, we present in this thesis some portable solutions that are OS and VMM-undependant, and also very simple approaches that can be deployed easily by simple users. The last thing to say about sHype is its weakness against DoS attacks since there is no alerting mechanism that can detect that a VM is not responding : this can be done also by our approach.

2.6.3 VAX VMM security kernel

This project [55] was one of the first attempts to design a secure hypervisor. VAX aims to develop a security kernel which was carried out on the virtual address extension (VAX) architecture designed by Digital Equipment Corporation during the 1970s. This is why the VMM security kernel of Karger et al. is often also called the VAX security kernel. VAX supports DAC and MAC for all VMs. It enforces the Bell-La Padula and Biba models for integrity. Furthermore, the security kernel was carefully designed in order to prevent covert-channels. Self-Virtualization is also supported by VAX : it is the ability of a virtual machine monitor to run in one of its own VMs and create second-level VMs which is very useful for developing and debugging the VMM itself.

In VAX, the user has to authenticate herself to the VAX VMM before accessing any VM. For this purpose the VAX hypervisor offers a trusted process running in the kernel called the *Server*. This process only executes verified machine code and does not accept any user-written code. If a user wants to interact with the VAX hypervisor, a trusted path between a server process and the user is established. The server provides commands that allow the user to connect to a VM

depending on his access rights. In case the user has the necessary rights to connect to a VM another trusted path is established between the user and the VM, allowing him to interact with the OS running in the VM. VAX has shown a good performance which is extremely important, because getting good performance is very hard. It requires detailed analysis of what portions of the kernel are performance-critical and a willingness to redesign those portions for performance and possibly re-code them in assembly language or to provide microcode performance assistance.

Discussion

It is true that the VAX hypervisor is an old project, but this does not make it unimportant : in fact this project was a perfect example for the projects started later. It has clarified many important things about the security of hypervisors and has stressed some relevant points that have to be treated carefully to design a secure hypervisor. Besides, VAX represents a good implementation for security policy models such as Bell-La Padula and Biba.

2.6.4 Terra

Terra [56] is a virtualization-based architecture for trusted computing. This project introduces the *Trusted Virtual Machine Monitor (TVMM)*, that partitions a tamper-resistant hardware platform into multiple, isolated virtual machines (VM), providing the appearance of multiple boxes on a single, general-purpose platform. VMs are classified into *open-box VMs* and *closed-box VMs*. *Open-box VMs* are not different from ordinary VMs running ont top of Xen for example : no special security mechanisms are implemented for this kind of VMs. *Closed-box VMs* implement the semantics of a closed-box platform. Their content cannot be manipulated or inspected by the administrator of the system. Only the creator of this VM can access it. This is achieved through the use of three main security mechanisms : (1) *Attestation* which allows an application running in a *closed-box VM* to identify itself to a remote party, this can be done through cryptographic mechanisms. Then, a *chain of trust* is established starting from this application and ending at this remote party. (2) *Root secure*: even the platform administrator cannot break the isolation of a closed-box VM. (3) *Trusted Path*: this is essential for building secure applications. In the TVMM, users can easily identify VMs that they are communicating with, and each VM is able to ensure that it is interacting with a human user. This ensures the privacy of communications between VMs and users and prevents snooping by malicious applications.

Discussion

The main goal of Terra was to make the communication between users and VMs as secure as possible. The notion of open/closed box VMs prevents some families of attacks against hypervisors. The remote attestation ensures a secure channel of communication between the different parties. On the other hand, the deployment of Terra is still difficult for simple users and needs to be more intuitive. In addition, terra does not provide access control mechanisms such as MAC which seems to be a serious weakness of this architecture. Therefore the designers of Terra decided to make it as flexible as possible by minimizing the control of information flows which weakens the overall security of this software. We overcome some of these problems in our work by providing very easy-to-deploy software and strong formal security policies that prevents families of attacks from being executed.

2.6.5 Other Contributions

In [57], Bleikert *et al.* studied the automated information flow analysis of heterogeneous virtualized infrastructures. They proposed an analysis system that performs a static information flow analysis based on graph traversal. The system unifies diverse virtualization environments in a graph representation and computes the transitive closure of information flow and isolation rules over the graph and diagnoses isolation breaches from that automatically. The analysis is based on explicitly specified trust rules. The implemented tool is independant from the vendor and can unify different systems suc as : Xen, KVM, VMWare and PowerVM. The static analysis covers all the resources : hardware, hypervisor, storage and network resources. This technique is applicable to the isolation analysis of complex configurations of large virtualized datacenters that include heterogeneous server hardware, different VMMs, and many virtual (and physical) networking and storage resources. This approach is interesting for static analysis. However it does not enforce any kind of security policy and is only useful in the case of large-scale infrastructures with a diversity of underlying platforms. The core hypervisor does not take advantage from this technique since it focuses only on flow analysis. Another point is that the analysis is restricted to a binary decision, whether information flows or not, and does not support *traffic types*.

NetTop [90] provides infrastructure for controlling information flows and resource sharing between VMs. While the granularity level in these systems is a VM, we focus in our work at the granularity of a process.

In [52], Kurniadi *et al.* use virtual machine monitors for implementing honeypots. This is a different use of virtualized systems, but shows that hypervisors can also be useful for experimentation, testing and diagnosis. The authors implement a VMM-based intrusion detection and monitoring system for collecting information about attacks on honeypots. Their first step was to implement a sensor mechanism that monitors honeypots for intrusions by dynamically rewriting the binary of a running kernel image. Then, they compared the performance impact on three implementations built on UML (User Mode Linux) and Xen. The third step was to apply this mechanism to honeypots that are connected to Internet. Finally, they analysed and classified the detected attacks. Whereas this approach is very useful for diagnosis, the implemented sensors work only on specific platforms and do not report the detected attacks to an IDS for example to do the forensic which is very important for this kind of approaches. In this thesis we propose a sensor-based approach for intrusion detection but our goal is to secure the virtualized platform and not to implement honeypots, the advantage of our implementation is that all the events and alerts can be saved on the disc, then the security officer can access the reported events and study their impact on the system.

ReHype [32] is a system implemented on top of the Xen hypervisor that allows the recovery from hypervisor failures. This system is able to preserve the state of running VMs while booting a new instance of the hypervisor. Besides, it can protect the recovered hypervisor, resolve inconsistencies between different parts of hypervisor state as well as between the hypervisor and VMs and between the hypervisor and the hardware. The authors identified the specific sources of state corruptions and inconsistencies, determined which of those are most likely to prevent successful recovery, and devised mechanisms to overcome these problems. Recovery is very important in virtualized systems and ReHype represents a very efficient tool that implements this feature.

Another interesting contribution is the *BitVisor* [67] hypervisor. BitVisor implements a new architecture called *parapass-through* (see Figure 2.9). This latter allows most of the I/O access from guest VMs to pass-through the hypervisor and enforces storage encryption of ATA devices.

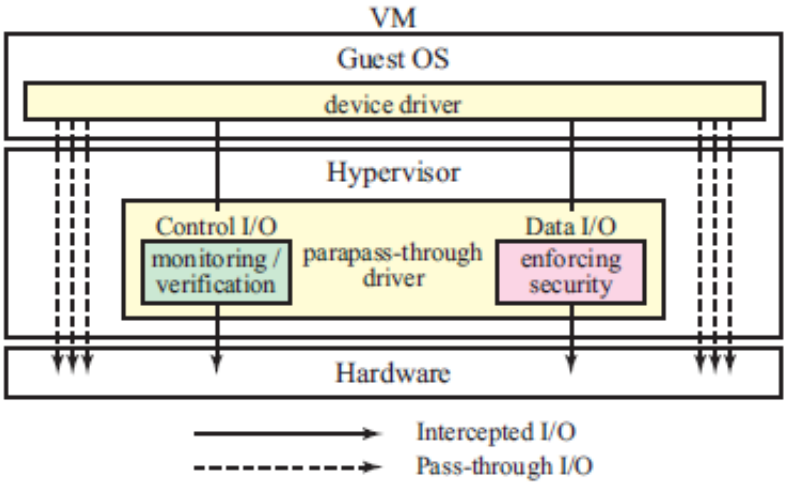

Figure 2.9: The *para* pass-through architecture (source : [67])

If all the access is pass-through, the hypervisor is almost useless. Different from fully pass-through access, para pass-through hypervisors intercept a part of access to (1) protect hypervisors from the guest OS, and (2) enforce security functionalities. The access to be intercepted includes memory access and I/O access. Intercepting memory access is necessary to protect memory regions of the hypervisor and handle memory-mapped I/Os (MMIOs). Intercepting I/O access is necessary to protect the hypervisor and enforce security functionalities upon the I/Os for specific devices.

Chapter 3

Securing Virtual Machines

3.1 Introduction

Virtual Machine technology is going mainstream. Motivated by cost savings, server consolidation and disaster recovery. IT organizations are changing the way they deploy applications and desktops. But industry pundits agree that full-on deployment of virtual machines has been impeded by a critical weakness: security. Traditional security architectures and products are inadequate for this new topology due to its specific architecture and security requirements. Many aspects of virtual platforms have to be taken into consideration when designing dedicated security solutions. It is more challenging to protect a virtualized system with 10 virtual machines than trying to secure only one isolated physical machine.

We introduce in this chapter a new idea for securing virtualized platforms. It is based on the notion of decentralized supervision in physical networks and adapts it to virtualized systems. Our objectif is to secure all running VMs and protect them against internal and external attacks, reduce the cost of this supervison mechanism and centralize event logging. Our approach is cost-effective, efficient against families of attacks and have the advanatge of isolating the defense system (which is an IDS in this case) and protecting it against malicious users. We design and implement our approach on top of the the Xen hypervisor [12] using the Orchids IDS [91] and the SELinux auditd daemon. This approach has many advantages and is quite efficient against many security threats but has also some limits that we discuss at the end of this chapter and present more in-depth discussion in the next chapter.

3.2 Related Work

Much work has been done on enhancing the security of computer systems. Most implemented, host-based IDS run a program for security on the same operating system (OS) as protected programs and potential malware. This may be simply necessary, as with Janus [86], Systrace [93], Sekar *et al.*'s finite-state automaton learning system [99], or Piga-IDS [79], where the IDS must intercept and check each system call before execution. Call this an *interception* architecture: each system call is first checked for conformance against a security policy by the IDS; if the call is validated, then it is executed, otherwise the IDS forces the call to return immediately with an error code, without executing the call.

A *virtualized* system such as Xen [119], VirtualBox [115], VMWare [116], or QEmu [95] allows one to emulate one or several so-called *guest* operating systems (OS) in one or several *virtual*

machines (VM). The different VMs execute as though they were physically distinct machines, and can communicate through ordinary network connections (possibly emulated in software). The various VMs run under the control of a so-called *virtual machine monitor* (VMM) or *hypervisor*, which one can think of as being a thin, highly-privileged layer between the hardware and the VMs. See Figure 2.1, which is perhaps more typical of Xen than of the other cited hypervisors. The solid arrows are meant to represent the flow of control during system calls. When a guest OS makes a system call, its hardware layer is emulated through calls to the hypervisor. The hypervisor then calls the actual hardware drivers (or emulations thereof) implemented in a specific, high privilege VM called *domain zero*. Domain zero is the only VM to have access to the actual hardware, but is also responsible for administering the other VMs, in particular killing VMs, creating new VMs, or changing the emulated hardware interface presented to the VMs.

In recent years, virtualization has been seen by several as an opportunity for enforcing better security. The fact that two distinct VMs indeed run in separate sandboxes was indeed brought forward as an argument in this direction. However, one should realize that there is no conceptual distinction, from the point of view of protection, between having a high privilege VMM and lower-privileged VMs, and using a standard Unix operating system with a high privilege kernel and lower-privileged processes. Local-to-root exploits on Unix are bound to be imitated in the form of attacks that would allow one process running in one VM to gain control of the full VMM, in particular of the full hardware abstraction layer presented to the VMs. Indeed, this is exactly what has started to appear, with Wojtczuk's attack notably [117].

Some of the recent security solutions using virtualization are sHype [98] and NetTop [90]. They provide infrastructure for controlling information flows and resource sharing between VMs. While the granularity level in these systems is a VM, our system controls execution at the granularity of a process.

Livewire [85] is an intrusion detection system that controls the behavior of a VM from the outside of the VM. Livewire uses knowledge of the guest OS to understand the behavior in a monitored VM. Livewire's VMM intercepts only write accesses to a non-writable memory area and accesses to network devices. On the other hand, our architecture can intercept and control all system calls invoked in target VMs.

G. W. Dunlap describes an experience of use of virtual machines for the security of systems [68]. The proposal defines an intermediate layer between the monitor and the host system, called Revirt. This layer captures the data sent through the syslog process (the standard UNIX logging daemon) of the virtual machine and sends it to the host system for recording and later analysis. However, if the virtual system is compromised, the log messages can be manipulated by the invader and consequently are no more reliable.

Stefan Berger describes the trusted computing in virtual machine [69]. By virtualization the TPM chipset, a single TMP chipset can provide the trusted computing service for each VM on the same hardware platform.

In [92], Onoue *et al.* propose a security system that controls the execution of processes from the outside of VMs. It consists of a modified VMM and a program running in a trusted VM. The system intercepts system calls invoked in a monitored VM and controls the execution according to a security policy. Thus, this is a an interception system. To fill the semantic gap between low-level events and high-level behavior, the system uses knowledge of the structure of a given operating system kernel. The user creates this knowledge with a tool when recompiling the OS. In contrast, we do not need to rebuild the OS, and only need to rely on standard event-reporting daemons such as `auditd`, which comes with SELinux [113], but is an otherwise independent component.

3.3 System Supervision In Virtual Environments

We present in this section some interesting approaches for supervising system execution and detecting malicious behaviors in virtual environments. Some approaches achieve local supervison, *i.e* every VM is equiped with necessary mechanisms for detecting/stopping attacks. Other ones implement decentralized supervision where only one remote VM contains security mechanisms that are able to monitor the whole virtualized system and prevent attacks.

In most VMM implementations many security approaches require the ability to monitor frequently executing events, such as host-based intrusion detection systems that intercept every system call throughout the system, LSM (Linux Security Module) [82] and SELinux that hook into a large number of kernel events to enforce specific security policies, or even instruction-level monitoring used by several offline analysis approaches. Due to the overhead involved in out-of-VM monitoring, many such approaches either are not designed for production systems. While keeping a monitor inside the VM can be efficient, the key challenge is to ensure at least the same level of security achieved by an out-of-VM approach.

3.3.1 Local Supervision Approaches

Among the various approaches proposed for local VM supervison in the late 10 years, SIM (Secure In-VM Monitoring) [83] is one of the most efficient and low-cost techniques that aims to protect the VMM and VMs. In SIM the authors utilize contemporary hardware memory protection and hardware virtualization features available in recent processors to create a hypervisor protected address space where a monitor can execute and access data in native speeds and to which execution is transferred in a controlled manner that does not require hypervisor involvement. Two important properties are ensured by this technique : (1) Fast invocation : where invoking a monitor handler should not involve any privilege level change. (2) Data read/write at native speed : the monitor code should be able to read and write any system data at native speed without any hypervisor intervention. The main feature of this approach is that the code of the monitor is isolated and protected by the idea of having two adress spaces : a trusted and an untrusted adress space. The switching from a space to another can be done without the intervention of the hypervisor. Something that arises the performance of the whole system. While this approach guarantees the efficiency of the monitoring and the detection pf policy violations, no global view of the system is given by the current implementation which may reduce the intervention ability of the administrator in case of network attacks or complex attacks where many VMs are involved.

Another interesting approach XSM/FLASK (detailed in chapter 2). This approach is provided by the Xen hypervisor which implements a security framework called XSM, and FLASK is an implementation of a security model using this framework (at the time of writing, it is the only one). FLASK defines a mandatory access control policy providing fine-grained controls over Xen domains, allowing the policy writer to define what interactions between domains, devices, and the hypervisor are permitted. This approach offers for instance the ability to prevent two domains from communicating via event channels or grants, controls which domains can use device passthrough (and which devices), can restrict or audit operations performed by privileged domains and finally prevents a privileged domain from arbitrarily mapping pages from other domains. Some of these examples require Dom0 Disaggregation to be useful, since the domain build process requires the ability to write to the new domain's memory. On the other hand, this approach has many limits that we present in the next paragraph.

3.3.2 Disadvantages of Local Supervision

Despite the high quality of protection that local supervison approaches give to VMMs, they still have many disadvantages and weaknesses. First, implementing a local approach means that every VM is equiped with necessary mechanisms for monitoring and detection. This reduces consequently the ability of the defense system to intervene in remote VMs in case of cross-VM attacks, and reduces the general view of the administrator of the whole virtualized platform. The latest point is of interest because of the increasing complexity of virtual environments and the need to have the largest view of the system with the most precise details about each VM/component. Besides, these same security mechanisms need to have access rights to remote VMs in order to communicate and send defense commands in case any attack is detected. This advantage is not given by local approaches. Moreover, some attacks called network attacks can escape this kind of approaches. Owing to the complexities of the virtual environment, network attacks become even harder-to-detect when virtual machines are introduced to the network. Besides, implementing local supervision does not help the system administrator have easy and efficient administration tasks. In fact, local policies need some configuration from time to time, and assuming the complexity of such systems, the administrator does not have enough tools and mechanisms to share upgrade with all VMs in such local approaches. For example, a VM's configuration is stored as a single file, which makes it easier for an attacker to copy or delete these files and potentially steal a whole VM (and its stored information). This is due to the limited system view given to the administrator. Another disadvantage of this approach is its inefficiency against Cross-VM vulnerabilities that come from the co-residence of VMs which makes information easy to exfiltrate across VM boundary. For instance, in Cross-VM attacks, the attacker sends HTTP requests to the target VM and observe correlation with cache utilization or even obtain and compare Xen Dom0 address. A Cross-VM attack can then occur corrupting the integrity, confidentiality and availability of the attacked VM. To detect this kind of attacks, the system administrator needs several technologies and methods that are not available in simple local supervision (network filtering, network monitoring, global policies...).

To summarize, we can say that local supervision approaches are not the most convenient approach for securing virtualized systems. Since they are unable to prevent many vulnerabilities and detect different malicious behaviors that need larger vision of the system. We present in the next section another approach that implements decentralized supervision, we will then compare the two approaches and propose our own architecture/imlementation for securing virtual machines.

3.3.3 Decentralized Supervision Approaches

While local interception architectures have the advantage of allowing the IDS to counter any attack just before they are completed. This way, and assuming the security policy that the IDS enforces is sufficiently complete, no attack ever succeeds on S that would make reveal or alter sensitive data, make it unstable, or leave a backdoor open (by which we also include trojans and bots).

Decentralized approaches the IDS is meant to work in a *decentralized* setting. In this case, the IDS does not run on the same host as the supervised host, S. While in a local interception architecture, the IDS would run as a process on S itself, in a decentralized setting only a small so-called *sensor* running on S collects relevant events on S and sends them through some network link to the IDS, which runs on another, dedicated host M.

