End-to-end security architecture for cloud computing environments

Aurelien Wailly

École Doctorale Informatique, Télécommunications et Électronique de Paris

THÈSE DE DOCTORAT CONJOINT TÉLECOM SUDPARIS et L'UNIVERSITÉ PIERRE ET MARIE CURIE

n.º 2014TELE0020

préparée à **SAMOVAR** et à **Orange Labs**

Présentée par

Aurélien WAILLY

pour obtenir le grade de

Docteur de l'Université Pierre et Marie Curie
Spécialité : Informatique

Architecture de sécurité de bout en bout et mécanismes d'autoprotection pour les environnements Cloud

Soutenue publiquement le 30 septembre 2014 devant le jury composé de :

M. Hervé Debar	Directeur de thése	Professeur, Télécom SudParis
M. Marc Lacoste	Encadrant de thése	Expert Recherche, Orange Labs
M. Jean-Marc Menaud	Rapporteur	Professeur, École des Mines de Nantes
M. Éric Totel	Rapporteur	Professeur, Supélec
M. André-Luc Beylot	Examinateur	Professeur, INPT/ENSEEIHT
M. Thierry Coupaye	Examinateur	Directeur de Recherche, Orange Labs
M. Frédéric Cuppens	Examinateur	Professeur, Télécom Bretagne
M. Jean Leneutre	Examinateur	Maître de conférence, Télécom ParisTech
M. Pierre Sens	Examinateur	Professeur, Université Pierre et Marie Curie

Dedicated to my lovely wife.
Nihale

ABSTRACT

Since several years the virtualization of infrastructures became one of the major research challenges. These new virtual machines consume less energy while delivering new services: live migration, consolidation and isolation between tenants. However, many attacks hinder the global adoption of Cloud computing.

VESPA

Self-protection has recently raised growing interest as possible element of answer to the cloud computing infrastructure protection challenge. Faced with multiple threats and heterogeneous defense mechanisms, the autonomic approach proposes simpler, stronger, and more efficient cloud security management. Yet, previous solutions fall at the last hurdle as they overlook key features of the cloud, by lack of flexible security policies, cross-layered defense, multiple control granularities, and open security architectures.

This thesis presents VESPA, a self-protection architecture for cloud infrastructures overcoming such limitations. VESPA is policy-based, and enforces security at two levels, both within and across infrastructure layers. Flexible coordination between self-protection loops allows enforcing a rich spectrum of security strategies such as cross-layer detection and reaction. A multi-plane extensible architecture also enables simple integration of commodity detection and reaction components. Evaluation of a VESPA implementation shows that the design is applicable for effective and flexible self-protection of cloud infrastructures.

KungFuVisor

Recently, some of the most powerful attacks against cloud computing infrastructures target their very foundation: the hypervisor or Virtual Machine Monitor (VMM). In many cases, the main attack vector is a poorly confined device driver in the virtualization layer, enabling to bypass resource isolation and take complete infrastructure control. Current architectures offer no protection against such attacks. At best, they attempt to contain but do not eradicate the detected threat, usually with static, hard-to-manage defense strategies. This thesis proposes an altogether different approach by presenting KungFuVisor, derived from VESPA. It is a framework to build self-defending hypervisors. The framework regulates hypervisor protection through several coordinated autonomic security loops which supervise different VMM layers through well-defined hooks. Thus, interactions between a device driver and its VMM environment may be strictly monitored and controlled automatically. The result is a very flexible self-protection architecture, enabling to enforce dynamically a rich spectrum of remediation actions over different parts of the VMM, also facilitating defense strategy administration.

Conclusion

VESPA is a generic, flexible and open architecture enhancing virtualized systems security. We showed the application to three different protection scheme: virus infection, mobile clouds and hypervisor drivers. Through the analysis and the evaluation of the architecture, we showed that cloud infrastructure security can be enhanced.

RÉSUMÉ

Depuis plusieurs années la virtualisation des infrastructures est devenue un des enjeux majeurs dans la recherche. Ces nouveaux types de machines fournissent des consommations d'énergie moindres, ainsi que de nouvelles opportunités. La migration à chaud, la consolidation et l'isolation sont les principaux atouts de la virtualisation. Cependant plusieurs attaques les ont mis à mal et freinent l'adoption de ce nouveau paradigme.

VESPA : Architecture de sécurité pour les environnements virtualisés

L'auto protection est récemment devenue un des centres d'intérêt permettant de répondre aux enjeux de protection des infrastructures cloud computing. Face à de multiples menaces et des mécanismes de défense hétérogènes, l'approche autonomique propose une gestion simplifiée, robuste et plus efficace de la sécurité du cloud. Aujourd'hui, les solutions existantes ne s'adaptent pas complètement car elles ne prennent pas en compte les fonctionnalités clefs du cloud. Le manque de politiques de sécurité flexibles, une défense multi-niveaux, des contrôles à granularité variable, ou encore une architecture de sécurité ouverte.