Decentralized architectures (see Figure 3.1) make the IDS more resistant to attacks on S (which may be any of $S_1, \ldots, S_4$ in the figure): to kill the IDS, one would have to attack the supervision machine M, but M is meant to only execute the IDS and no other application, and

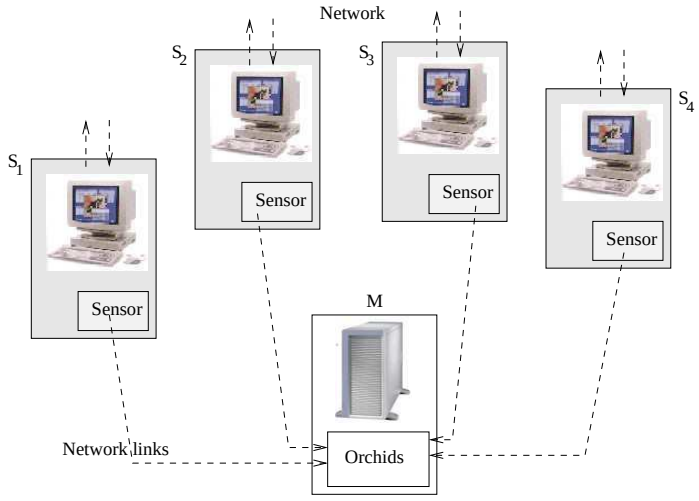

Figure 3.1: Decentralized Supervision

has only limited network connectivity. In addition to the link from S to M used to report events to the IDS, we also usually have a (secure) link from M to S, allowing the IDS to issue commands to retaliate to attacks on S. While this may take time (e.g., some tens or hundreds of milliseconds on a LAN), this sometimes has the advantage to let the IDS *learn* about intruder behavior once they have completed an attack. This is important for forensics.

Decentralized architectures are also not limited to supervising just one host S. It is particularly interesting to let the supervision machine M collect events from several different hosts at once, from network equipment (routers, hubs, etc., typically through logs or MIB SNMP calls), and correlate between them, turning the IDS into a mix between host-based and network-based IDS.

We shall argue in the next section that one can simulate such a decentralized architecture, at minimal cost, on a single machine, using modern virtualization technology. We shall also see that this has some additional advantages.

3.4 Proposed Architecture

As explained earlier, local interception approaches are vulnerable to local attacks, because the intruder can disable or tamper them. Thus, monitoring data coming from a compromised system cannot be considered reliable. The isolation offered by virtual machines provides a solution to this problem. The proposal presented here allows building more reliable virtualized platforms for intrusion detection.

Our proposal's main idea is to encapsulate both the systems to monitor and the surveillance system inside virtual machines. The intrusion detection and response mechanisms are implemented outside the virtual machine, i.e. out of reach of intruders. Figure 3.2 illustrates the main components of the proposed architecture.

We run a fast, modern IPS such as Orchids [91, 87] in another VM to monitor, and react against, security breaches that may happen on the users' environment in each of the guest OSes present in a virtualized system.

One can see this architecture as an implementation of a decentralized supervision architecture on a single physical host.

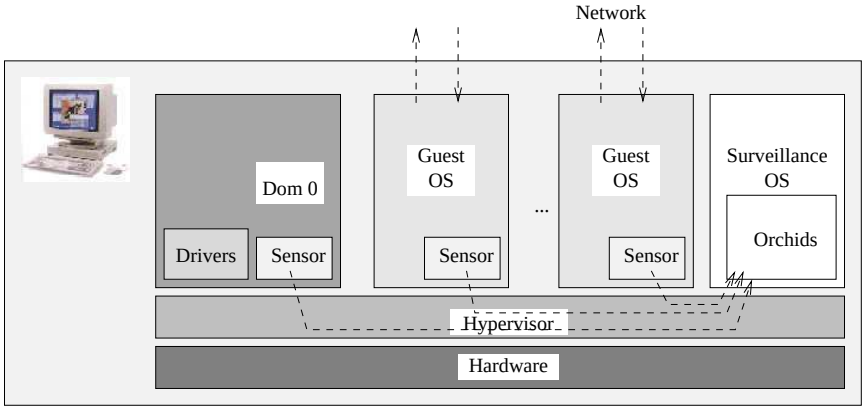

Figure 3.2: Proposed Architecture

We argue that this solution has several advantages. First, there is a clear advantage in terms of cost, compared to the decentralized architecture: we save the cost of the additional supervision host M.

Second, compared to a standard, unsupervised OS, the user does not need to change her usual environment, or to install any new security package. Only a small sensor has to run on her virtual machine to report events to Orchids. Orchids accepts events from a variety of sensors. In our current implementation, each guest OS reports sequences of system calls through the standard `auditd` daemon, a component of SELinux [113], which one can even run without the need for installing or running SELinux itself. (Earlier, we used Snare, however this now seems obsolescent.) Linux auditd sensor is a built-in mechanism in the kernel, which allows one to intercept changes to monitored files and write them to a log on the disk or send them to a local socket. Auditd intercepts almost all system calls and gives a detailed summary in real time of the performed system calls. we can let auditd supervise some specific users or system calls depending on what we want to audit.

The bulk of the supervision effort is effected in a different VM, thus reducing the installation effort to editing a few configuration files, to describe the connections between the guest OSes and the supervision OS mainly. In particular, we do not need to recompile any OS kernel with our architecture, except possibly to make sure that `auditd` is installed and activated.

A third advantage, compared with interception architectures, and which we naturally share with decentralized architectures, is that isolating the IPS in its own VM makes it resistant to attacks from the outside. Indeed, Orchids runs in a VM that has no other network connection to the outside world than those it requires to monitor the guest OSes, and which runs no other application that could possibly introduce local vulnerabilities.

Orchids should have high privileges to be able to retaliate to attacks on each guest OS. For example, we use `ssh` connections between Orchids and each VM kernel to be able to kill offending processes or disable offending user accounts. (The necessary local network links, running in the opposite direction as the sensor-to-Orchids event reporting links shown in Figure 4.7, are not drawn.)

The Orchids detection system recognizes scenarios by simulating known finite automata, from a given event flow. This method allows the writing of powerful stateful rules suitable for intrusion detection.

Orchids is composed of five main parts: a set of rule definitions (in a dedicated specification language), a rule compiler which translates rule definitions into an internal automata representa-

tion, a set compiled rules which is the knowledge base of the whole system, a massively parallel virtual machine which simulates non-deterministic finite automata, and a set of input modules which decodes data incoming from external sources.

Next, we cannot expect an ordinary user to manage her own machine, or, for that matter, to keep an attack signature base up to date. Although Orchids requires rather few signatures, since one signature can match several attacks (including some zero-day attacks [87]), Orchids is still fundamentally a misuse intrusion prevention system, and requires some maintenance, if only to write new rules for new families of attacks. A standard solution to this problem is to install a link between the application, here Orchids, and a trusted server, with a regularly triggered task that inquires about security updates from the server. We do not wish to let the Orchids virtual machine communicate along any link with the outside world, if possible. Trusted servers can be hacked, and in any case emitting security updates requires an infrastructure, and resources.

However, running on a virtualized architecture offers additional benefits. One of them is that Orchids can now ask domain zero to *kill* an entire VM. This is necessary when a guest OS has been subject to an attack with consequences that we cannot assess precisely. For example, the `do_brk()` attack [114] and its siblings, or the `vmsplice()` attack [94] allow the attacker not just to gain root access, but direct access to the *kernel*. Note that this means, for example, that the attacker has immediate access to the whole process table, as well as to the memory zones of all the processes. While current exploits seem not to have used this opportunity, such attack vectors in principle allow an attacker to become completely stealthy, e.g., by making its own processes invisible to the OS. In this case, the OS is essentially in an unpredictable state.

The important point is that we can always revert any guest OS to a previous, safe state, using virtualization. Indeed, each VM can be *checkpointed*, i.e., one can save the complete instantaneous state of a given VM on disk, including processes, network connections, signals. Assuming that we checkpoint each VM at regular intervals, it is then feasible to have Orchids retaliate by killing a VM in extreme cases and replacing it by an earlier, safe checkpoint.

Orchids can also detect VMs that have died because of fast denial-of-service attacks (e.g., the double `listen()` attack [81], which causes instant kernel lock-up), by pinging each VM at regular intervals: in this case, too, Orchids can kill the VM and reinstall a previous checkpoint. We react similarly to attacks on guest OSes that are suspected of having succeeded in getting kernel privileges and of, say, disabling the local `auditd` daemon.

Killing VMs and restoring checkpoints is clearly something that we cannot afford with physical hosts instead of VMs.

It would be tempting to allow Orchids to run *inside* domain zero to do so. Instead, we run Orchids in a separate guest OS, with another `ssh` connection to issue VM administration commands to be executed by a shell in domain zero. I.e., we make domain zero *delegate* surveillance to a separate VM running Orchids, while the latter trusts domain zero to administer the other guest VMs. We do so in order to sandbox each from the other one. Although we have taken precautions against this prospect, there is still a possibility that a wily attacker would manage to cause denial-of-service attacks on Orchids by crafting events causing blow-up in the internal Orchids surveillance algorithm (see [87]), and we don't want this to cause the collapse of the whole host. Conversely, if domain zero itself is under attack, we would like Orchids to be able to detect this and react against it.

To our knowledge, this simple architecture has not been put forward in previous publications, although some proposals already consider managing the security of virtualized architectures, as we have discussed earlier.

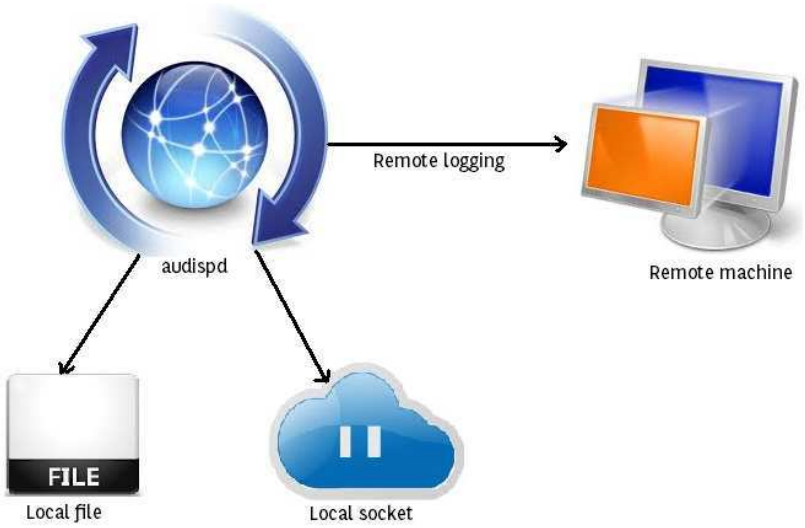

Figure 3.3: The Implemented Remote Logging

3.5 Remote Logging

As explained earlier, we equip every virtual machine with the SElinux auditd sensor. This daemon captures system calls according to a configuration file containing details about what we want to audit. To accomplish its mission, the auditd daemon relies on an engine called *audispd*. This is the dispatcher of the daemon, it is responsible of sending the reported events to the specified targets. These targets are either a local file or a local socket, and audispd is unable to report to a remote target.

Since our idea is to preserve the decentralisation criterion of our architecture, we needed to make audispd able to report to remote targets via the network. To deal with this, we designed and implemented a new functionality in the auditd dispatcher. This functionality makes audispd able to report events via the virtual network (TCP channel) to a remote target (see figure 3.3).

Besides,and from the IDS side, the need was to make Orchids behave like a server that receives information from different hosts and reacts according to the correlation of the events if an attack matches. The reaction is perfomed through the network by sending commands that are able to kill the offending remote processes and sometimes by asking Dom0 to completely stop the VM and restart from an early checkpoint. This can be done in case of fast Dos attacks that can freeze the whole VM.

3.6 Discussion

The proposed architecture has pros. and cons.: first, compared to other architectures, this one is very easy to deploy. The sensor comes with almost all 2.6 Linux kernels and no further configuration is needed except adding the system calls that one wants to audit. From the supervision Vm side, one have just to install Orchdis which is preconfigured to work with auditd. This makes the system administrator's life easier.

Second, we argue that this approach has the advantage of working with a powerful hypervi-

sor which is Xen. Indeed, Xen represents a thin hypervisor model consisting of only 2 MB of executable, relying on service domains for functionality, needs no device drivers and keeps domains/guests isolated. These characteristics can not be found in other virtualization tools such as VMware ESX which needs device drivers and a base of management where hardware support depends on VMware created drivers.

Another advantage of this approach is the fact that defense mechanisms are already implemented in Orchids and we benefit from this functionality and make it work for all the VMs. This gives another dimension to our IDS. This point is of importance because designing a complete and efficient solution with defense mechanisms, most of the time, is not an easy task.

On the other hand, one can notice that our approach relies on the network for communicating information between the IDS and the different VMs. This can be a real source of problems. An attacker located in a simple VM can try to break the access rules to the configuration files of auditd and stop the remote logging mechanism. To deal with this, Orchids can easily detect that one VM is active but is not reporting. For now the remedy to this problem is not yet implemented but we feel that the best solution is to report this issue to the administrator who will try to diagnose this VM and restart the sensor. If the problem persists, Orchdis can kill the VM and restart later from a safe checkpoint (free from vulnerabilities).

Another weakness of relying on the network is the latency related to the network (the term latency refers to any of several kinds of delays typically incurred in processing of network data. A so-called low latency network connection is one that generally experiences small delay times, while a high latency connection generally suffers from long delays). Actually we are unable to react efficiently according to fast DoS attacks that can rapidly freeze the VM even before the reception of the reported events by Orchids.

The main weakness of our approach is the fact that we rely on the reported system calls to detect intrusions. The problem is that some new attacks are stealthy and undetectable using this approach. For instance the Wojtczuk's attack [117] on the Xen hypervisor is completely undetectable by our approach. The objective of our thesis was not to detect this specific attack but at least we try to offer an easy way to avoid the dammage caused by this attack. In fact we will make sure that the attack can not be run at all (see chapter 4).

Another problem related to our approach and that we adress in the next chapter is the absence of a specific security policy that can be dedicated to this environment. We feel that a rigourous access control policy can be complementary to our supervision/detection approach. More details about how we adress this problem will be given later.

3.7 Conclusion

In this chapter, we have presented a new architecture for intrusion detection that simulates decentralized supervision on one single host. Our primary aim was to secure running virtual machines against attacks by deploying Orchids and sensors reporting at real time to it. This architecture was implemented on top of the Xen hypervisor and its main advantage is cost saving. Regarding the effectiveness of the detection mechanisms, many improvements can be brought to our implementation. Much work can be done on DoS attacks detection, on securing commmunication channels and especially on optimizing the content of the reported events. This can be done by improving the way that auditd logs the captured events. As further work, it would be interesting to extend this implementation to other interesting vrtualization solutions such as KVM [41] or VMWare. It would be also challenging to explore ways to avoid killing VMs in case of DoS attacks in order to preserve a good level of the service continuity.

Chapter 4

Protecting Sensitive Resources

4.1 Introduction

In chapter 3, we have presented a new approach for securing virtual machines. This idea is based on a decentralized intrusion detection mechanism ensured by the Orchids IDS and the *auditd* sensor. Despite the advantages that such an approach can offer, it remains unable to protect sensitive resources efficiently due to the lack of a security policy strategy.

In this chapter we introduce a new way to model security policies and deploy them. Our primary goal will be to protect sensitive resources such as the domain0, and at the same time prevent some stealthy attacks that we can not detect. We introduce a new language for modelling security policies accompanied with a procedure for the automatic translation of policies into automata representing attacks signatures that enrich the IDS signature base.

4.2 Related Work

In this section, we present two approaches that are similar to our proposal. The first one is *Proof Carrying Code (PCC)*. PCC comes with the idea that end-users become able to verify security properties about an application via a formal proof that accompanies the executable code. The user can then decide if the application is safe by comapring the result of the verification to the local security policy.

The second approach is *Model Carrying Code (MCC)* where end-users can profit from a fully automated verification procedure to determine if a downloaded code satisfies their security policies. Alternatively, an automated procedure can sift through a catalog of acceptable policies to identify one that is compatible with the model. In the next two sections, we give a brief presentation of these approaches in order to clarify the idea of verifying models against security policies, this helps understand our approach which does not have exactly the same goal but shares many details with these methods especially in the modelling and verification part.

4.2.1 Proof Carrying Code

Overview Proof-Carrying Code (PCC) [89] reveals many advantages for safe execution of untrusted code. The technique needs that the producer and the consumer of the code perform some necessary actions : first, the consumer needs to establish a set of rules (safety) that guarantees the safe behaviour of a program. Then, the producer has to create a formal proof for the untrusted code. This proof is used by the receiver of the code as an entry to his *proof validator*

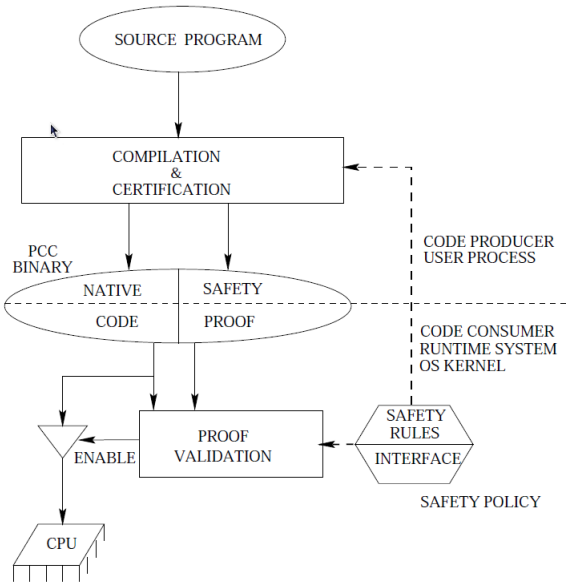

Figure 4.1: Proof Carrying Code [89]

(given by the producer) in order to check that the code is safe. PCC has many uses in systems whose trusted computing base is dynamic : extensible operating systems, Internet browsers, active network nodes and safety-critical embedded controllers. These examples need most of the time mobile code or regular updates.

Despite the large amount of effort in establishing and formally proving the safety of the untrusted code, almost the entire burden of doing this is on the code producer. The code consumer, on the other hand, has only to perform a fast, simple, and easy-to-trust proof-checking process. The trustworthiness of the proof-checker is an important advantage over approaches that involve the use of complex compilers or interpreters in the code consumer.

The consumer does not care about the nature of the proof. The proofs could be generated by hand, but sometimes it is necessary to rely on a theorem prover. Besides, the consumer does not have to trust the proof-generation process. Moreover, any modification (either accidental or malicious) will result in one of these outcomes : (1) the proof is not valid, the program is not accepted, (2) the proof is valid but is not a safety one, so it will be rejected, (3) the proof is valid despite the modifications, in this case the guarantee of safety will hold.

Another interesting feature of PCC is the continious checking of intrinsic properties of the code without caring about its origin thus cryptography mechanisms are not needed. In this sense the proagrams are self-certifying. On the other hand, the static verification of the untrusted code before its execution saves time and detects hazardous operations early thus avoiding the situations when the code consumer must kill the untrusted process after it has acquired resources or modified state.

Discussion Despite the elegant design of PCC and its easy comprehension, this method is very difficult to implement efficiently. First, proofs are not easy to encode because trivial encoding of properties of programs is very large. Second, the verification part of the proof is not an easy task because it needs a small, fast and independant checker, also, the proofs must be terse. Finally, the producer have to provide a proof that is fully related to the real execution of the program,

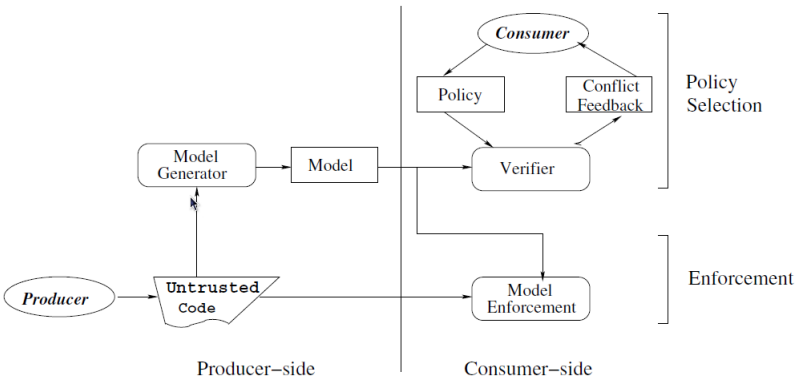

Figure 4.2: Model Carrying Code [103]

something that is not totally ensured. In our approach, we do not provide proofs, but only safety properties at the consumer side, we do not care about their translation into models, because we rely on an automatic tool. Then the verification is done by a model-checker (Orchids) that verifies these properties at real time.