Cette thèse présente VESPA, une architecture d'autoprotection pour les infrastructures cloud répondant aux limitations précédentes. VESPA est construit autour de politiques, et peut réguler la sécurité à deux niveaux, directement au sein d'une couche ou bien à l'aide d'une approche inter couche. La coordination flexible entre les boucles d'autoprotection permet de mettre en place un large spectre de stratégies de sécurité comme des détections et des réactions impliquant plusieurs niveaux. Une architecture extensible multi plans permet d'intégrer simplement des éléments de sécurité déjà présents. L'évaluation de plusieurs implémentations de VESPA montre que cette architecture délivre une autoprotection efficace et flexible des infrastructures cloud.

KungFuVisor : Protection autonomique des hyperviseurs

Depuis peu, les attaques les plus critiques contre les infrastructures cloud visent les briques les plus sensibles: l'hyperviseur ou moniteur de machine virtuelle (VMM). A chaque fois, le vecteur d'attaque principal est un pilote de périphérique mal confiné dans la couche de virtualisation. Cela permet d'outrepasser l'isolation des ressources et de prendre le contrôle complet de l'infrastructure. Les architectures actuelles n'offrent que peu de protections face à ces attaques. Au mieux, les hyperviseurs tentent de confiner la menace détectée mais sans l'éradiquer. Les mécanismes de défense mis en jeu sont statiques et difficile à gérer.

Cette thèse propose une approche différente en présentant KungFuVisor, un canevas logiciel pour créer des hyperviseurs autoprotégés spécialisant l'architecture VESPA. Le canevas logiciel régule la protection de l'hyperviseur au travers de plusieurs boucles d'auto-protection synchronisées. Ces boucles autonomiques supervisent plusieurs couches de l'hyperviseur grâce à plusieurs mécanismes interception. Les interactions entre les drivers et le VMM peuvent alors être finement surveillées et contrôlées de façon automatique. Le résultat est une architecture d'autoprotection flexible permettant de mettre en place une large panoplie de réactions. Le contrôle des différentes parties du VMM permet d'administrer des stratégies de défense simplement.

Conclusion

VESPA est une architecture générique, flexible et ouverte permettant d'améliorer la sécurité des systèmes basés sur la virtualisation. Nous avons montré son application à trois types de protection différents : les attaques virales, la gestion hétérogène multi-domaines et l'hyperviseur. À travers l'étude et la validation des architectures proposées, nous avons montré que la sécurité des infrastructures cloud peut être améliorée grâce à VESPA.

REMERCIEMENTS

J'aimerais tout d'abord remercier mon directeur de thèse, Hervé Debar, pour son aide et son soutien inconditionnel tout au long de cette thèse. Les conseils, cours et TP qu'il a su me donner ont été les piliers techniques de ce manuscrit.

Je remercie mon encadrant de thèse à Orange, Marc Lacoste, qui a fait preuve d'une patience infinie et a su me rendre autonome. Sa rigueur implacable m'a permis d'atteindre une curiosité scientifique que je n'aurais jamais espérée.

Je tiens à remercier les membres du jury. André-Luc Beylot pour ses nombreuses années d'appui, son partage unique des connaissances et sans qui ce document n'aurait jamais vu le jour. Jean-Marc Menaud et Jean Leneutre pour leur suivi tout au long de ce parcours, autour de projets collaboratifs et des différents cours qu'ils m'ont confiés. Et finalement Eric Totel, Pierre Sens, Frédéric Cuppens et Thierry Coupaye pour leur relecture minutieuse de ce document et pour leurs commentaires utiles.

Je remercie également Julie Marlot pour l'accueil chaleureux au sein d'Orange, d'une équipe généreuse et dans des conditions de travail uniques. Je félicite mes deux co-bureaux Loïc et Xiao pour avoir tenu aussi longtemps. Merci à Emmanuel, Benoît, Ludovic, Mathieu, Isabelle, Aymeric, Gaëtan, Aymerick, Jean-Michel, Pascal, Nicolas, Georgian, Christophe, Pierre, Benjamin, Sébastien, Nizar, Sok-Yen, Daniel, Sabri, Laura, Gilles, Matt, Tiphaine, Xavier, Vincent, Olivier, Alex, Kévin, Ryad, Sylvie, Kahina, Jean-François, Jean-Philippe, Adam, Yannick, Saïd et Amira.

Je remercie l'équipe du laboratoire SAMOVAR pour son support et ses conseils. Merci à Yosra, Gustavo, Nabil, Frédéric, François-Xavier, Maryline et Françoise.