4.2.2 Model Carrying Code

Overview MCC [103] introduces program behavioral models (see Figure 4.2) which help bridge the semantic gap between low-level binary code and high-level security policies. These models capture security-related properties of the code, but do not capture aspects of the code that pertain only to its functional correctness. The model is stated in terms of the security-relevant operations made by the code, the arguments of these operations, and the sequencing relationships among them. These operations correspond to system calls, but alternatives such as function calls are also possible. While models can be created manually, doing so would be a time-consuming process that would affect the usability of the approach. Therefore, the authors developed a model extraction approach that can automatically generate the required models. Since the model extraction takes place at the producer end, it can operate on source code rather than binary code. It can also benefit from the test suites developed by the code producer to test his/her source code. The consumer wants to be assured that the code will satisfy a security policy selected by him/her. The use of a security behavior model enables to decompose this assurance argument into two parts: policy satisfaction which checks whether the model satisfies the policy, and model safety which checks if the model captures a safe approximation of program behavior.

In more details, a *producer* generates both the program D to be downloaded (e.g., the device driver), and a *model* of it, M. The *consumer* checks the model against a local *policy* P. Instead of merely rejecting the program D if its model M does not satisfy, the consumer computes an *enforcement model* M' that satisfies both M and P, and generates a monitor that checks whether P satisfies M' at run-time. Any violation is flagged and reported.

In MCC, models, as well as policies and enforcement models, are taken to be extended finite-state automata (EFSA), i.e., finite state automata augmented with finitely many state variables meant to hold values over some fixed domain. A typical example, taken from op. cit., is the EFSA of Figure 4.3. This is meant to describe the normal executions of D as doing a series of

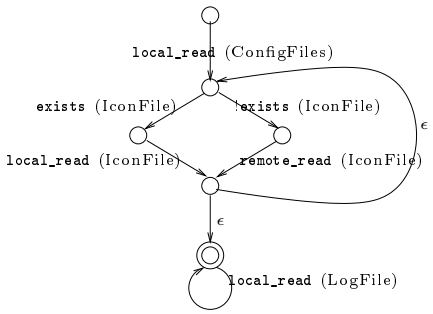

Figure 4.3: An EFSA Model, after Sekar *et al.* [103]

system calls as follows. Each system call is abstracted, e.g., the first expected system call from D is a call to `local_read` with argument bound to the ConfigFiles state variable. Then D is expected to either take the left or the right transition going down, depending on whether some tested file (in variable IconFile) exists or not. In the first case, D should call `local_read`, in the second case, D should call `remote_read`. The transitions labeled ϵ are meant to be firable spontaneously.

Typical policies considered by Sekar *et al.* are invariants of the form "any program should either access local files or access the network but not both" (this is violated above, assuming obvious meanings for system calls), or that only files from the `/var/log/httpd/` directory should be read by D. Policies are again expressed as EFSA, and enforcement models can be computed as a form of synchronized product denoting the intersection of the languages of M and P.

Discussion The EFSA models used in MCC are sufficiently close to the automaton-based model used in Orchids (which appeared about at the same time, see the second part of [97]; or see [87] for a more in-depth treatment) that the EFSA built in the MCC approach can be fed almost directly to Orchids. In our approach, we use Orchids for EFSA checking. More details about the proposed approach will be given in the following sections.

4.3 Threat Model

4.3.1 Sensitive Resources

The hypervisor is not alone in its task of administering the guest domains on the system. A special privileged domain called Domain0 serves as an administrative interface to Xen. Domain0 is the first domain launched when the system is booted, and it can be used to create and configure all other regular guest domains. Domain0 has direct physical access to all hardware, and it exports the simplified generic class devices to each DomU. The critical spots in our implementation presented in the previous chapter are the VMM (hypervisor) itself, domain zero, and the surveillance VM running Orchids. Attacking the latter is a nuisance, but is not so much of a problem as attacking the VMM or domain zero, which would lead to complete subversion of the system. Moreover, the fact that Orchids runs in an isolated VM averts most of the effects of any vulnerability that Orchids may have.

Attacks against the VMM are much more devastating. Wojtczuk's 2008 attacks on Xen 2 [117] allow one to take control of the VMM, hence of the whole machine, by rewriting arbitrary code and data using DMA channels, and almost without the processor's intervention... quite a

fantastic technique, and certainly one that breaks the common idea that every change in stored code or data must be effected by some program running on one of the processors. Indeed, here a separate, standard chip is actually used to rewrite the code and data. Once an attacker has taken control over the VMM, one cannot hope to react in any effective way. In particular, the VMM controls entirely the hardware abstraction layer that is presented to each of the guest OSes: no network link, no disk storage facility, no keyboard input can be trusted by any guest OS any longer. Worse, the VMM also controls some of the features of the processor itself, or of the MMU, making memory or register contents themselves unreliable.

We currently have no idea how to prevent attacks such as Wojtczuk's, apart from unsatisfactory, temporary remedies such as checkpointing some selected memory areas in the VMM code and data areas. However, we claim that averting such attacks is best done by making sure that they cannot be run at all. Indeed, Wojtczuk's attacks only work provided the attacker already has *root access* to domain zero, and this is already quite a predicament. We therefore concentrate on ensuring that no unauthorized user can gain root access to domain zero.

Normally, only the system administrator should have access to domain zero. (In enterprise circles, the administrator may be a person with the specific role of an administrator. In family circles, we may either decide that there should be no administrator, and that the system should self-administer; or that one particular user has administrator responsibilities.) We shall assume that this administrator is *benevolent*, i.e., will not consciously run exploits against the system. However, he may do so without knowing it while *updating* his system...

4.3.2 Automatic Updates and Security Issues

Either in host-based architectures or in virtualized ones, automatic updates represent a serious threat to the security of systems. As shown earlier, attacking a simple VM or a managing VM such as Dom0 can be much more devastating than attacking a simple architecture with one single host. The attacker can take the control of the whole virtualized system (sometimes hundreds of VMs with virtual servers and critical data!). This can be done by downloading malicious updates for programs or drivers, these updates may contain trojans that are triggered once the update is executed. We will present this kind of attack scenarios in the folllowing sections with more details. Now let us explain how automatic updates can be a source of threat for computer systems in general.

Every day, millions of computer users and system administrators update software some manually, some automatically, and some unknowingly. In 2002, corporations spent more than 2 billion on patch management for operating systems alone [77]. Indeed, many CERT Technical Cyber Security Alerts suggest applying patches, upgrades, or updates to resolve security vulnerabilities. And system administrator tend to use content distribution networks to download software updates. These updates help to patch everyday bugs, plug security vulnerabilities, and secure critical infrastructure. Yet challenges remain for secure content distribution: many deployed software update mechanisms are insecure. Users and system administrators are between two choices : either let the update mechanism download the patches or keep the computer isolated from the network. The latter choice reduces the "life expectancy" of the computer system. The latter idea is not of interest, thus almost all operating systems, software and even shareware are equiped with mechanisms that check for new updates, and most of these systems can be configured to automatically download and install the updates, and sometimes without notifying the user.

On the other hand, many update systems are themselves riddled with security vulnerabilities. Kevin Fu *et al.* from the University of Massachusetts studied the so-called *secure mechanisms* for automatic updates. The results are disappointing [78]. Many software update mechanisms

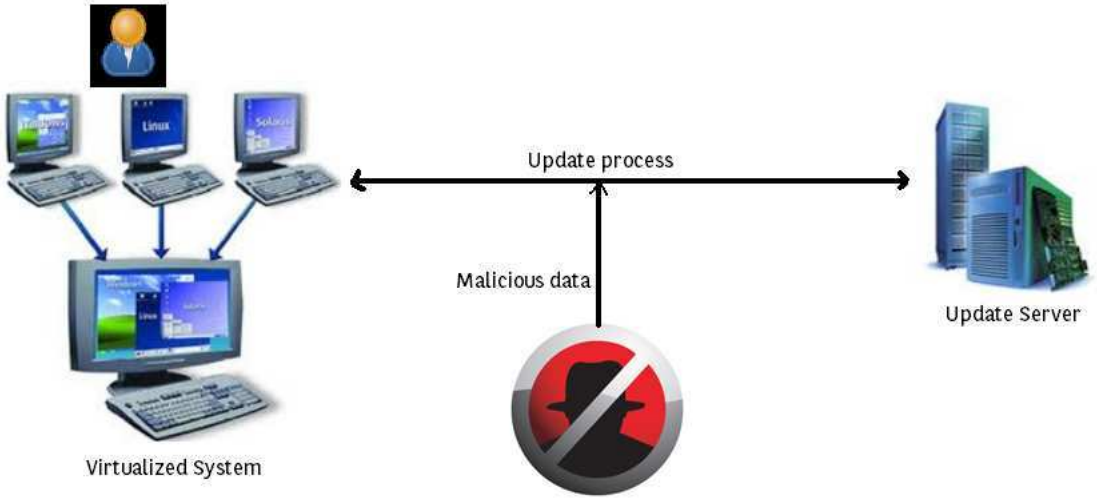

Figure 4.4: Attacking System Updates

implemented in famous software such as Microsoft Windows Update, Mozilla Firefox, Adobe Acrobat Reader and McAfee VirusScan lack basic security measures such as verification of digital signatures. Left open and unprotected, the update channels serve as an ideal backdoor for spreading malicious code. The main problem of these update mechanisms is the authentication of the updates in order to ensure their legitimacy. But it is also very important that software have an authenticated connection to the update server. As the name implies, having an update authenticated means that there is some way for the software doing the update to assure itself that the update is an authentic version from the intended source. Without authentication, a clever hacker can arrange a man-in-the-middle attack to insert an exploit in the update stream.

Most of these unsecured update systems simply go to a Web or FTP server, check the time stamp on the most recent file and download the file if it's new enough. The address of the server is usually hard-coded into the program doing the update, although occasionally it is stored in a configuration file. The attacker can simply to redirect the program making the update to a server controlled by the attacker himself. This is quite easy : with DNS-based attack the program can be redirected to the wrong website. To do this, the attacker run his software in an a café with wireless connection to Internet, waits until the victim does a DNS query, he catches the IP adress of the update server, and then can answer the DNS query before the official DNS server, and redirect the victim to the wrong destination. Even if updates are signed, an attacker capable of intercepting DNS requests or diverting Internet traffic can still use an update service to take over an unsuspecting victim's computer. A signature on an update just means that the update is authentic, it doesn't mean that the update is any good.

4.3.3 A possible Attack Scenario

One of the most tangible risks that can occur is the failure to keep up with the constant, labor-intensive process of patching, maintaining and securing each virtual server in a company. Unlike the physical servers on which they sit, which are launched and configured by hands-on IT managers who also install the latest patches, virtual machines tend to be launched from server images that may have been created, configured and patched weeks or months before.

One possible scenario is this: the administrator needs to upgrade some library or driver, and downloads a new version; this version contains a trojan horse, which runs Wojtczuk's attack. Modern automatic update mechanisms use cryptographic mechanisms, including certificates and cryptographic hashing mechanisms [118], to prevent attackers from running man-in-the-middle attacks, say, and substitute a driver with a trojan horse for a valid driver. However, there is no guarantee that the authentic driver served by the automatic update server is itself free of trojans. There is at least one actual known case of a manufacturer shipping a trojan (hopefully by mistake): some Video iPods were shipped with the Windows `RavMonE.exe` virus [96], causing immediate infection of any Windows host to which the iPods were connected.

4.4 Preliminaries

Related Work. We briefly present in this section some contributions that are related to our approach. Most of them implement run-time supervision and enforcement of security policies. Systems such as SELinux (op. cit.) are based on a security policy, but fail to recognize illegal *sequences* of legal actions. To give a simple example, it may be perfectly legal for user A to copy some private data D to some public directory such as `/tmp`, and for user B to read any data from `/tmp`, although our security policy forbids any (direct) flow of sensitive data from A to B. Such sequences of actions are called *transitive flows* of data in the literature. To our knowledge, Zimmerman *et al.* [120, 121, 122] were the first to propose an IDS that is able to check for illegal transitive flows. Briffaut [79] shows that even more general policies can be efficiently enforced, including non-reachability properties and Chinese Wall policies, among others; in general, Briffaut uses a simple and general policy language. We propose another, perhaps more principled, language in Section 4.5, based on linear temporal logic (LTL). Using the latter is naturally related to a more ancient proposal by Roger *etal.* in [97]. However, LTL as defined in (the first part of) the latter paper only uses future operators, and is arguably ill-suited to intrusion detection (as discussed in op. cit. already). Here, instead we use a fragment of LTL *with past*, which, although equivalent to ordinary LTL with only future operators as far as satisfiability is concerned (for some fixed initial state only, and up to an exponential-size blowup), will turn out to be much more convenient to specify policies, and easy to compile to rules that can be fed to the Orchids IPS [91, 87].

Linear Temporal Logic. As the language we propose in the next section is a variant of LTL (Linear Temporal Logic) with past operators. We give a brief presentation of this language. We start by presenting temporal logics.

The term Temporal Logic has been used to cover all approaches to the representation of temporal information within a logical framework. This logic can be used as a formalism for clarifying philosophical issues about time, as a framework within which to define the semantics of temporal expressions in natural language, as a language for encoding temporal knowledge in artificial intelligence, and as a tool for handling the temporal aspects of the execution of computer programs.

LTL is a modal temporal logic with modalities referring to time. It was first proposed for the formal verification of computer programs by Amir Pnueli in 1977 [100]. It has become the standard language for linear-time model checking. Model checking is the automatic verification that a model (typically a transition system) of a system possesses certain (un)desired properties. LTL is supported by many model checkers such as SPIN [101].

The alphabet of LTL is composed of:

- atomic proposition symbols p,q,r,...,

- boolean connectives $\wedge, \vee, \rightarrow, \leftrightarrow$
- temporal connectives $\bigcirc, \Box, \Diamond, \Re, U$.

The set of LTL formulae is defined inductively, as follows:

- any atomic proposition is a formula,
- if φ and ψ are formulae, then φ and $\varphi \bullet \psi$, for $\bullet \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ are also formulae,
- if φ and ψ are formulae, then $\bigcirc\varphi, \Box\varphi, \Diamond\varphi, \varphi\ U\ \psi, \varphi\Re\psi$ are formulae,
- nothing else is a formula.

Past LTL and Safety Properties. In [102], LTL, which has only future operators, is extended with past operators. This allows the easy writing of specifications which can be shorter, easier and more intuitive. LTL with past operators has been proved to be more succint than pure-future temporal logic [18]. Consider the following example taken from [18] where future-time modalities such as F ("sometimes in the future"), G ("always in the future") and U ("until") are complemented with their past-time counterparts (F^{-1} for "once in the past", G^{-1} for "always in the past" and S or U^{-1} for "since", ...). The statement "every request is eventually granted " is expressed by :

$$\mathrm{G}(\text{request} \Rightarrow \mathrm{F}\ \text{grant})$$

However, with past-time modalities, the statement can be expressed as follows, "a grant should be preceeded by a request :

$$\mathrm{G}(\text{request} \Rightarrow F^{-1}\ \text{grant})$$

LTL with past reveals very useful in dealing with safety properties. Informally, safety properties are properties of systems where every violation of a property occurs after a finite execution of the system. Safety properties are relevant in many areas of formal methods. Testing methods based on executing a finite input and observing the output can only detect safety property violations. Monitoring executions of programs is also an area where safety properties are relevant as monitoring also only can detect failures of safety properties. Naturally, formal specifications are also verified to make sure that a given safety property holds.

All of the above mentioned uses of safety properties can be accomplished by specifying the properties as finite automata. While automata are useful in many cases, a more declarative approach, such as using a temporal logic, is usually preferred. Many model checking tools, such as SPIN [101], support linear temporal logic (LTL). In the automata theoretic approach to verification [60, 61, 62], LTL formulas are verified by translating their negation to Buchi automata, which are then synchronised with the system. If the synchronised system has an accepting execution, the property does not hold. One could benefit from using finite automata instead of Buchi automata if the given LTL property is a safety property. Reasoning about finite automata is simpler than reasoning about Buchi automata. For explicit state model checkers, reasoning about Buchi automata requires slightly more complicated algorithms. In the symbolic context, emptiness checking with BDDs is in practice significantly slower than simple reachability [63]. For model checkers based on net unfoldings, such as [64], handling safety is much easier than full LTL [65].

Unfortunately, there are some complexity related challenges in translating LTL formulas to finite automata. A finite automaton specifying every finite violation of a LTL safety property can be doubly exponential in the size of the formula [66].

4.5 A Line of Defense: LTL with Past and Orchdis

4.5.1 The Proposed Language

Consider the case whereby we have just downloaded a device driver, and we would like to check whether it is free of a trojan. Necula and Lee pioneered the idea of shipping a device driver with a small, formal proof of its properties, and checking whether this proof is correct before installing and running the driver. This is *proof-carrying code* [89]. More suited to security, and somehow more practical is Sekar *et al.*'s idea of *model-carrying code* (MCC) [103]. Both techniques allow one to accept and execute code even from untrusted producers.

However, we feel that a higher-level language, allowing one to write acceptable policies for automatic updates in a concise and readable manner, would be a plus. There have been many proposals of higher-level languages already, including linear temporal logic (LTL) [97], chronicles [88], or the BMSL language by Sekar and Uppuluri [104], later improved upon by Brown and Ryan [80]. It is not our purpose to introduce yet another language here, but to notice that a simple variant of LTL *with past* will serve our purpose well and is efficiently and straightforwardly translated to Orchids rules—which we equate with EFSA here, for readability, glossing over inessential details.

Consider the following fragment of LTL with past. We split the formulae in several sorts. $F^\bullet$ will always denote *present tense* formulae, which one can evaluate by just looking at the current event:

$F^\bullet$	::=	$P(\vec{x}) \mid cond(\vec{x})$	atomic formula
	$\mid$	$\bot$	false
	$\mid$	$F^\bullet \wedge F^\bullet$	conjunction
	$\mid$	$F^\bullet \vee F^\bullet$	disjunction

Atomic formulae check for specific occurrences of events, e.g., `local_read` (IconFile) will typically match the current event provided it is of the form `local_read` applied to some argument, which is bound to the state variable IconFile. In the above syntax, $\vec{x}$ denotes a list of state variables, while $cond(\vec{x})$ denotes any computable formula of $\vec{x}$, e.g., to check that IconFile is a file in some specific set of allowed directories. This is as in [104, 80]. We abbreviate $P(\vec{x}) \mid \top$, where $\top$ is some formula denoting true, as $P(\vec{x})$.

Note that we do not allow for negations in present tense formulae. If needed, we allow certain negations of atomic formulae as atomic formulae themselves, e.g., `!exists` (IconFile). However, we believe that even this should not be necessary. Disjunctions were missing in [104], and were added in [80].