Je n'oublierai pas les nuits blanches de détente avec les acharnés des CTF, où les montées de dopamine correspondaient aux nouvelles compétences que j'ai pu développer. Merci à Alexis, André, Ivan, Jonathan, Alex, Eloi, Sofian, Clément, Florent et Jean-Baptiste.

Mes remerciements vont aussi à mes parents Frédéric et Anne-Marie, ma sœur Jessica, ainsi qu'à toute ma famille. Aux amis qui ont espéré ce jour au travers de l'éternelle question "as-tu soutenu ta thèse ?", j'aurais dû leur répondre une belle citation "du bon usage de la lenteur". Merci à Manu, Romain, David, Alban, Teddy, Etienne, Philippe, Eléonore, Marion et Noémie.

Enfin, je remercie infiniment ma douce Nihale pour sa vision unique et ses longues relectures. Nous avons construit ces bébés ensemble.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ACL	Access Control List
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ARP	Address Resolution Protocol
CGA	Cryptographically Generated Addresses
CPU	Central Processing Unit
CU	Control Unit
CVE	Common Vulnerabilities and Exposures
DAC	Discretionary Access Control
DMA	Direct Memory Access
DoS	Denial of Service
DTE	Domain-Type Enforcement
EPT	Extended Page Table
Flask	Flux Advanced Security Kernel
HSV	Hypervisor-secure virtualization
IaaS	Infrastructure-as-a-Service
IDS	Intrusion Detection System
IDPS	Intrusion Detection and Prevention System
IDT	Interrupt Descriptor Table
IO	Input/Output

IOMMU	Input/Output Memory Management Unit
IPC	Inter-Process Communication
IPS	Intrusion Prevention System
ISA	Instruction Set Architecture
LoC	Line Of Codes
LSM	Linux Security Module
MITM	Man-In-The-Middle
MMU	Memory Management Unit
MMIO	Memory Mapped I/O
MVC	Model View Controller
NAC	Network Access Control
NIDS	Network IDS
NIPS	Network IPS
NVD	National Vulnerability Database
OrBAC	Organization-Based Access Control
OS	Operating System
PCI	Peripheral Component Interconnect
PET	Privacy-Enhancing Technology
PID	Process IDentifier
PKI	Public-Key Infrastructure
RBAC	Role-Based Access Control
RPC	Remote Procedure Call
ROP	Return-Oriented Programming
RTC	Real Time Clock
RTT	Round Trip Time
RVI	Rapid Virtualization Indexing
SLA	Service-Level Agreement
SMP	Symmetric MultiProcessing
SVM	Secure Virtual Machine
TEE	Trusted Execution Environment
TCB	Trusted Computing Base

TLB	Translation Look-aside Buffer
TPM	Trusted Platform Module
UML	Unified Modeling Language
USB	Universal Serial Bus
VESPA	Virtual Environment Self-Protection Architecture
VLAN	Virtual Local Area Network
VM	Virtual Machine
VMM	Virtual Machine Monitor
VMX	Virtual-Machine Extensions
VNIC	Network Interface Controller
VO	Vertical Orchestrator
VPN	Virtual Private Network
XSM	Xen Security Module

1 INTRODUCTION AND BACKGROUND

1.1 NEW CHALLENGES FOR DISTRIBUTED SYSTEMS

Computers have evolved from complex and massive mainframes to light and handy workstations. Thus, we are experiencing new ways to operate and use machines. Physical server virtualization enables on-demand allocation of memory, computer or disk space to meet needs overtime. New services lighten IT management, migrating virtual machines between countries to concentrate workloads and cutting costs. This is the **Cloud computing** era.

This disruptive distributed computing model for large-scale networks contracts out corporate IT to third parties. This shared pool of computing, storage, networking and services become accessible rapidly and on demand. Fore casted benefits include flexible and dynamic provisioning, simpler and automated administration of data centers, and sharing of nearly unlimited CPU, bandwidth, or disk space.

Unfortunately, security is viewed as one of the main adoption stoppers to cloud computing. The complexity of infrastructures leaves the door open to various threats coming from the outside and from the inside [3]. Intrusions, malware or security policy-violations of curious or malicious users are just but a few. This is particularly true at the foundation: the infrastructure-level cloud model, also known as Infrastructure-as-a-Service (IaaS).

If traditional security techniques such as encryption remain relevant for cloud infrastructures, those new threats need specific protection. However, few solutions are available to tackle those challenges. The tools are **heterogeneous and fragmented**, with lack of an overall vision to compose them into an integrated security architecture for cloud environments.

However, several problematics define the areas for further research. The threats depend on the cloud service delivery and deployment models. Also, the defenses activation needs short response times and the manual security maintenance is impossible. Thus, a **flexible, dynamic, and automated security management** of cloud data centers is clearly lacking today. This thesis provides elements of answer to those unsolved issues.