Next, we define *past tense* formulae, which can be evaluated by looking at the current event and all past events, but none of the events to come. Denote past tense formulae by $F^\leftarrow$:

$F^\leftarrow$	::=	$F^\bullet$	present tense formulae
	$\mid$	$F^\leftarrow \wedge F^\leftarrow$	conjunction
	$\mid$	$F^\leftarrow \vee F^\leftarrow$	disjunction
	$\mid$	$F^\leftarrow \diagdown F^\bullet$	without
	$\mid$	Start	initial state

All present formulae are (trivial) past formulae, and past formulae can also be combined using conjunction and disjunction. The novelty is the "without" constructor: $F^\leftarrow \diagdown F^\bullet$ holds iff $F^\leftarrow$ held at some point in the past, and since then, $F^\bullet$ never happened. Apart from the without operator, the semantics of our logic is standard. We shall see below that it allows us to encode a number of useful idioms. The past tense formula Start will also be explained below.

$$
\begin{array}{rcll}
\blacksquare\neg F^\bullet & \stackrel{\text{def}}{=} & \text{Start} \diagdown F^\bullet & \text{“}F^\bullet \text{ never happened in the past”} \\
\blacklozenge F^\leftarrow & \stackrel{\text{def}}{=} & F^\leftarrow \diagdown \bot & \text{“}F^\leftarrow \text{ was once true in the past”} \\
F^\leftarrow \rightarrow F^\bullet & \stackrel{\text{def}}{=} & \blacklozenge F^\leftarrow \wedge F^\bullet & \text{“}F^\leftarrow \text{ was once true, and now } F^\bullet \text{ is”} \\
F^\leftarrow \rightarrow F_1^\bullet \rightarrow F_2^\bullet \rightarrow \ldots \rightarrow F_n^\bullet & \stackrel{\text{def}}{=} & (\ldots((F^\leftarrow \rightarrow F_1^\bullet) \rightarrow F_2^\bullet) \rightarrow \ldots) \rightarrow F_n^\bullet & \\
F_1^\bullet; F_2^\bullet; \ldots; F_n^\bullet & \stackrel{\text{def}}{=} & \text{Start} \rightarrow F_1^\bullet \rightarrow \ldots \rightarrow F_n^\bullet & \text{Chronicle}
\end{array}
$$

Figure 4.5: Some Useful Idioms

Formally, present tense formulae $F^\bullet$ are evaluated on a current event e, while past tense formulae $F^\leftarrow$ are evaluated on a stream of events $\vec{e} = e_1, e_2, \ldots, e_n$, where the current event is e_n, and all others are the past events. (We warn the reader that the semantics is meant to reason logically on the formulae, but is not indicative of the way they are evaluated in practice. In particular, although we are considering past tense formulae, and their semantics refer to past events, our algorithm will never need to read back past events.) The semantics of the without operator is that $\vec{e} = e_1, e_2, \ldots, e_n$ satisfies $F^\leftarrow \diagdown F^\bullet$ if and only if there is an integer m, with $0 \leq m < n$, such that the proper prefix of events $e_1, e_2, \ldots, e_m$ satisfies $F^\leftarrow$ for some values of the variables that occur in $F^\leftarrow$ (“$F^\leftarrow$ held at some point in the past”), and none of e_{m+1}, …, e_n satisfies $F^\bullet$ (“since then, $F^\bullet$ never happened”)—precisely, none of e_{m+1}, …, e_n satisfies $F^\bullet$ *with the values of the variables obtained* so as to satisfy $F^\leftarrow$; this makes perfect sense if all the variables that occur in $F^\leftarrow$ already occur in $F^\bullet$, something we shall now assume.

The past tense formula Start has trivial semantics: it only holds on the empty sequence of events (i.e., when $n = 0$), i.e., it only holds when we have not received any event yet. This is not meant to have any practical use, except to be able to encode useful idioms with only a limited supply of temporal operators. For example, one can define the formula $\blacksquare\neg F^\bullet$ (“$F^\bullet$ never happened in the past”) as $\text{Start} \diagdown F^\bullet$.

The without operator allows one to encode other past temporal modalities, see Figure 4.5. In particular, we retrieve the *chronicle* $F_1^\bullet; F_2^\bullet; \ldots; F_n^\bullet$ [88], meaning that events matching $F_1^\bullet$, then $F_2^\bullet$, …, then $F_n^\bullet$ have occurred in this order before, not necessarily in a consecutive fashion. More complex sequences can be expressed. Notably, one can also express disjunctions as in [80], e.g., disjunctions of chronicles, or formulae such as $(\texttt{login}(Uid) \diagdown \texttt{logout}(Uid)) \wedge \texttt{local\_read}(Uid, ConfigFile)$ to state that user Uid logged in, then read some $ConfigFile$ locally, without logging out inbetween.

Let us turn to more practical details. First, we do not claim that only Start and the without ($\diagdown$) operator should be used. The actual language will include syntactic sugar for chronicles, box ($\blacksquare$) and diamond ($\blacklozenge$) modalities, and possibly others, representing common patterns. The classical past tense LTL modality $\mathcal{S}$ (“since”) is also definable, assuming negation, by $F \mathcal{S} G = G \diagdown \neg F$, but seems less interesting in a security context.

Second, as already explained in [97, 104, 80], we see each event e as a formula $P(fld_1, fld_2, \ldots, fld_m)$, where fld_1, fld_2, …, fld_m are taken from some domain of values—typically strings, or integers, or time values. This is an abstraction meant to simplify mathematical description. For example, using `auditd` as event collection mechanism, we get events in the form of strings such as:

```
1276848926.326:1234 syscall=102 success=yes a0=2 a1=1 a2=6 pid=7651
```

which read as follows: the event was collected at date 1276848926.326, written as the number of seconds since the epoch (January 01, 1970, 0h00 UTC), and is event number 1234

(i.e., we are looking at event e_{1234} is our notation); this was a call to the `socket()` function (code 102), with parameters `PF_INET` (Internet domain, where `PF_INET` is defined as 2 in `/usr/include/socket.h`—`a0` is the first parameter to the system call), `SOCK_STREAM` (= 1; `a1` is connection type here), and with the TCP protocol (number 6, passed as third argument `a2`); this was issued by process number 7651 and returned with success. Additional fieldss that are not relevant to the example are not shown. This event will be understood in our formalization as event e_{1234}, denoting `syscall` $(1276848926.326, 102, \texttt{"yes"}, 2, 1, 6, 7651)$. The event e_{1234} satisfies the atomic formula `syscall` $(Time, Call, Res, Dom, Conn, Prot, Pid) \mid Res = \texttt{"yes"}$ but neither `audit` (X) nor `syscall` $(Time, Call, Res, Dom, Conn, Prot, Pid) \mid Time \leq 1276848925$.

4.5.2 The Translation Algorithm

Here we will explain how we can detect when a given sequence of events $\vec{e}$ satisfies a given formula in our logic, algorithmically. To this end, we define a translation to the Orchids language, or to EFSA, and rely on Orchids' extremely efficient model-checking engine [87]. The translation is based on the idea of *history variables*, an old idea in model-checking safety properties in propositional LTL. Our LTL is *not* propositional, as atomic formulae contain free variables—one may think of our LTL as being first-order, with an implicit outer layer of existential quantifiers on all variables that occur—but a similar technique works.

It is easier to define the translation for an extended language, where the construction $F^{\leftarrow} \diagdown F^{\bullet}$ is supplemented with a new construction $F^{\leftarrow} \diagdown^* F^{\bullet}$ (*weak without*), which is meant to hold iff $F^{\leftarrow}$ once held in the past, *or holds now*, and $F^{\bullet}$ did not become true afterwards.

The *subformulae* of a formula F are defined as usual, as consisting of F plus all subformulae of its immediate subformulae. To avoid some technical subtleties, we shall assume that Start is also considered a subformula of any past tense formula. The *immediate subformulae* of $F \wedge G$, $F \vee G$, $F \diagdown^* G$ are F and G, while atomic formulae, $\bot$ and Start don't have any immediate subformula. To make the description of the algorithm smoother, we shall assume that the immediate subformulae of $F \diagdown G$ are not F and G, but rather $F \diagdown^* G$ and G. Indeed, we are reproducing a form of Fischer-Ladner closure here [84].

Given a fixed past-tense formula $F^{\leftarrow}$, we build an EFSA that monitors exactly when a sequence of events will satisfy $F^{\leftarrow}$. To make the description of the algorithm simpler, we shall assume a slight extension of Sekar *et al.*'s EFSA where state variables can be assigned values on traversing a transition. Accordingly, we label the EFSA transitions with a sequence of *actions* $\$x_1 := e_1; \$x_2 := e_2; \ldots; \$x_k := e_k$, where $\$x_1$, $\$x_2$, …, $\$x_k$ are state variables, and e_1, e_2, …, e_k are expressions, which may depend on the state variables. This is actually possible in the Orchids rule language, although the view that is given of it in [87] does not mention it. Also, we will only need these state variables to have two values, 0 (false) or 1 (true), so it is in principle possible to dispense with all of them, encoding their values in the EFSA's finite control. (Instead of having three states, the resulting EFSA would then have $3\,2^k$ states.)

Given a fixed $F^{\leftarrow}$, our EFSA has only three states q_{init} (the initial state), q, and q_{alert} (the final, acceptance state). We create state variables $\$x_i$, $1 \leq i \leq k$, one per subformula of $F^{\leftarrow}$. Let F_1, F_2, …, F_k be these subformulae (present or past tense), and sort them so that any subformula of F_i occurs before F_i, i.e., as F_j for some $j < i$. (This is a well-known *topological sort*.) In particular, F_k is just $F^{\leftarrow}$ itself. Without loss of generality, let Start occur as F_1. The idea is that the EFSA will run along, monitoring incoming events, and updating $\$x_i$ for each i, in such a way that, at all times, $\$x_i$ equals 1 if the corresponding subformula F_i holds on the sequence $\vec{e}$ of events already seen, and equals 0 otherwise.

There is a single transition from q_{init} to q, which is triggered without having to read any event at all. This is an *ϵ-transition* in the sense of [87], and behaves similarly to the transitions

`exists` (IconFile) and `!exists` (IconFile) of Figure 4.3. It is labeled with the actions $\$x_1 := 1; \$x_2 := 0; \ldots; \$x_k := 0$ (Start holds, but no other subformula is currently true).

There is also a single ϵ-transition from q to q_{alert}. This is labeled by no action at all, but is guarded by the condition $\$x_k == 1$. I.e., this transition can only be triggered if $\$x_k$ equals 1. By the discussion above, this will only ever happen when F_k, i.e., $F^{\leftarrow}$ becomes true.

Finally, there is a single (non-ϵ) transition from q to itself. Since it is not an ϵ-transition, it will only fire on reading a new event e [87]. It is labeled with the following actions, written in order of increasing values of i, $1 \leq i \leq k$:

$\$x_1$	:=	0	(Start is no longer true)
$\$x_i$	:=	$P(\vec{x}) \wedge cond(\vec{x})$	(for each i such that F_i is atomic, i.e., F_i is $P(\vec{x}) \mid cond(\vec{x})$)
$\$x_i$	:=	0	(if F_i is $\perp$)
$\$x_i$	:=	$and(\$x_j, \$x_k)$	(if $F_i = F_j \wedge F_k$)
$\$x_i$	:=	$or(\$x_j, \$x_k)$	(if $F_i = F_j \vee F_k$)
$\$x_i$	:=	$or(\$x_j, and(not(\$x_k), \$x_i))$	(if $F_i = F_j \nwarrow^* F_k$)
$\$x_i$	:=	$and(not(\$x_k), \$x_\ell)$	(if $F_i = F_j \nwarrow F_k$, and $F_j \nwarrow^* F_k$ is F_ℓ, $\ell < i$)

Here, *and*, *or* and *not* are truth-table implementations of the familiar Boolean connectives, e.g., $and(0,1)$ equals 0, while $and(1,1)$ equals 1. We assume that $P(\vec{x})$, i.e., $P(x_1, \ldots, x_n)$ will equal 1 if the current event is of the form $P(s_1, \ldots, s_n)$, and provided each x_j that was already bound was bound to s_j exactly, in which case those variable x_j that were still unbound will be bound to the corresponding s_j. E.g., if x_1 is bound to 102 but x_2 is unbound, then $P(x_1, x_2)$ will equal 1 if the current event is $P(102, 6)$ (binding x_2 to 6), or $P(102, 7)$ (binding x_2 to 7), but will equal 0 if the current event is $Q(102, 6)$ for some $Q \neq P$, or $P(101, 6)$. We hope that this operational view of matching predicates is clearer than the formal view (which simply treats x_1, ..., x_n as existentially quantified variables, whose values will be found as just described).

The interesting case is when F_i is a without formula $F_j \nwarrow F_k$, or $F_j \nwarrow^* F_k$. $F_j \nwarrow F_k$ will become true after reading event e whenever $F_j \nwarrow^* F_k$ was already true before reading it, and F_k is still false, i.e., when $\$x_\ell = 1$ and $\$x_k = 0$, where ℓ is the index such that $F_j \nwarrow^* F_k$ occurs in the list of subformulae of $F^{\leftarrow}$ as F_ℓ. So in this case we should update $\$x_i$ to $and(not(\$x_k), \$x_\ell)$, as shown above. This relies on updating variables corresponding to weak without formulae $F_j \nwarrow^* F_k$: $F_j \nwarrow^* F_k$ becomes true after reading event e iff either F_j becomes true ($\$x_j = 1$), or $F_j \nwarrow^* F_k$ was already true before ($\$x_i$ was already equal to 1) and F_k is false on event e ($\$x_k$ equals 0), whence the formula $\$x_i := or(\$x_j, and(not(\$x_k), \$x_i))$ in this case.

Note that our LTL fragment only deals with safety formulae of a particular form. It is easy to extend this fragment to one handling with more general *obligation formulae*, which are Boolean combinations of safety formulae.

From Policy Formulas to EFSA. Now we give a more concrete descritpion of the translation described above. We present in details how a given formula written in our language can be translated to the EFSA of Orchids representing the attack signature.

Given a formula F with atomic formulas $P_1, ..., P_m$ (m>=1). For each i, we save the information about how P_i appears in F, either negated or not (we consider the formula G as negated in the formula $F \nwarrow G$). We translate F into an EFSA of Orchids by first creating a state *q_detect* which will be responsible of warning us when an event e occurs. This event should unfluence enough the values of the atomic formulas in order to change the value of F. And it will be the case when:

- if P_i is *true* in the current event and P_i appears positively (not negated) in F.

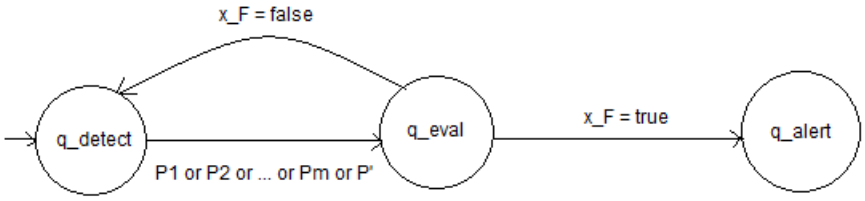

Figure 4.6: The generated EFSA

- if P_i is $false$ in the current event and P_i appears as negated in F.

For ease, we can under-approximate, and decide to be averted in a superset of the cases where P_i is $false$. For instance, if P_i says "the current event is the syscall *open* with a first parameter having the same value as the variable X", let $P'_i = true$; if P_i says "the current event is an *open* syscall", we can be more precise and write P'_i ="the current event is not a call to the *open* function".

P'_i is an under-upproximation of the negation of P_i. To simplify this step, we consider $P'_i = true$ if at least one of the P_i appears as negated in F (*i.e.*, we had a $F \setminus G$ with $G \neq false$), and $P' = false$ otherwise.
The state q_detect will be just an if condition of the form:

state q_detect
{
if (P_1 or P_2 or ... or P_m or P') goto q_eval;
}

Then, the state q_eval performs only epsilon-transitions (no if):

state q_eval
{
$x_1 = P_1$; (true of false depending on the value of P)
$x_2 = P_2$;
...
$x_m = P_m$;

/* Calculate the value of F and save it in the variable x_F,
based on the algorithm cited above*/

if (x_F) goto q_alert;

goto q_detect;
}

The q_alert state contains reporting, defensive and offensive commands performed by Orchids. Other types of actions can also be added to this state. This is in the case where the atomic formulas P_i are free from logic variables (first order). Otherwise, the statements "$\$x_i = P_i$" have to replaced by a matching mechanism. For instance, if P_i = *"syscall = fopen, arg1=X"*, the

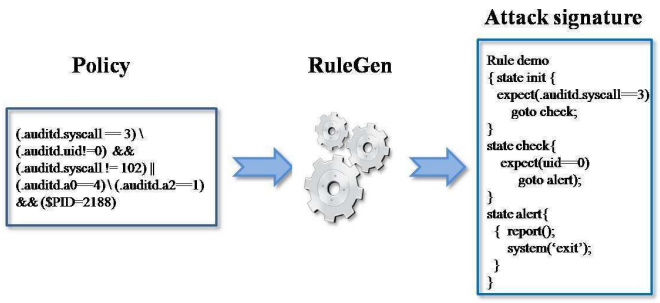

Figure 4.7: The RuleGen tool

"$\$x_i = P_i$" have to be replaced by:

if (.syscall = "fopen" $\wedge$ isset(X) $\wedge$ .arg1 = X) goto q_1;
if (.syscall = "fopen" $\wedge$!isset(X)) goto q_2;
if (.syscall != "fopen") goto q_3;

q1 { $\$x_1$ = true ; goto q_eval_follow; }
q2 { $\$x_1$ = true ; X = .arg1 ; goto q_eval_follow; }
q3 { $\$x_1$ = false ; goto q_eval_follow; }

Then, in q_eval_follow we do the same thing for $\$x(i+1) = P(i+1)$, etc. This ends when we notice that we have tested all the atomic formulas. One can notice that it is a large sequence of epsilon-transtions. No one can read a new event except q_detect.

This completes the description of the translation. We now rely on Orchids' fast, real-time monitoring engine to alert us in case any policy violation, expressed in our fragment of LTL, is detected.

The RuleGen Tool. RuleGen [1, 2] implements the algorithm cited above. It translates formulas written in our language into EFSA representing attacks signatures. RuleGen is fully automatic and does not need user intervention at any phase of the translation. RuleGen helps the administrator avoid the complexity of writing Orchids' rules. This is important since the attack base of Orchids needs to be updated frequently and sometimes quickly.

4.6 Facing a Malicious Driver

We give in this section a case study of the presented idea by simulating the following attack scenario : the administrator of a Xen system tries to download a new driver and installs it in Dom0. This driver is malicious and contains two exploits. We will show how relying on RuleGen and Orchids can help the administartor prevent the disaster. The malicious driver is a modified version of FUSE [123], a generic filesystem driver. This modified version of FUSE contains two real-world DoS attacks that are executed automatically once the driver is loaded.

N.B. We do not claim that the chosen attacks are the most suited to this scenario, our objective is to give a simple use case with simple attacks. The procedure can be applied on much more complicated attacks. We aim to show how from simple logic formulas, one can protect a complex virtualized system.

In order to simulate the real world attack scenario, we followed these steps :

1. Inject the two attacks in the driver source code and upload it on a remote server;
2. Write the formulas corresponding to the attacks (N.B. here we know exactly what we want to prevent our system from, in most cases one can write generic policy formulas in order to generate rules protecting from families of attacks);
3. Launch *RuleGen* and translate the written formulas into attacks signatures and add them to Orchids;
4. Log in to the Dom0 and download the malicious driver;
5. Install the driver and let Orchids deal with the attacks.

The first attack [124] is a DoS attack consisting of two calls to the *listen* function (linux/-socket.h) on the same ATM (Asynchronous Transfer Mode) socket descriptor. Linux 2.6.x kernels and many Linux distributions are vulnerable to this attack. Once executed, this attack makes the Dom0 unavailable and the administrator becomes unable to react since his administration platform is not responding. Consequently, all running VMs will be unavailable.

We want to make sure that the attack will be executed automatically once the filesystem driver is mounted. We modify the source code of the driver mounting module as follows :

```
//FUSE driver file : fusermount.c
switch (ch) {
    case 'u':
      unmount = 1;
      /******* The attack code ******/
      int sock = socket(PF_ATMSVC, 0, 37);
      listen(sock, 7);
      listen(sock, 2);
      system("/bin/cat /proc/net/atm/pvc");
      /********************/
      break;
    case 'h':
      usage();
```

The Corresponding Formula.

The corresponding formula can be written as follows :

$$![(\blacklozenge(\$PID == .auditd.pid \wedge .auditd.syscall == 102 \wedge .auditd.a0 == 4) \wedge (.auditd.pid == \$PID \wedge .auditd.syscall == 102 \wedge .auditd.a0 == 4))]$$

This formula describes the negation of two events correlated by the variable $\$PID$ (the process identifier) and connected with the "$\wedge$" (*and*) operator. The first event is a *socketcall* system call (code 102) with the first argument $a0 = 4$ (the *listen* function). The *pid* of the process is captured from the *.auditd.pid* field and stored in the variable $\$PID$. The second event is similar to the first one, but must be triggered by the same process, and should come later since the first one is preceded by the diamond $\blacklozenge$ operator (which means that it happened once in the past).

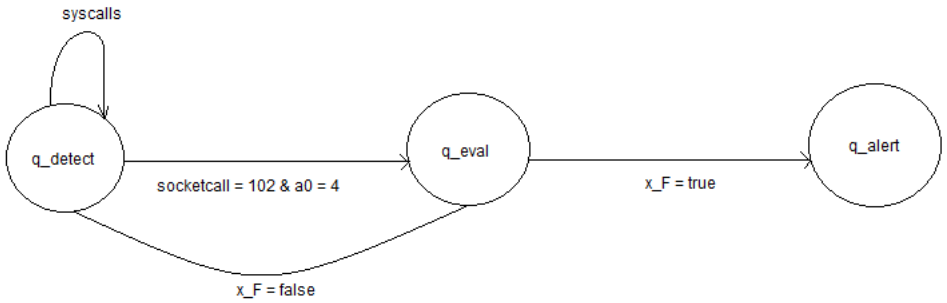

Figure 4.8: The *listen_atm* attack

The generated EFSA.

RuleGen parses the formula and generates the EFSA corresponding to the attack signature. The first state is *q_detect*, this state waits for a *listen* function call (a *socketcall* system call with the value 4 for the first parameter) and at the same time saves the *pid* of the process triggering this event. Once the second state is reached, we are sure that time has elapsed, and the expected event was triggered. The second state *q_eval* calculates the value of the x_F variable. If $x\_F = true$, Orchids moves to the *q_alert* state. The *q_alert* state is responsible of killing the offending process and reporting to the administrator. The generated EFSA corresponds to Figure 4.8.

The second attack is also a DoS attack [125]. It goes in an infinite loop trying to obtain numerous file-lock leases, which will consume excessive kernel log memory. Once the leases timeout, the event will be logged, and kernel memory will be consumed. Many Linux 2.6.x kernels are vulnerable to this attack.

Here, we do the same thing as for the first attack, we inject the code of the exploit in another location in the FUSE source code to make sure that it will triggered the kernel starts using the driver.

```
//FUSE file : fusermount.c
static int open_fuse_device(char **devp)
{
  int fd = try_open_fuse_device(devp);
  /***** lock_lease_dos attack ****/
  int r;
  while(1)
  {
    //lock
    r = fcntl(fd, F_SETLEASE, F_RDLCK);
    //unlock
    r = fcntl(fd, F_SETLEASE, F_UNLCK);
  }
```

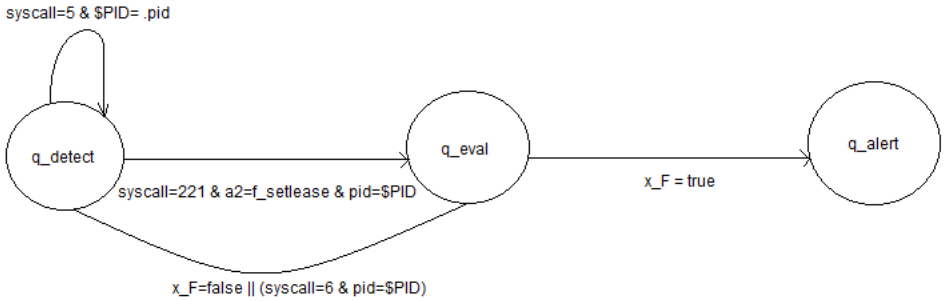

Figure 4.9: The *lock_lease_dos* attack

```
  if (fd >= -1) return fd;
  fprintf(stderr,"%s: fuse device error");
  return -1;
}
```

When the filesystem is mounted, the *fusermount* program (fusermount.c) tries to open *"/dev/-fuse"* (the *open_fuse_device()* function). At this moment, we are sure that the attack is being executed.

The Corresponding Formula.

The corresponding formula can be written as follows :

$$![((\blacklozenge(.auditd.syscall == 5 \wedge \$PID == .auditd.pid) \wedge (loop \wedge (.auditd.syscall == 221 \wedge .auditd.a2 == "f\_setlease" \wedge .auditd.pid == \$PID))) \setminus (.auditd.syscall == 6 \wedge .auditd.pid == \$PID))]$$

This formula can be read as follows: every process that makes a call to the *open* function (code 5) and then makes numerous locks ($fcntl64$ system call with code 221, and with the parameter "f_setlease") on a descriptor without closing it (*close* system call has the code 6), represents an attempt to make the system unavailable. The keyword *loop* is used when we need to express successive calls to the same event.

The generated EFSA.

As shown earlier, *RuleGen* transforms this formula into an EFSA representing the attack signature that feeds the base of Orchids without any adaptation. The generated EFSA corresponds to Figure 4.9

4.6.1 Experiments

We deployed our solution on a 1000 MHz Intel Core Duo machine with 4096 KB cache running Xen 3.3.1 as hypervisor. Dom0 is a 32-bit Fedora 11 Linux with 2 GB of RAM. We also use two guest VMs: Fedora 10 and Ubuntu 8 with 1 GB and 512 MB RAM, respectively. We perform a set of experiments to evaluate RuleGen and Orchids performance on the target platform using the malicious FUSE driver. Practical results look promising: Orchids can detect simultaneously the two DoS attacks presented earlier and stop them before the system crashes.

4.7 Conclusion and Further Work

We have presented in this chapter a new procedure for securing the sensitive resources of a virtualized system such as the Dom0. We have introduced a variant of the LTL lannguage with new past operators and showed how policies written in this language can be easily translated to attack signatures that we use to detect attacks on the system. Our procedure can be improved at many levels. First, some restrictions related to the language should be removed especially for expressing recursive calls to the *without* operator. Second, the translation also can be optimized in order to be more specific to the Orchids language. Finally, we feel that the expressiveness of the language should benefit from a more in-depth analysis in order to enrich it with more operators.

Chapter 5

Securing Communication In a Virtual Environment

5.1 Introduction

We discuss in this chapter the security threats related to communication in virtual networks *i.e.* networks built between virtual machines. We introduce in section 5.6 a multilevel security policy that covers network-related operations and VMM management primitives. We detail this policy by presenting the different constraints that must be respected by each operation.

5.2 Multilevel Networking

Computer networks became essential for sharing resources. Long before computers were routinely wired to the Internet, sites were building local area networks to share printers and files. Multilevel data sharing had to be addressed in a networking environment especially in the defense community. Initially, the community embraced networks of cheap computers as a way to temporarily sidestep the MLS problem. Instead of tackling the problem of data sharing, many organizations simply deployed separate networks to operate at different security levels, each running in system high mode. This approach did not help the intelligence community. Many projects and departments needed to process information carrying a variety of compartments and code words. It simply wasn't practical to provide individual networks for every possible combination of compartments and code words, since there were so many to handle. Furthermore, intelligence analysts often spent their time combining information from different compartments to produce a document with a different classification. In practice, this work demanded an MLS desktop and often required communications over an MLS network. Thus, MLS networking took two different paths in the 1990s. The intelligence community continued to pursue MLS products. This reflected the needs of intelligence analysts. In networking, this called for labeled networks, that is, networks that carried classification labels on their traffic to ensure that MLS restrictions were enforced. Many other military organizations, however, took a different path. Computers in most military organizations tended to cluster into networks handling data up to a specified security level, operating in system high mode. This choice was not driven by an architectural vision; it was more likely the effect of the desktop networking architecture emerging in the commercial marketplace combined with existing military computer security policies. Ultimately, this strategy was named multiple single levels (MSL) or multiple independent levels of security

(MILS). The objective of a labeled network is to prevent leakage of classified information. The leakage could occur through eavesdropping on the network infrastructure or by leaking data to an uncleared destination. This yielded two different approaches to labeled networking. The more complex approach used cryptography to keep different security levels separate and to prevent eavesdropping. The simpler approach inserted security labels into network traffic and relied on a reference monitor mechanism installed in network interfaces to restrict message delivery. In practice, the cryptographic hardware and key management processes have often been too expensive to use in certain large scale MLS network applications. Instead, sites have relied on physical security to protect their MLS networks from eavesdropping. This has been particularly true in the intelligence community, where the proliferation of compartments and codewords have made it impractical to use cryptography to keep security levels separate.

5.3 Virtual Networks

Modern hypervisors offer the ability to build virtual networks between virtual machines. These networks (see Figure 5.1) are very useful in both personal and professional activities since they offer the same opportunities as physical networks, but in a much lower cost in terms of hardware and time. On the other hand, these networks are facing many security threats due to the absence of rigourous security policies that protect the sensitive ressources of the network. We propose a multilevel security policy model for securing communication in virtual networks, this policy covers not only network operations, but also operations related to the management of the virtual architecture.

Hypervisors allows one to emulate one or several so-called *guest* operating systems (OS) in one or several *virtual machines* (VM). The different VMs execute as though they were physically distinct machines, and can communicate through ordinary network connections. A virtual network can be built between VMs, this allows them to communicate by simple network primitives. This kind of networks can be seen as a solution to the complexity of building physical networks : building and configuring a virtual network is a very easy task. On the other hand, most of the security threats we face in a non-virtualized environment exist in virtualized environments as well. Furthermore, virtual networks have other security weaknesses related to the the architecture of the network, since everything is located in the same machine. This needs serious defence and rigourous security policies. We propose in this chapter a multi-level security policy that covers common network operations and administrative actions. We take into consideration the constraints that must be satisfied during the communication between VMs and propose the policy model and discuss its implementation.

Figure 5.2 shows the three main technologies doing network virtualization : service, device and link virtualization.

A body of existing work has already examined the issues arised by virtualized architectures [106][107][108]. However, not enough work was done for securing virtual networks between VMs. The introduction of the Xen Security Modules (XSM) framework enables the enforcement of comprehensive control over the resources of the hypervisor. The XSM policy model is based on SELinux [113], so VMM policies will be comprehensive, but determining whether a security goal is enforced correctly seems to be non-trivial for beginning users due to the complexity of policy rules organization. Garfinkel et al. proposed Terra [56], a flexible architecture that offers a wide range of security mechanisms mainly the classification of virtual machines into *open-box VMs* and *closed-box VMs*. This has the disadvantage of dealing with abstracted VMs and having to install a monitor called *TVMM*. sHype [98] is one of the best-known security architecture for hypervisors : its primary goal was to control the information flows between VMs. sHype is based

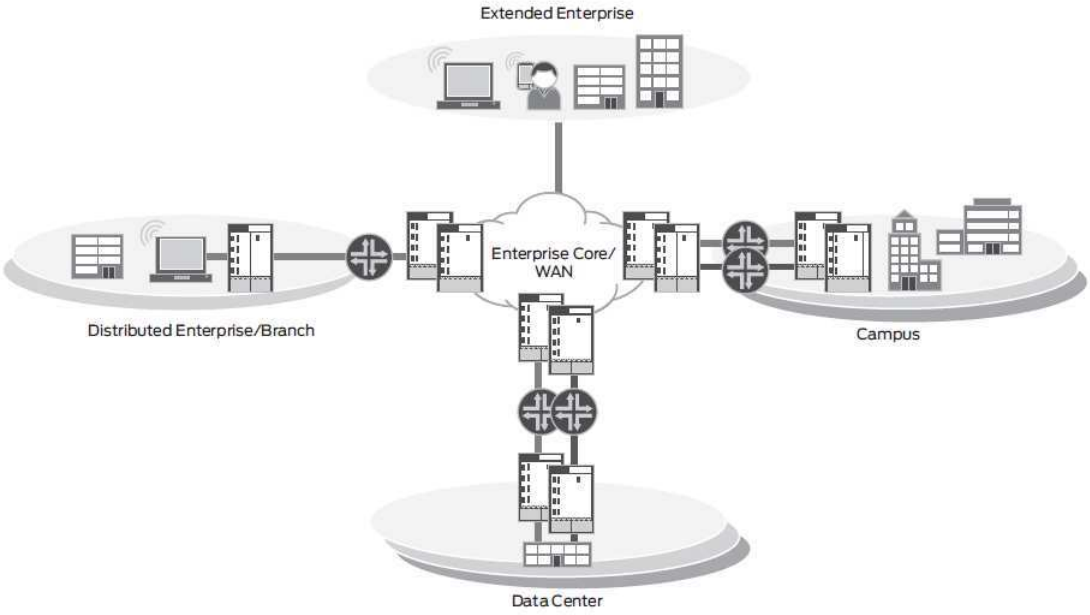

Figure 5.1: A Virtual Network

on the Xen hypervisor and does not protect other virtualized architecture.

In [110] [111], a role-based access control policy was introduced to VMMs by Hirano et al. This policy focuses only on the access between guest VMs and the VMM layer, and does not treat inter-VM communication. The security policy model we propose in this paper is comprehensive, easy to implement and covers almost all network operations performed by the VMs. Besides, our model covers management operations that can be performed by the administrator of the virtualized system which is a plus, and is not offered by the approaches cited above.

5.4 Advantages and Security Threats of Virtual Networks

We call *virtual network* the local network built between virtual machines in an hypervisor-based architecture.

We argue that these networks have several advantages : First, a virtual network reduces the networking hardware investment (fewer cables, hubs) and eliminates dependencies on hardware. Second, one can consolidate hardware by connecting guest systems that run in virtual machines in a single host. Also, consolidating servers in a virtual network allows one to reduce or eliminate the overhead associated with traditional networking components. Besides, by defining a virtual network on a single processor, one does not need to consider network traffic outside the processor. As a result : a high degree of network availability and performance.

In [5] we showed that virtual networks can be very useful for intrusion detection by proposing a decentralized supervision architecture on a single physical host based on the Xen hypervisor. This architecture is based on a virtual network allowing the communication between ordinary VMs, the surveillance VM and the administration VM called *domain*0. See Figure 3.2, which is perhaps more typical of Xen than other hypervisors.

On the other hand, the rapid scaling in virtual networks can tax the security system. In

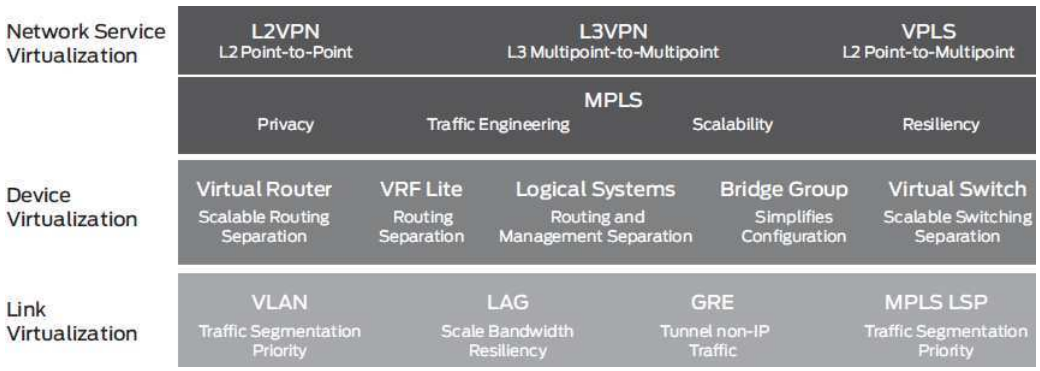

Figure 5.2: Network Virtualization Technologies

fact, the fast and unpredictable growth that can occur with VMs can exacerbate management tasks and significantly multiply the impact of catastrophic events, e.g. worm attacks where all machines should be patched, scanned for vulnerabilities, and purged of malicious code.

Collections of specialized VMs give rise to a phenomenon in which large numbers of machines appear and disappear from the network sporadically. While conventional networks can rapidly "anneal" into a known good configuration state, with many transient machines getting the network to converge to a "known state" can be nearly impossible.

For example, when worms hit conventional networks they will typically infect all vulnerable machines fairly quickly. Once this happens, administrators can usually identify which machines are infected, then cleanup infected machines and patch them to prevent re-infection, rapidly bringing the network back into a steady state.

Besides, in an unregulated virtual environment, such a steady state is often never reached. Infected machines appear briefly, infect other machines, and disappear before they can be detected, their owner identified, etc. Vulnerable machines appear briefly and either become infected or reappear in a vulnerable state at a later time. Also, new and potentially vulnerable virtual machines are created on an ongoing basis, due to copying, sharing, etc. As a result, worm infections tend to persist at a low level indefinitely, periodically flaring up again when conditions are right. The requirement that machines be online in conventional approaches to patch management, virus and vulnerability scanning, and machine configuration also creates a conflict between security and usability. VMs that have been long dormant can require significant time and effort to patch and maintain. This results in users either forgoing regular maintenance of their VMs, thus increasing the number of vulnerable machines at a site, or losing the ability to spontaneously create and use machines, thus eliminating a major virtue of VMs.

For instance, rolling back a machine by the checkpoint and restore mechanism can re-expose patched vulnerabilities, reactivate vulnerable services, re-enable previously disabled accounts or passwords, use previously retired encryption keys, and change firewalls to expose vulnerabilities. It can also reintroduce worms, viruses, and other malicious code that had previously been removed.

A subtler issue can break many existing security protocols. Simply put, the problem is that while VMs may be rolled back, an attacker's memory of what has already been seen cannot. For example, with a one-time password system like S/KEY where a user's real password is combined in an offline device with a short set of characters and a decrementing counter to form a single-use password. In this system passwords are transmitted in the clear and security is entirely reliant on the attacker not having seen previous sessions. If a machine running S/KEY is rolled back,

an attacker can simply replay previously sniffed passwords.

A more subtle problem arises in protocols that rely on the "freshness" of their random number source e.g. for generating session keys or nonces. Consider a virtual machine that has been rolled back to a point after a random number has been chosen, but before it has been used, then resumes execution. In this case, randomness that must be "fresh" for security purposes is reused.

5.5 Security Policy Models

5.5.1 Bell-LaPadula model

The Bell-Lapadula formal model [112] was first proposed by David Bell and Leonard LaPadula. This is a model of multi-level security proposed to the Department of Defense in 1973. This model uses mathematical concepts to define the security state of a system. Although this model has undergone several reviews and was subsequently improved (Biba model), it remains today the first reference model in security. The security theorem which is the foundation of this model states that a system is secure if and only if the initial state is a secure state and that all the state-transitions of the system are secure, then every intermediate state will also be secure. According to this theory, to show that a system is secure, we have to model by a state machine and to prove that the initial state is secure and all the transitions are secure. In the Bell LaPadula model, a computer system is described by a state machine that controls all access requests made by subjects on objects. Subjects are active entities of the model, objects represents passive entities. The model defines several security levels. Each object or subject can be classified corresponding to its sensitivity and have a level between the following ones: unclassified, confidential, secret and top-secret.

Two main properties are used for mandatory access: the *simple-security property (ss-property)* and the **-Property*. According to the *ss-Proprety*, a subject can read an object if and only if its security level is greater or equal than the object level. This ensures the confidentiality property.

The **-Property* or *star-property* says that a subject at a given security level must not write to any object at a lower security level (no write-down). It is also known as the Confinement property.

The model defines also the rules of access to objects :

- *Read-Only*: the subject has only read rights.
- *Append*: the subject has write permissions on the object but does not have read permissions.
- *Execute*: the subject has only execute permissions but can not read or write to the object.
- *Read-Write*: the subject has both read and write permissions.

Several security levels are used to manage the access rights.Subjects having the highest level have always the right to read all the objects of the model. Also a subject with high security level in the model can not write down to an object with a lower security level. A subject with a low security level can write to an object with a higher level. This is legitimized by the fact that subjects with higher levels have the read right on these objects (*-Property). The verification of the star-property requires the control of all information flows between subjects and objects in the system. When implementing this model, the existence of covert channels can cause problems. To prevent this, a more restrictive version of BLP uses the following rules :

- *No Read Up* When a subject requests a read access to an object, its security clearance must be greater or equal than the object level.

- *No Write Down* When a subject tries to write to an object, its security clearance must lower or equal than the object security level.

The imlementation of this model without any adaptation to the system environment can be very difficult. Also, the attribution of labels to some subjects or objects is not an easy task. Some properties were added to this model in order to make it easy-to-implement. In addition, among the limitations of this model is the fact that its only concern is confidentiality which can limits consequently the access and the sharing of information. One can mention also that BLP does not have any integrity or availability policies. Moreover, it allows covert channels and assumes only fixed rights such as *tranquility*.

5.5.2 Biba model

The Biba integrity model [109] was published at Mitre one year after the BLP model. When Biba noticed that the BLP policy did not provide protection against a user at level X writing information at level Y when X was a lower security level than Y. Thus a low security user could overwrite highly classified documents unless some sort of integrity policy were in place. Biba chose the mathematical dual of the BLP policy wherein there are a set of integrity levels, a relation between them, and two rules which, if properly implemented, have been mathematically proven to prevent information at any given integrity level from flowing to a higher integrity level. Typical integrity levels are "untrusted", "slightly trusted", "trusted", "very trusted", "so trusted that we don't need a higher level of trust", etc. The first rule is that a subject at a given integrity level X cannot write information to another integrity level Y if X is lower integrity than Y. This rule assures that low integrity subjects cannot corrupt high integrity subjects (called "no write up"). The second rule is that a subject at a given integrity level Y cannot read information from another integrity level X if X is lower integrity than Y. This rule assures that high integrity subjects cannot become corrupt by reading low integrity information (called "no read down"). Under the Biba integrity model a subject can execute a program or read a data file if the integrity of the object is higher than or equal to that of the subject. A subject is not permitted to read a data or program file which has a lower integrity. A high integrity process thus exists in an isolated environment in which everything visible has high integrity. This is exactly the environment desired for processes which are part of the TCB. The set of TCB programs can therefore be defined to be that set of program files whose integrity is greater than or equal to the lowest integrity used by any TCB subject.Similarly, the set of TCB data can be defined to be that set of data files whose integrity dominates the lowest integrity used by any TCBsubject. Let us examine some implications here. A privileged process running with the highest possible integrity will be able to read data which also has the highest possible integrity, but not data with any lower integrity. No matter what a user with a lower integrity puts on the system, even if it's an executable trojan horse in the privileged process's normal execution path, the privileged process can not be effected by the attack. Furthermore, the attacker would not be able to put the evil file into a directory which the privileged process could read, as the lower integrity process would not be able to modify the directory to do so. Processes with low integrity will be able to look at, but not touch, system data. Where other secure systems count on discretionary permissions alone to protect system data thatthe unprivileged user would want to see, such as the userid to user name mappings, the system with integrity can simply make these files the highest possible and not worry as much about traditional permissions.

5.5.3 DTE model

The *DTE (Domain and Type Enforcement)* model [105] is a high level access control model. DTE was present for years in certain commercial operating systems, the model uses *strong typing* implemented in the TAM model and constitutes a platform on which access control policies suc as BLP and Biba can be implemented. Typically, in an operating system, the security policies defined by DTE aims to :

- restrict the resources available for programs, especially for priviliged ones.
- control the access to sensitive resources and prevent the unauthorized access to these resources by other programs.

A global Domain Definition Table (DDT) contains the allowed interactions, where domains and types form rows and columns, and each cell holds a set of access modes. Subject-to-subject access control is based on a global Domain Interaction Table (DIT) with subjects as both descriptors and, again, a set of access modes, e.g. signal, create or destroy, in the cells. In contrast to the original TE model, DTE supports implicit attribute maintenance. This means that values may be only kept on a higher level of the directory and file hierarchy, but are used for all levels below as well. Also, the specification language allows to specify types by lookup path prefixes.

The first process on a system, the init process, gets a predefined initial domain assigned. Each process can enter another domain by executing a program bound to it, a so-called entry point. An entry point may be executed to explicitely enter one of its associated domains, if the subject's current domain has *exec* right on the target domain. The auto access right to a domain automatically selects this domain, if one of its entry points gets executed. The user-domain relationship is entirely built on entry points like command shells etc. However, a DTE aware login program can select from all domains associated with an entry point to avoid individual copies for each domain. The DTE model avoids the concept of users and only focuses on programs. User representation and role assignment are placed under the discretion of unspecific DTE aware applications outside the scope of the model. Another DTE drawback is that roles can only be changed through entry point programs. Dynamic role changes are specially useful for user based server programs.

5.5.4 Multilevel Security

Multi-level security was formalized by Bell and La-Padula [112] in order to control how information is allowed to flow between subjects in a system. These subjects are given a sensitivity level, or security clearance, and objects are also given a similar security classification. MLS policies attempt to restrict how information may flow between designated sensitivities. As an example, consider a military application with 4 sensitivities, ordered from least to most sensitive: Unclassified (UC), Confidential (CO), Secret (S), and Top Secret (TS). In this case, TS dominates S. Note that in this example the sensitivites form a total ordering; each sensitivity is either higher, lower, or equal to another. This is not always the case.

Multilevel security (MLS) has posed a challenge to the computer security community since the 1960s. MLS sounds like a mundane problem in access control: allow information to flow freely between recipients in a computing system who have appropriate security clearances while preventing leaks to unauthorized recipients. However, MLS systems incorporate two essential features: first, the system must enforce these restrictions regardless of the actions of system users or administrators, and second, MLS systems strive to enforce these restrictions with incredibly high reliability. This has led developers to implement specialized security mechanisms and to

apply sophisticated techniques to review, analyze, and test those mechanisms for correct and reliable behavior. Despite this, MLS systems have rarely provided the degree of security desired by their most demanding customers in the military services, intelligence organizations, and related agencies. The high costs associated with developing MLS products, combined with the limited size of the user community, have also prevented MLS capabilities from appearing in commercial products.

However, constraining how information may flow within a system is at the heart of many protection mechanisms and many security policies have direct interpretations in terms of multilevel security style controls. These include: Chinese Walls [72][73]; separation of duties and well formed transactions [74][75] and Role-Based Access Control [76].

Let us assume that we have a collection of trusted and untrusted VMs and we would like to connect them to form a secure virtual network. A network is said to be multilevel secure if it is able to protect multilevel information and users. That is the information handled by the network can have different classifications and the network users may have varying clearance levels.

5.6 The Proposed Security Policy Model

In developing the security policy, we combine certain features of some well computer security models such as the Bell-LaPadula model together with issues relevant to network security. Informally, the network discretionary and mandatory access control policy can be described as follows : we assume that the information required to provide discretionary access control resides within each network component, rather than in a centralized access control centre. The network discretionary access control policy is based on the identity of the network components, implemented in the form of an authorized connection list. This list determines whether a connection is allowed to be established between two network entities. The individual components may in addition impose their own controls over their users - e.g. the controls imposed when there is no network connection.

The network mandatory security policy requires appropriate labelling mechanisms to be present. One can either explicitly label the information transferred over the network or associate an implicit label with a virtual circuit connection. In our model we have the following scheme :
(a) Each network component is appropriately labelled. A mandatory policy based on the labels of the network components is imposed and it determines whether a requested connection between two entities is granted or not.
(b) Information transferred over the network is appropriately labelled. A mandatory security policy is used to control the flow of information between different subjects and objects, when performing different operations involving information transfer over the virtual network.

5.6.1 Modelling approach

The network security policy model we describe here is a state-machine based model. Essentially a state machine model describes a system as a collection of entities and values. At any time, these entities and values stand in a particular set of relationships. This set of relationships constitutes the state of the system. Whenever any of these relationships change the state of the system changes. The common type of analysis that can be carried out using such a model is the reachabitity graph analysis. The reachability graph analysis is used to determine whether the system will reach a given state or not. For instance, we may identify a subset of states W which represent "insecure" states and if the system reaches a state within this subset W, then

Figure 5.3: A dedicated VM for I/O

the system is said to be insecure. In describing such a state machine based security model, we need to perform the following steps :

- Define security related state variables in the network system.
- Define the requirements of a secure network state.
- Define the network operations which describe the system state transitions.

We make the following assumptions :

1. Reliable user authentication exists within each VM.
2. Only a user with the role of *Admin* can assign security classes to network subjects and network components, and assign roles to users.
3. Reliable transfer of information across the network.

5.6.2 Model Representation

In order to be generic, our model needs to take into consideration the recent development in virtualized systems area, thus we will deal with Input/Output devices as separated VMs : in fact VMware, Xen and many other hypervisors tend to dedicate a whole VM for I/O [8], and sometimes for the processor (see Figure 5.3), which reduces consequently the overhead for communicating the I/O and processor commands.

We define a network security model, MODEL, as follows :

$$MODEL =< S, O, s_0 >$$

where S is the set of States, O is the set of system Operations and s_0 is the initial system state.

Let us first define the basic sets used to describe the model:

- Sub : Set of all network subjects. This includes the set of all Users (Users) and all Processes (Procs) in the network. That is : $Sub = Procs \cup Users$

- Obj : Set of all network objects. This includes both the set of Network Components (NC) and Information Units (IU). That is : $Obj = NC \cup IU$.
  Typically, the set of Network Components includes virtual machines (VMs), Input-Output Devices (IOD) and Output Devices (OD) whereas Information Units include files and messages. That is : $NC = VMs \cup IOD \cup OD$

- $SCls$: Set of Security Classes. We assume that a partial ordering relation $\geq$ is defined on the set of security classes.

- Rset : Set of user roles. This includes for instance the role *Admin* dedicated to the administrator of the network who is typically the administrator of the whole virtualized architecture.

We use the notation x_s, to denote the element x at state s.

System State

We only consider the security relevant state variables. Each state $s \in S$ can be regarded as a 11-tuple as follows :
$s =< Sub_s, Obj_s, authlist, connlist, accset, subcls, objcls,$
$curcls, subrefobj, role, currole, curvm >$

Let us now briefly describe the terms involved in the state definition :
- Sub_s and Obj_s defines respectively the sets of subjects and objects at the state s.
- $authlist$ is a set of elements of the form (sub, nc) where $sub \in Sub_s$ and $nc \in Obj_s$. The existence of an element (sub_1, nc_1) in the set indicates that the subject sub_1 has an access right to connect to the network component nc_1.
- $connlist$ is again a set of elements of the form (sub, nc). This set gives the current set of authorized connections at that state.
- $accset$ is a set of elements of the form $(sub, iuobj)$, where $sub \in Sub_s$, and $iuobj \in Obj_s$. The existence of an element $(sub_1, iuobj_1)$ in the set indicates that the subject sub_1 has an access right to bind to the object $iuobj_1$.
- $subcls : Sub \rightarrow SCls$, is a function which maps each subject to a security class.
- $objcls : Obj \rightarrow SCls$, is a function which maps each object to a security class.
- $curcls : Sub \rightarrow SCls$, is a function which determines the current security class of a subject.
- $subrefobj : Sub \rightarrow PS(Obj)$, is a mapping which indicates the set of objects referenced by a subject at that state.
- $role : Users \rightarrow PS(Rset)$, gives the authorized set of roles for a user.
- $currole : Users \rightarrow Rset$, gives the current role of a user.
- $curvm : Users \rightarrow NC$, is a function which gives the VM in which a user is logged on.
- $view : Sub \rightarrow Obj$, is a function that determines the objects that can be viewed by a subject.

Secure State

To define the necessary conditions for a secure state, we need to consider the different phases gone through by the system during its operation, we focus on typical network operations :

Login Phase : We require that if the user is logging through a VM, he must have appropriate clearance with respect to the VM. That is, the security class of the user must be above the security class of the VM in which the user is attempting to log on. In addition, the current security class of the user must be below the maximum security class of that user and the role of the user must belong to the authorized role set allocated to that user. So we have the following constraint:

- *Proposition 1 : Login Constraint* :
A state s satisfies the Login Constraint if $\forall x \in Users$:

- $subcls(x) \geq objcls(curvm(x))$
- $subcls(x) \geq curcls(x)$

Connect Phase : Having logged-on to the virtual network, a user may wish to establish a connection with another network component (VM or I/O VM). In determining whether such a connection request is to be granted, both network discretionary and mandatory security policies on connections need to be satisfied. The discretionary access control requirement is specified using the authorization list which should contain an entry involving the requesting subject and the network component. If the network component in question is a VM then the current security class of the subject must at least be equal to the lowest security class of that VM. On the other hand, if the network component is an output device, then the security class of the subject must be below the security class of that component. Hence we have the following constraint:
Proposition 2 : Connect Constraint :
A state s satisfies the Connect Constraint if $\forall(sub, nc) \in connlist$:

- $(sub, nc) \in authlist$
- if $nc \in VMs$, then $curcls(sub) \geq objcls(nc)$
- if $nc \in OD$ then $objcls(nc) \geq curcls(sub)$

Other Conditions We require two additional conditions :
(1) The classification of the information that can be "viewed" through an I/O device must not be greater than the classification of that device.
(2) The role of the users at a state belong to the set of authorized roles. Now we can give the definition of a secure state as follows :
- **Definition** : A state s is *Secure* if :

- s satisfies the *Login* Constraint
- s satisfies the *Connect* Constraint
- $\forall z \in (IOD_s \cup OD_s), \forall x \in IU_s,$
$x \in view(z) \Rightarrow objcls(z) \geq subcls(x).$

We assume that the initial system state s_0 is defined in such a way that it satisfies all the conditions of the secure state described above.

5.7 Operations and their security requirements

In this section we will present the security constraints that must be satisfied by the different operations performed by the user of the virtual network : this includes vitual machines management operations done by the administrator (create/remove a VM, checkpoint/restore a VM), network operations such as *connect* and *bind* operations and finally operations related to the policy management (assign a security class to an object, assign a role to a user, etc).

5.7.1 Virtual machines managment operations

Create a new VM : Only the administrator of the virtual network is allowed to create new virtual machines. Once created, a new VM must be labelled by a security class which should be dominated by the security class of the Dom_0. This leads to the following constraints : if a subject *sub* wants to create a new virtual machine *newVM* then:

- $Admin \in role(sub)$ and $currole(sub) = Admin$
- $objcls(Dom_0) \geq objcls(newVM)$
- $NC'_s = NC_s \cup \{newVM\}$

Remove a VM : Only a user with the role *Admin* is allowed to remove virtual machines. The only VM that cannot be removed is the administration VM, even by the administrator of the system (this is the normal case, but when we have other sensitive VMs such as the surveillance VM in our architecture, we can add restriction concerning the removal of this VM). This leads us to define the set *sensitiveVMs* which includes the Dom_0 in the case of Xen, the surveillance VM and may include other important VMs that cannot be removed. We have the following constraints : if a user *sub* wants to remove a virtual machine *VM* then:

- $currole(sub) = Admin$
- $VM \notin sensitiveVMs$
- $authlist'_s = authlist_s \setminus (x, VM)$, where $x \in Sub$.
- $connlist'_s = connlist_s \setminus (x, VM)$, where $x \in Sub$.

After removing the VM the lists *authlist* and *connlist* are updated by removing the pairs where the deleted VM occurs.

Checkpoint and restore a VM : These functionalities are offred by most modern hypervisors. By creating checkpoints for a virtual machine, one can restore the virtual machine to a previous state. A typical use of checkpoints is to create a temporary backup before applying updates to the VM. The *restore* operation enables to revert the virtual machine to its previous state if the update fails or adversely affects the virtual machine. Any user can checkpoint and restore his own VM, the user with the role *Admin* can do this with any VM. To make sure that these two operations do not represent security threats, we need the following constraints.

If a user *sub* wants to checkpoint a virtual machine *vm1* then:

- $curvm(sub) = vm1$ or $currole(sub) = Admin$
- $VM \neq Dom_0$

In addition to these constraints, when restored, a VM must keep the same security class as before the checkpoint. Let s and z be respectively the states of the system bebore and after the checkpoint, we should have :

- $objcls_z(vm1) = objcls_s(vm1)$

5.7.2 Network operations

Connect operation : The operation $connect(sub, nc)$ allows a subject sub to connect to a remote network entity nc. From the Connect Constraint given earlier, for this operation to be secure, we require that :

- $(sub, nc) \in authlist$
- if $nc \in VMs$, then $curcls(sub) \geq objcls(nc)$
  or
  if $nc \in OD$ then $objcls(nc) \geq subcls(sub)$

After the operation is performed we should have : $(sub, nc) \in connlist'$ and $nc \in subrefobj(sub)$.

Having connected to a remote VM, a subject can perform operations which allow the manipulation of information objects. We envisage the information manipulation phase to consist of two stages : a binding stage and a manipulation stage. The binding stage involves a subject linking itself to the VM on which the operation is to be performed. At the manipulation stage, typically the operations include those operations defined by the Bell-LaPadula model such as *read*, *append*, *write* and *execute*. In our model, we will only consider one basic manipulation operation which allows the transfer of an object from one VM to another, as this is perhaps the most important operation from the network point of view. This operation causes information to flow from one entity to another over the network. (In fact, this operation will form part of other operations as well. For instance, consider a read operation, whereby a user reads a file stored in a remote entity. This operation must include the transfer of the file from the remote network component to the local network component in which the user resides.) There are also other operations which modify certain security attributes of objects and subjects. In the usual computer security model, these include operations for assigning and changing security classes to users and information objects and assigning and modifying access sets for information unit objects. Note that in general for any operation to be performed, the subject must have authorized access to the connection with the remote entity. That is, the *Connect Constraint* must be satisfied to begin with.

Bind operation : The operation $bind(iuobj, nc)$ allows a subject sub to link an information object $iuobj$ in a network component nc. The constraints that must be satisfied by this operation are:

- $(sub, iuobj) \in accset(iuobj)$
- $curcls(sub) \geq objcls(iuobj)$
- for any $sb \in Sub_s$, $iuobj \notin subrefobj(sb)$

After the operation is performed, we should have $iuobj \in subrefobj'(sub)$. Where $subrefobj'$ refers to the new state s'.

Note that we have included a simple access control based on *accset* at the remote network component. In practice, a comprehensive access control mechanism is likely to be provided by a mechanism located in the remote entity. Note that we could have defined the bind operation as part of the connect operation, thereby making the connection to a particular information object at the connect stage rather than to a network component.

Transfer operation :
The operation *transfer(iuobj1,nc1,iuobj2,nc2)* allows a subject sub to append the contents of an information unit object $iuobj1$ in a network component object $nc1$ to the contents of another information unit object $iuobj2$ in a network component object $nc2$. For this operation to be secure, we require that :

- $objcls(iuobj2) \geq objcls(iuobj1)$
- $curcls(sub) \geq objcls(iuobj1)$

Further both $iuobj1$ and $iuobj2$ referenced by the subject sub must not be referenced by any other object. That is, for any $sb \in Sub_s$, $sb \neq sub$, $iuobj1$ and $iuobj2 \notin subrefobj(sb)$. Also $iuobj1$ and $iuobj2 \in subrefobj(sub)$.

After the operation is performed the security classes of the objects $iuobj1$ and $iuobj2$ remain unchanged. That is,

- $objcls'(iuobj1) = objcls(iuobj1)$
- $objcls'(iuobj2) = objcls(iuobj2)$

where $objcls'$ refers to the new state s'.

Unbind : The operation $unbind(sub, iuobj)$ allows a subject sub to release its link to an information object $iuobj$. That is, before this operation $iuobj \in subrefobj(sub)$. After the operation, we have $iuobj \notin subrefobj(sub)$.

5.7.3 Security-related operations

Let us now consider some typical operations which modify certain security attributes of objects and subjects. In the usual computer security model, these include operations for assigning and changing security classes to users and information objects and assigning and modifying access sets for information unit objects. In the case of our network security model, we need additional operations such as to assign security classes of network component objects, to set authorization list and operations, to assign and change roles of the users. Let us consider some of these operations. We will use the notation x and x' to refer to x at states s and s'.

Assign-cls-nc : The operation *assign-cls-nc(nc,scls)* allows a subject sub to set the security class of a network component object nc, to $scls$. That is, $objcls'(nc) = \{scls\}$. This operation can be performed only when the component is not being used. Further, only the virtualized system administrator (Admin) has the authority to set the security class of a network component object. That is, if this operation is to be performed at state s then the following must be true :
If there exists any $nc \in NC$ such that $objcls(nc) \neq objcls'(nc)$ then :

- for any subject $sb \in Sub_s (sb \neq sub)$, $nc \notin subrefobj(sb)$ and $(sb, nc) \notin connlist$
- $Admin \in role(sub)$ and $currole(sub) = Admin$.

Assign-cls-user : The operation *assign-cls-user(usr, scls)* allows a subject sub to set the security class of a user, usr, to $scls$. That is, $subcls'(usr) = scls$. Typically the conditions we require for this operation to be secure are :
If there exists any $usr \in Users$ such that $subcls(usr) \neq subcls'(usr)$ then :

- $Admin \in role(sub)$ and $currole(sub) = Admin$
- if the user is logged in at state s ($i.e\ usr \in Users_s$), then $subcls'(usr) \geq curcls(usr)$. (note that $curcls'(usr) = curcls(usr)$).

Assign-curcls-user : The operation *assign-curcls-user(usr, scls)* allows a subject sub to set the current security class of a user usr to $scls$. That is, $curcls'(usr) = scls$. The conditions required for this operation to be secure can be described as follows : If there exists any $usr \in Users$ such that $curcls(usr) \neq curcls'(usr)$ then :

- $Admin \in role(sub)$ and $currole(sub) = Admin$ or $usr = sub$.
- $subcls(usr) \geq curcls'(usr)$
- if the user is logged onto a terminal at state s, then $curcls'(usr) \geq objcls(curvm(usr))$.
- if the user is connected to a network component at state s which is not an output device, that is, $(usr, nc) \in connlist$ and $nc \notin OD$, then $curcls'(usr) \geq objcls(nc)$
- if the user is logged in and is connected to an output device, that is, $(usr, nc) \in connlist$ and $nc \in OD$, then $objcls(nc) \geq curcls'(usr)$.

Assign-role-user : The operation *assign-role-user(usr,rlset)* allows a subject sub to assign a role set $rlset$ to a user usr, That is $role'(usr) = \{rlset\}$. For this operation to be secure, we need the following condition to be hold :
If there exists any $usr \in Users$ such that $role(usr) \neq role'(usr)$ then :

- $Admin \in role(sub)$ and $currole(sub) = Admin$
- if the user is logged in at state s, then $currole(usr) \in role'(usr)$.

Assign-currole-user : The operation *assign-currole-user(usr,rl)* allows a subject sub to change the current role of a user usr to rl. That is, $currole'(usr) = rl$. The security requirements of this operation are :
If there exists any $usr \in Users$ such that $currole(usr) \neq currole'(usr)$ then :

- Only the user himself or a subject whose current role is $Admin$ has the authority to change the current role of the user. That is, $Admin \in role(sub)$ and $currole(sub) = Admin$ or $usr = sub$.
- the new role rl must be in the set of authorized roles of the user. That is, $currole'(usr) \in role(usr)$.

Setauthlist : The operation $setauthlist(al)$ allows a subject to set the authorization list. The $authlist$ is of the form (sb, nc), where $sb \in Sub$ and $nc \in NC$. Again, this operation can only be performed by a subject who can act as a $Admin$. That is, if $al \notin authlist$ and $al \in authlist'$ then $Admin \in role(sub)$ and $currole(sub) = Admin$ where sub is the subject performing this operation.

5.8 Conclusion and Further Work

The flexibility that makes virtual networks such a useful technology can also undermine security within organizations and individual hosts. Current research on virtual machines has focused largely on the implementation of virtualization and its applications. But less effort was done for securing communication under virtualized systems. We proposed in this chapter a security policy model for communication under virtual networks, this model can be implemented easily under most virtualized architectures. Currently, we are extending our security policy to cover not only local networks, but also wide networks composed of many virtualized systems involving policy agreements and the protection of information flows that leave the control of the local hypervisor. We need to establish trust into the semantics and enforcement of the security policy governing the remote hypervisor system before allowing information flow to and from such a system.

Chapter 6

Conclusion and Perspectives

It is hard not to love virtualization. The ability to create dozens of virtual servers (or appliances) as files within a single physical server can cut power consumption, save space, make IT admins jobs easier, and allow them create separate environments for testing new applications at will. No wonder this is one of the fastest growing technologies in businesses large and small. But everything has its drawbacks, and virtualization is no exception. Nowadays, virtualization means paying more attention to security.

In this dissertation, we interested ourselves in the security of virtualized systems. We proposed ideas, approaches and methods that increase the security of such platforms and most of the time prevent some potential threats.

In this concluding chapter, we discuss other research directions. We believe that the presented results can be improved at many levels and sometimes adapted to more security threats.

In chapter 3, we presented an implementation of a decentralized supervision system that offers the ability to control all the running virtual machines from outside by deploying an IDS and its sensors. This architecture can be used either to protect the VMs or even to offer a secure decentralized system for simple users. We feel that a more hypervisor-independant implementation would be more interesting, because for now our implementation works only with the Xen hypervisor, and it would be a plus to adapt it to other virtualization solutions. Another important improvement would be to encrypt the messages sent from the sensors to Orchids : actually the data sent via in the VLAN is unencrypted and a possible threat can be a sniffing mechanism that discovers a lot of sensitive information about the target IDS, the surveillance VM, etc...which represents a potential risk that we have to avoid.

Moroever, we have seen in this chapter that our implementation reveals a considerable lack of efficiency against fast attacks on remote VMs. This is due to the latency of the virtual network (which is actually lower then in real physical networks). One can suggest to install Orchids directly on the target VM. This makes our architecture loose its most important features such as remote control, decentralization and exposes the IDS to attacks. For now, we have no idea how to resolve this issue.

It would be also challenging to explore ways to avoid killing VMs in case of DoS attacks in order to preserve a good level of service continuity.

In chapter 4, we aimed to protect sensitive resources such as the Domain0, the VM that the administrator uses to do all critical administration actions such as creating/killing VMs, making checkpoints etc...The most convenient idea was to study the existence of security polcies that control the access to these resources and propose an easy approach that permits the writing of policies and deploying them quickly and automatically. To this end, we introduced a high-

level language allowing to write suitable security policies, it is a fragment of LTL with new past operators. We showed how this language is more convenient to our aim then temporal languages with future operators. Then we introduced an algorithm based on history variables allowing the automatic translation of policies into EFSA describing attacks. This permits to feed the attacks base of the IDS to make it able to detetct and stop more attacks. Our objective was not to write policies, this depends on the administrator needs which can change over time, but we aimed to design a high-level procedure that can be valuable and useful for different users, platforms and needs. This contribution can be improved at many levels. First, some restrictions related to the language we proposed can be studied and removed especially while writing complex formulas requiring recursive calls to the same operator. Second, for the moment, some keywords was defined in order to facilitate the translation procedure, but this is still not enough : the keywords list should be enriched and fixed with an aim to give more flexibility for formulas writing. Another idea is to improve the syntax from the IDS side (*i.e.* the desctription of the EFSA in the form of Orchids rules). This was done for instance for the "if" statements that have sometimes different semantics depending on what we need to check (either the occurence of an event or a simple expression evaluation). Another important research direction related to this contribution is, given a formula written in our language, to be able to check that this formula will not cause a denial of service due to its translation complexity. This requires a static analysis procedure that takes as input the formula and returns back an indication about the risk related to the translation and deployement of this formula. Another related subject will be the following : given a linear model (events e_1,...,e_n), a fixed time k between 1 and n, and a formula F in our logic, to be able to decide if F is true at the moment k in this model.

In chapter 5, we proposed a multi-level security policy model for virtual LANs. We aimed to design a generic model that represents the most important network features of a virtual network of VMs. This model can be implemented and used to guarantee the security of communication. This is important, since the architecture presented in chapter 3 relies on a virtual LAN for communicating information between the IDS and its sensors. We take into consideration the different components of a virtual LAN with not only the different network communication operations, but also we added to our model some other management and securty operations. We study also security management in this chapter. For the moment, the security requirements are specified, and the security policy that can be developped around this model is defined. The important improvement that can perfectly complement our model will be to work on the verification of the system security at an instant t while taking into account the actions performed on the system. A possible idea will be to use well-known verification and model checking procedures to verify the security of this model at each stage reached by system actions. Another interesting improvement would be to extend our model to large scale networks composed by many VLANs. This can introduce more complexity to the modelling approach, but represents an interesting research direction.

Finally we can say that a lot of work can be done for enfancing the security of virtualized systems since many issues are already existing. The question will be : how long this technology will keep convincing users to adopt it in order to maintain their system security needs?

Appendix A

The Xen Hypervisor

A.1 Introduction

Xen is an open-source para-virtualizing virtual machine monitor (VMM), or hypervisor, for the x86 processor architecture. Xen can securely execute multiple virtual machines on a single physical system with close-to-native performance. Xen facilitates enterprise-grade functionality, including : virtual machines with performance close to native hardware, live migration of running virtual machines between physical hosts, Intel and AMD Virtualization Technology for unmodified guest operating systems (including Microsoft Windows) and excellent hardware support (supports almost all Linux device drivers).

A.2 Booting a Xen System

Booting the system into Xen will bring you up into the privileged management domain, Domain0. At that point you are ready to create guest domains and boot them using the *xm create* command.

A.2.1 Booting Domain0

After installation and configuration is complete, reboot the system and and choose the new Xen option when the Grub screen appears. What follows should look much like a conventional Linux boot. The first portion of the output comes from Xen itself, supplying low level information about itself and the underlying hardware. The last portion of the output comes from XenLinux. When the boot completes, you should be able to log into your system as usual. If you are unable to log in, you should still be able to reboot with your normal Linux kernel by selecting it at the GRUB prompt. The first step in creating a new domain is to prepare a root filesystem for it to boot. Typically, this might be stored in a normal partition, an LVM or other volume manager partition, a disk file or on an NFS server. A simple way to do this is simply to boot from your standard OS install CD and install the distribution into another partition on your hard drive.

A.2.2 Booting Guest Domains

Before you can start an additional domain, you must create a configuration file. We provide two example files which you can use as a starting point:

- */etc/xen/xmexample1* is a simple template configuration file for describing a single VM.

- */etc/xen/xmexample2* file is a template description that is intended to be reused for multiple virtual machines. Setting the value of the *vmid* variable on the *xm* command line fills in parts of this template.

There are also a number of other examples which you may find useful. Copy one of these files and edit it as appropriate. Typical values you may wish to edit include:

kernel Set this to the path of the kernel you compiled for use with Xen (e.g. kernel = "/boot/vmlinuz-2.6-xenU")

memory Set this to the size of the domain's memory in megabytes (e.g. memory = 64)

disk Set the first entry in this list to calculate the offset of the domain's root partition, based on the domain ID. Set the second to the location of /usr if you are sharing it between domains (e.g. disk = ['phy:your hard drive%d,sda1,w' % (base partition number + vmid), 'phy:your usr partition,sda6,r']

dhcp Uncomment the *dhcp* variable, so that the domain will receive its IP address from a DHCP server (e.g. dhcp="dhcp")

You may also want to edit the vif variable in order to choose the MAC address of the virtual ethernet interface yourself. For example: *vif = ['mac=00:16:3E:F6:BB:B3']* If you do not set this variable, xend will automatically generate a random MAC address from the range 00:16:3E:xx:xx:xx, assigned by IEEE to XenSource as an OUI (organizationally unique identifier). XenSource Inc. gives permission for anyone to use addresses randomly allocated from this range for use by their Xen domains.

A.2.3 Starting / Stopping Domains Automatically

It is possible to have certain domains start automatically at boot time and to have dom0 wait for all running domains to shutdown before it shuts down the system. To specify a domain is to start at boot-time, place its configuration file (or a link to it) under /etc/xen/auto/.

A Sys-V style init script for Red Hat and LSB-compliant systems is provided and will be automatically copied to /etc/init.d/ during install. You can then enable it in the appropriate way for your distribution. For instance, on Red Hat:

```
# chkconfig --add xendomains
```

By default, this will start the boot-time domains in runlevels 3, 4 and 5. You can also use the service command to run this script manually, e.g:

```
# service xendomains start
```

Starts all the domains with config files under /etc/xen/auto/.

```
# service xendomains stop
```

Shuts down all running Xen domains.

A.3 Network Configuration

For many users, the default installation should work "out of the box". More complicated network setups, for instance with multiple Ethernet interfaces and/or existing bridging setups will require some special configuration. The purpose of this section is to describe the mechanisms provided by xend to allow a flexible configuration for Xen's virtual networking.

A.3.1 Xen virtual network topology

Each domain network interface is connected to a virtual network interface in dom0 by a point to point link (effectively a "virtual crossover cable"). These devices are named vif<domid>.<vifid> (e.g. vif1.0 for the first interface in domain 1, vif3.1 for the second interface in domain 3). Traffic on these virtual interfaces is handled in domain 0 using standard Linux mechanisms for bridging, routing, rate limiting, etc. Xend calls on two shell scripts to perform initial configuration of the network and configuration of new virtual interfaces. By default, these scripts configure a single bridge for all the virtual interfaces. Arbitrary routing / bridging configurations can be configured by customizing the scripts, as described in the following section.

A.3.2 Xen networking scripts

Xen's virtual networking is configured by two shell scripts (by default network-bridge and vif-bridge). These are called automatically by xend when certain events occur, with arguments to the scripts providing further contextual information. These scripts are found by default in /etc/xen/scripts. The names and locations of the scripts can be configured in /etc/xen/xend-config.sxp.

network-bridge This script is called whenever xend is started or stopped to respectively initialize or tear down the Xen virtual network. In the default configuration initialization creates the bridge 'xen-br0' and moves eth0 onto that bridge, modifying the routing accordingly. When xend exits, it deletes the Xen bridge and removes eth0, restoring the normal IP and routing configuration.

vif-bridge This script is called for every domain virtual interface and can configure firewalling rules and add the vif to the appropriate bridge. By default, this adds and removes VIFs on the default Xen bridge. Other example scripts are available (network-route and vif-route, network-nat and vif-nat). For more complex network setups (e.g. where routing is required or integrate with existing bridges) these scripts may be replaced with customized variants for your site's preferred configuration.

Appendix B

The SELinux Auditd System

Modern Linux kernel (2.6.x) comes with *auditd* daemon. It is responsible for writing audit records to the disk. It allows one to comprehensively log and track access to files, directories, and resources of the system, as well as trace system calls. It enables the monitoring of the system for application misbehavior or code malfunctions. By creating a sophisticated set of rules including file watches and system call auditing, security officers can make sure that any violation of security policies is noted and properly addressed.

The kernel part is included in Linux, and activated in most Linux distributions (including Squeeze). The following options must be enabled in the kernel :

```
CONFIG_AUDIT=y
CONFIG_AUDITSYSCALL=y
CONFIG_AUDIT_WATCH=y
CONFIG_AUDIT_TREE=y
```

To be able to use it, we need to install the userspace tools :

```
[user@laptop tmp] aptitude install auditd audispd-plugins
```

B.1 Audit rules

The main command to control audit rules is auditctl To show the current status of the audit system:

```
[user@laptop tmp] auditctl -s
```

To list the rules :

```
[user@laptop tmp] auditctl -l
LIST_RULES: exit,always arch=3221225534 (0xc000003e) watch=/etc/hosts syscall=open
```

Removing all rules :

```
[user@laptop tmp] auditctl -D
No rules
```

B.2 Processes

Now, suppose we want to log the creation of all new processes from a specific user :

```
[user@laptop tmp] auditctl -a exit,always -S execve -F uid=1000
```

Log all executions of a specific program (any user) :

```
[user@laptop tmp] auditctl -A exit,always -F path=/path/to/executable
                           -S execve
```

Watching for ptrace system calls (very verbose, one trace call can result in many ptrace syscals) :

```
[user@laptop tmp] auditctl -a entry,always -F arch=b64 -S ptrace -k info_scan
```

The *-k* option is used to specify a custom key for this event (31 chars max). This can be used to filtering when searching for events. Now, a funnier use of the filters: monitor execution of all programs with the setuid bit and owner root. Finding these is easy, because the uid running the program will be non-0 while the effective uid will be 0 :

```
[user@laptop tmp] auditctl -A exit,always -F arch=b64 -F euid=0 -F 'uid!=0' -S execve
```

Log all syscalls done by some program (emulate strace, without the nice decoding of all arguments) :

```
[user@laptop tmp] auditctl -a exit,always -S all -F pid=19845
```

B.3 Files

Audit all files opened by some user :

```
[user@laptop tmp] auditctl -a exit,always -S open -F uid=1000
```

Audit all accesses to a specific file :

```
[user@laptop tmp] auditctl -a exit,always -F arch=b64 -F path=/etc/hosts -S open
```

Log all unsuccessful file open calls :

```
[user@laptop tmp] auditctl -a exit,always -S open -F success=0
```

In the same idea, log all unsuccessful writes :

```
[user@laptop tmp] auditctl -a exit,always -S write -F success=0
```

B.4 Reporting

To see the events, either run : *"tail -F /var/log/audit/audit.log"*

```
type=SYSCALL msg=audit(1308608275.954:25072): arch=c000003e syscall=59
success=yes exit=0 a0=7fff3e038690 a1=7faaa6418e80 a2=d99190 a3=0 items=2
ppid=6854 pid=14762 auid=4)
type=EXECVE msg=audit(1308608275.954:25072): argc=2 a0="ls" a1="--color=auto"
type=CWD msg=audit(1308608275.954:25072):  cwd="/home/pollux/GIT/admin/SELINUX"
```

It is clear that the result is very verbose. One can also recognize SELinux information, and that is indeed the case since SELinux is using auditd a lot. We can also use the very powerful *ausearch* and *aureport* commands.

Get the list of ptrace syscalls (monitored as above) for the last 5 minutes :

```
[user@laptop tmp] ausearch -ts recent -sc ptrace -i
```

→"ts" is the time start option, "sc" is for syscall

Since we specified a custom key when creating the filter, we are also able to query events based on the key :

```
[user@laptop tmp] ausearch -ts -k info_scan -i
```

Search by user id :

```
[user@laptop tmp] ausearch -ui 1000 -ts recent
```

Search in a time range :

```
[user@laptop tmp] aureport -f --start 06/21/2011 23:00:00 --end 06/21/2011 23:10:00
```

Report on watched files :

```
[user@laptop tmp] aureport -f -ts recent
```

Output will be similar to :

```
[user@laptop tmp] aureport -f -ts recent
1. 06/21/2011 20:54:01 /root 4 no /bin/dash -1 28515
```

Here is the description of the columns (for the files report):

- first column is an index
- 2nd is the date of the event
- 3rd is the time of the event
- 4th is the file name
- 5th is the syscall id (use -i to make aureport display strings)
- 6th is the result of the system call
- 7th is the process that triggered the event
- 8th is the actual/audit uid (the initial uid of the session, which remains the same even if you change user with su after, for ex)
- 9th is the event id

Bibliography

[1] H. Benzina. Towards Designing Secure Vitualized Systems. In Proceedings of The Second International Conference on Digital Information and Communication Technology and its Applications (DICTAP 2012), Bangkok, Thailand. IEEE Computer Society Press, 2012.

[2] H. Benzina. A Network Policy Model for Virtualized Systems. In Proceedings of The Seventeenth IEEE Symposium on Computers and Communication (ISCC 2012). Cappadocia, Turkey. IEEE Computer Society Press, 2012.

[3] H. Benzina. Logic in Virtualized Systems. In Proceedings of the First International Conference on Computer Applications and Network Security (ICCANS 2011), Malé, Maldives. IEEE Computer Society Press, 2011.

[4] H. Benzina. Securing Hypervisors through Temporal Logic and Security Policies. Workshop on Formal methods for specifying and verifying critical systems 2011. Tunis, Tunisia.

[5] H. Benzina and J. Goubault-Larrecq. Some Ideas on Virtualized Systems Security, and Monitors. In The third International Workshop on Autonomous and Spontaneous Security (SETOP 2010), Athens, Greece. Springer LNCS 6514.

[6] Gerald J. Popek and Robert P. Goldberg (1974). Formal Requirements for Virtualizable Third Generation Architectures. Communications of the ACM 17 (7): 412-421.

[7] The GNU HUrd System, 2012. `http://www.gnu.org/software/hurd`.

[8] Vmware, 2012. `http://www.vmware.com/`.

[9] Virtualbox, 2012. `http://www.virtualbox.org/`.

[10] Virtual PC, 2012. `http://www.microsoft.com/windows/virtual-pc/default.aspx`

[11] Qemu, 2012. `http://www.qemu.org/`.

[12] Xen, 2005–2012. `http://www.xen.org/`.

[13] Anderson, J. (1980). Computer security threat monitoring and surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania.

[14] Denning, D. E. (1987). An intrusion-detection model. 13(2):222-232.

[15] Meier, M. (2004). Intrusion detection systems list and bibliography.

[16] Bace, R. et Mell, P. (2001). Intrusion detection systems. Technical report, National Institute of Standards and Technology (NIST).

[17] Wood, M. et Erlinger, M. (2007). Intrusion detection message exchange requirements. IETF Intrusion Detection Exchange FormatWorking Group. Request for Comments. Reference : rfc4766.

[18] F. Laroussinie, N. Markey, and Ph. Schnoebelen. Temporal logic with forgettable past. In Proc. 17th IEEE Symp. Logic in Computer Science (LICS'2002), Copenhagen, Denmark, July 2002, pages 383-392. IEEE Comp. Soc. Press, 2002.

[19] Bishop, M. (2003). Computer Security Art and Science. ISBN 0201440997. Addison-Wesley Professional.

[20] Eckmann, S., Vigna, G. et Kemmerer, R. (2000). Statl : An attack language for state-based intrusion detection.

[21] Pouzol, J.-P. et Ducass, M. (2002). Formal specification of intrusion signatures and detection rules. In CSFw'02 : Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02), page 64, Washington, DC, USA. IEEE Computer Society.

[22] J. Goubault-Larrecq and J. Olivain. A smell of Orchids. In M. Leucker, editor, *Proceedings of the 8th Workshop on Runtime Verification (RV'08)*, Lecture Notes in Computer Science, pages 1–20, Budapest, Hungary, Mar. 2008. Springer.

[23] Carrasco, R. C. et Oncina, J. (1994). Learning stochastic regular grammars by means of a state merging method. In Carrasco, R. C. et Oncina, J. : Grammatical Inference and Applications, Second International Colloquium, ICGI-94, Alicante, Spain, September 21-23, 1994, Proceedings, pages 139-152. Springer.

[24] Kosoresow, A. P. et Hofmeyr, S. A. (1997). Intrusion detection via system call traces. 14(5):35-42.

[25] Ron, D., Singer, Y. et Tishby, N. (1996). The power of amnesia : Learning probabilistic automata with variable memory length. 25(2-3):117-149.

[26] Kumar, S. et Spafford, E. H. (1994). A Pattern Matching Model for Misuse Intrusion Detection. In Proceedings of the 17th National Computer Security Conference, pages 11-21.

[27] Vigna, G. et Kemmerer, R. A. (1998). Netstat : A network-based intrusion detection approach. In 14th Annual Computer Security Applications Conference (ACSAC 1998), 7-11 December 1998, Scottsdale, AZ, USA, pages 2536. IEEE Computer Society.

[28] Mé, L. (1998). Gassata, a genetic algorithm as an alternative tool for security audit trail analysis. In the first international workshop on Recent Advances in Intrusion Detection.

[29] DuMouchel, W. et Schonlau, M. (1998). A fast computer intrusion detection algorithm based on hypothesis testing of command transition probabilities. In KDD, pages 189-193.

[30] Mounji, A. et Charlier, B. L. (1997). Continuous assessment of a unix configuration : Integrating intrusion detection and configuration analysis. In Proceedings of the Network and Distributed System Security Symposium, NDSS 1997, San Diego, California, USA. IEEE Computer Society.

[31] Michel, C. et Mé, L. (2001). Adele : An attack description language for knowledge-based intrusion detection. In Dupuy, M. et Paradinas, P. : Trusted Information : The New Decade Challenge, IFIP TC11 Sixteenth Annual Working Conference on Information Security (IFIP/Sec'01), June 11-13, 2001, Paris, France, pages 353-368. Kluwer.

[32] Michael Le, Yuval Tamir. ReHype: enabling VM survival across hypervisor failures. VEE 2011: 63-74

[33] Cuppens, F. et Ortalo, R. (2000). Lambda : A language to model a database for detection of attacks. In Debar, H., Mé, L. et Wu, S. F., In Recent Advances in Intrusion Detection, Third International Workshop, RAID 2000, Toulouse, France, October 2-4, 2000, Proceedings, pages 197-216. Springer.

[34] Mendel Rosenblum. The reincarnation of virtual machines. Queue, 2(5):34-40, August 2004.

[35] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: current technology and future trends. Computer, 38(5):39-47, May 2005.

[36] Blue Lane Technologies INC. 2012. `http://www.bluelane.com/`.

[37] Joanna Rutkowska. Subverting Vista kernel for Fun and Profit. Black Hat 2006, Las Vegas, USA.

[38] Trent Jaeger, Reiner Sailer, and Yogesh Sreenivasan. Managing the Risk of Covert Information Flows in Virtual Machine Systems. In Proceedings of the 12th ACM Symposium on Access Control Models and Technologies, pages 81-90, January 2007.

[39] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The Flawed Assumption of Security in Modern Computing Environments. In Proceedings of the 21th National Information Systems Security Conference, October 1998.

[40] Virgil D. Gligor. A Guide to Understanding Covert Channel Analysis of Trusted Systems. Technical report, NATIONAL COMPUTER SECURITY CENTER, November 1993.

[41] KVM, 2012. `http://www.linux-kvm.org/page/Main_Page`

[42] Jenni S. Reuben. A survey on virtual machine security. Technical report, Helsinki University of Technology, October 2007.

[43] Joel Kirch. Virtual Machine Security Guidelines. The Center for Internet Security, September 2007.

[44] Travis Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. Technical report, Google, Inc.

[45] Peter Ferrie. Attacks on Virtual Machine Emulators. In Proceedings of the AVAR 2006 Conference, pages 128-143, December 2006.

[46] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In Proceedings of the 21st National Information Systems Security Conference, pages 303-314, Oct. 1998.

[47] S. E. Minear. Providing Policy Control Over Object Operations in a Mach Based System. In Proceedings of the Fifth USENIX UNIX Security Symposium, pages 141-156, June 1995.

[48] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In Proceedings of the Eighth USENIX Security Symposium, pages 123-139, Aug. 1999.

[49] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert van Doorn, John L. Griffin, and Stefan Berger. Secure hypervisor approach to trusted virtualized systems. In 9. Deutscher IT-Sicherheitskongress, Bunde- samt fur Sicherheit in der Informationstechnik, May 2005.

[50] Common Criteria. Common Criteria for Information Technology Security Evaluation. `http://www.commoncriteriaportal.org`.

[51] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In Thirteenth USENIX Security Symposium, pages 223-238, August 2004.

[52] Kurniadi Asrigo, Lionel Litty, David Lie. Using VMM-based sensors to monitor honeypots. VEE 2006: 13-23

[53] J. P. Anderson et. al. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol? 1+2, Air Force Systems Command, USAF, 1972.

[54] Kernel Based Virtual Machine 2012. `http://www.linux-kvm.org`

[55] Paul A. Karger, Mary E. Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A VMM Security Kernel for the VAX Architecture. In Proceedings of the 1990 IEEE Computer Society Symposium on Security and Privacy, pages 2-19, May 1990.

[56] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a Virtual Machine-based Platform for Trusted Computing. SIGOPS Operating Systems Review, 37(5):193-206, 2003.

[57] Sören Bleikertz, Thomas G., Matthias Schunter, Konrad Eriksson: Automated Information Flow Analysis of Virtualized Infrastructures. ESORICS 2011: 392-415

[58] Nettop, 2004. `http://www.nsa.gov/research/tech_transfer/fact_sheets/nettop.shtml`.

[59] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium*, San Diego, CA, Feb. 2003.

[60] Kurshan, R.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeon University Press (1994).

[61] Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logic of programs. Journal of Computer and System Sciences 32, 1986, 183-221.

[62] Vardi, M.: An automata-theoretic approach to linear temporal logic. In: Logics for Concurrency: Structure versus Automata. Volume 1043 of LNCS. Springer 1996, 238266.

[63] Hardin, R., Kurshan, R., Shukla, S., Vardi, M.: A new heuristic for bad cycle detection using BDDs. In: Computer Aided Verification (CAV'97). Volume 1254 of LNCS., Springer (1997) 268-278.

[64] Esparza, J., Heljanko, K.: Implementing LTL model checking with net unfoldings. In: SPIN 2001. Volume 2057 of LNCS., Springer (2001) 3756.

[65] Heljanko, K.: Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets. PhD thesis, Helsinki University of Technology, Department of Computer Science and Engineering 2002.

[66] Kupferman, O., Vardi, M.: Model checking of safety properties. Formal Methods in System Design 19, 2001, 291-314.

[67] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, Kazuhiko Kato. BitVisor: A Thin Hypervisor for Enforcing i/o Device Security. VEE 2009: 121-130

[68] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen, Revirt: Enabling intrusion analysis through virtual machine logging and replay, Proceedings of 2002 Symposium on Operating Systems Design and Implementation (OSDI'02), December 2002, pp.211-224

[69] S. Berger, R. Caceres, et al. vTPM: Virtualizing the Trusted Platform Module, Proceedings of the USENIX Annual Technical Conference (USENIX'06). USENIX., May 30 - June 3, 2006, Boston, MA, USA, pp. 21-21

[70] K. Onoue, Y. Oyama, and A. Yonezawa. Control of system calls from outside of virtual machines. In R. L. Wainwright and H. Haddad, editors, *SAC*, pages 2116–1221. ACM, 2008.

[71] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Technical report, NSA, 2001.

[72] Foley, S.: Aggregation and separation as noninterference properties. Journal of Computer Security 1(2)(1992) 159-188

[73] Sandhu, R.: Lattice based access control models. IEEE Computer 26(11) (1993) 9-19

[74] Lee, T.: Using mandatory integrity to enforce 'commerical' security. In: Proceedings of the Symposium on Security and Privacy. (1988) 140-146

[75] Foley, S.: The specification and implementation of commercial security requirements including dynamic segregation of duties. In: ACM Conference on Computer and Communications Security. (1997) 125-134

[76] Sandhu, R.: Role hierarchies and constraints for lattice-based access controls. In: ESORICS. (1996)

[77] G. Brandman. Patching the interprise. ACM Queue, Mar. 2005.

[78] A. Bellissimo, J. Burgess, K. Fu. Secure Software Updates: Disappointments and New Challenges In USENIX Hot Topics Security Workshop (Hot-Sec), July 2006, Vancouver, Canada.

[79] J. Briffaut. *Formalisation et garantie de propriétés de sécurité système: Application à la détection d'intrusions.* PhD thesis, LIFO Université d'Orléans, ENSI Bourges, Dec. 2007.

[80] A. Brown and M. Ryan. Synthesising monitors from high-level policies for the safe execution of untrusted software. In *Information Security Practice and Experience*, pages 233–247. Springer Verlag LNCS 4991, 2008.

[81] H. Dias. Linux kernel 'net/atm/proc.c' local denial of service vulnerability. BugTraq Id 32676, CVE-2008-5079, Dec. 2008.

[82] Linux Security Modules 2012. `http://kernel.org/doc/htmldocs/lsm.html`

[83] Monirul I. Sharif, Wenke Lee, Weidong Cui, Andrea Lanzi. Secure in-VM Monitoring Using Hardware Virtualization. ACM Conference on Computer and Communications Security 2009: 477-487.

[84] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.

[85] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium*, San Diego, CA, Feb. 2003.

[86] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the 6th USENIX Security Symposium*, San Jose, CA, July 1996.

[87] J. Goubault-Larrecq and J. Olivain. A smell of Orchids. In M. Leucker, editor, *Proceedings of the 8th Workshop on Runtime Verification (RV'08)*, Lecture Notes in Computer Science, pages 1–20, Budapest, Hungary, Mar. 2008. Springer.

[88] B. Morin and H. Debar. Correlation of intrusion symptoms: an application of chronicles. In *Proceedings of the 6th International Conference on Recent Advances in Intrusion Detection (RAID'03)*, pages 94–112, 2003.

[89] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. *SIGOPS Operating Systems Review*, 30:229–243, Oct. 1996.

[90] Nettop, 2004. `http://www.nsa.gov/research/tech_transfer/fact_sheets/nettop.shtml`.

[91] J. Olivain and J. Goubault-Larrecq. The Orchids intrusion detection tool. In K. Etessami and S. Rajamani, editors, *17th Intl. Conf. Computer Aided Verification (CAV'05)*, pages 286–290. Springer LNCS 3576, 2005.

[92] K. Onoue, Y. Oyama, and A. Yonezawa. Control of system calls from outside of virtual machines. In R. L. Wainwright and H. Haddad, editors, *SAC*, pages 2116–1221. ACM, 2008.

[93] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, Aug. 2003.

[94] W. Purczyński and qaaz. Linux kernel prior to 2.6.24.2 '`vmsplice_to_pipe()`' local privilege escalation vulnerability. http://www.securityfocus.com/bid/27801, Feb. 2008.

[95] Qemu, 2010. `http://www.qemu.org/`.

[96] Small number of video iPods shipped with Windows virus. `http://www.apple.com/support/windowsvirus/`, 2010.

[97] M. Roger and J. Goubault-Larrecq. Log auditing through model checking. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 220–236. IEEE Comp. Soc. Press, 2001.

[98] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. Doorn. Building a MAC-based security architecture for the Xen opensource hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference*, Tucson, AZ, Dec. 2005.

[99] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[100] Amir Pnueli, The Temporal Logic of Programs. Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS),1977, 46-57.

[101] Gerard J. Holzmann: The Model Checker SPIN. IEEE Trans. Software Eng. 23(5):1997, 279-295.

[102] Paul Gastin, Denis Oddoux: LTL with Past and Two-Way Very-Weak Alternating Automata. MFCS 2003: 439-448

[103] R. Sekar, C. Ramakrishnan, I. Ramakrishnan, and S. Smolka. Model-carrying code (MCC): A new paradigm for mobile-code security. In *Proceedings of the New Security Paradigms Workshop (NSPW 2001)*, Cloudcroft, NM, Sept. 2001. ACM Press.

[104] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *SSYM 1999: Proceedings of the 8th conference on USENIX Security Symposium*, Berkeley, CA, 1999.

[105] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker and S. A. Haghighat, A Domain and Type Enforcement UNIX Prototype, In Proceedings of the 5th USENIX UNIX Security Symposium, June 1995.

[106] Adrian Baldwin, Chris Dalton, Simon Shiu, Krzysztof Kostienko, Qasim Rajpoot Providing Secure Services for a Virtual Infrastructure ACM SIGOPS Operating Systems Review archive Volume 43 Issue 1, January 2009 ACM New York, USA.

[107] Trent Jaeger, Reiner Sailer, Yogesh Sreenivasan. Managing the Risk of Covert Information Flows in Virtual Machine Systems. In Proceedings of ACM Symposium on Access Control Models and Technologies (SACMAT), 2007.

[108] Bernhard Jansen, HariGovind V. Ramasamy, Matthias Schunter. Policy Enforcement and Compliance Proofs for Xen Virtual Machines. In proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments.

[109] Biba, K. J. Integrity Considerations for Secure Computer Systems, MTR-3153, The Mitre Corporation, April 1977.

[110] Introducing Role-based Access Control to a Secure Virtual Machine Monitor: Security Policy Enforcement Mechanism for Distributed Computers. In : IEEE Asia-Pacific Services Computing Conference 2008.

[111] Till Mossakowski, Michael Drouineaud, Karsten Sohr. A temporal-logic extension of role-based access control covering dynamic separation of duties. In TIME-ICTL 2003. IEEE Computer Society.

[112] Bell, D.E., Padula, L.J.L.: Secure computer system: unified exposition and MULTICS interpretation. Report ESD-TR-75-306, The MITRE Corporation (1976)

[113] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Technical report, NSA, 2001.

[114] P. Starzetz. Linux kernel 2.4.22 `do_brk()` privilege escalation vulnerability. `http://www.k-otik.net/bugtraq/12.02.kernel.2422.php`, Dec. 2003. K-Otik ID 0446, CVE CAN-2003-0961.

[115] Virtualbox, 2010. `http://www.virtualbox.org/`.

[116] Vmware, 2010. `http://www.vmware.com/`.

[117] R. Wojtczuk. Subverting the Xen hypervisor. In *Black Hat'08*, Las Vegas, NV, 2008.

[118] [ms-wusp]: Windows update services: Client-server protocol specification. `http://msdn.microsoft.com/en-us/library/cc251937(PROT.13).aspx`, 2007–2010.

[119] Xen, 2005–2010. `http://www.xen.org/`.

[120] J. Zimmerman, L. Mé, and C. Bidan. Introducing reference flow control for detecting intrusion symptoms at the OS level. In *Proceeedings of the Recent Advances in Intrusion Detection Conference (RAID)*, pages 292–306, 2002.

[121] J. Zimmerman, L. Mé, and C. Bidan. Experimenting with a policy-based hids based on an information flow control model. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, page 364, Washington, DC, USA, 2003. IEEE Computer Society.

[122] J. Zimmerman, L. Mé, and C. Bidan. An improved reference flow control model for policy-based intrusion detection. In *Proceeedings of the European Symposium On Research in Computer Security (ESORICS)*, pages 291–308, 2003.

[123] FUSE, 2012. http://fuse.sourceforge.net/.

[124] CVE-2008-5079, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5079.

[125] CVE-2005-3857, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-3857.