1.1.1 *Information security principles*

This manuscript focuses on computer security applied to large-scale distributed systems. Thus, we define security building blocks and how to fulfil them, with an application to cloud environments.

Ensuring security means prevention of illicit access and alteration of the information while delivering legitimate access and modification of the information [146]. Illicit modifications are the results of security properties bypass. The security properties define who have access to system information, how to access them and what operations are allowed. These security properties are part of the **security policies**.

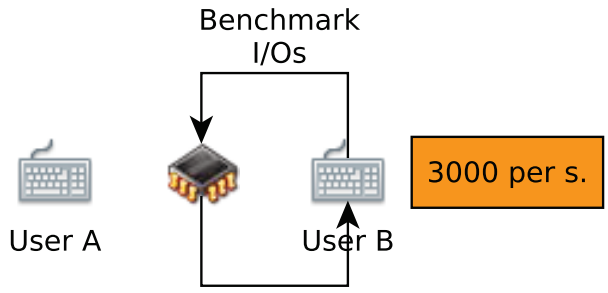


Figure 1: Compromising confidentiality: benchmarking

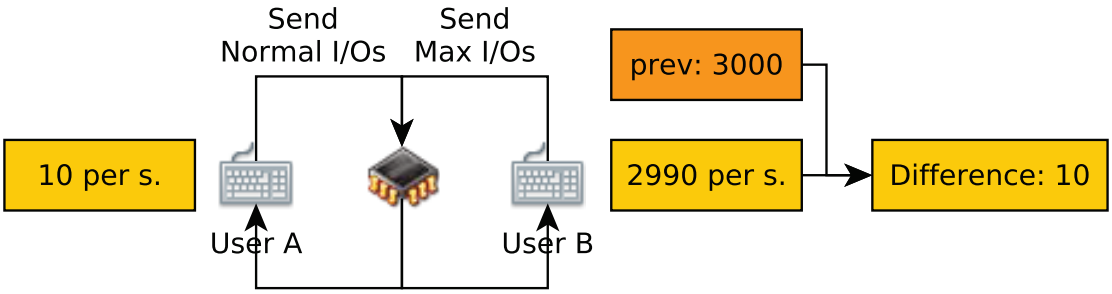


Figure 2: Compromising confidentiality: extracting data

1.1.1.1 *Confidentiality*

Confidentiality is the system capacity to prevent information disclosure, in other words to make information unreachable to users not allowed to access it. This definition covers not only information as data and programs, but also their existence and flows. Hence, all possible information paths have to be analyzed and secured. This work is exponential with the number of elements, not to say impossible with the diversity of cloud infrastructures.

Attacks against confidentiality aims at recovering information despite security policies. The example of covert channels [236] through greedy behavior, and differential analysis between processor response times allows an attacker to extract other tenant private certificates.

Let user A being mainly idle, and user B consuming all resources. User B benchmarks and profile interrupts (Figure 1) while he is the only CPU user. When user A sends an interrupt, user B detects a different access time as the processor cannot handle more interrupts (Figure 2). User B builds the activity of user A and breaks confidentiality. Section 2.2.3 details these attacks.

1.1.1.2 *Integrity*

Integrity is the system ability to prevent information corruption. Intentional corruption aims at benefiting from the service through information rewriting and erasure. Thus, an attacker may intercept traffic and perform a man-in-the-middle attack [18]. The traffic is filtered, modified and sent to the user with only small changes to be stealth and mimic original behavior.

Cloud infrastructures introduce new vectors for attackers to tamper with information. The example of hypervisor compromise [70] by isolation breakout shows that an attacker can threaten its integrity.

Unintentional corruption also appears on hardware [61] and corrupt information.

1.1.1.3 *Availability*

Availability is the system ability to warrant operational service. Attacks against the availability are closely related to integrity, where Denial of Service (DoS) is the destruction of information.

Attacks against availability in the context of virtualization are greedy behaviors, such as migrating a machine back and forth infinitely [190]. The virtualization layer wastes resources, and the service is down most of the time. Accidents occur through inattentive administrators, which shutdown the cooling system while working on physical servers, or unplugging a cable and disabling server access.

1.1.2 *Our approach*

Although most of the roadblocks defined in Appendix A are critical, in this thesis, we tackle the problem of IaaS resource isolation, both from the computing and networking perspectives. Thus we address the *hypervisor security*, *network isolation*, and *elastic security* roadblocks in the first two areas of the Appendix. We also address the *openness* and *end-to-end security* roadblocks, by defining building blocks for a reference cloud security framework. The following paragraphs give an overview of each roadblock.

The hypervisor is the building block of the system virtualization. Thus hypervisor security only addresses one layer, but involve multiple organizations while distributed. The virtualization ecosystem is under development and involves easy-to-deploy components, but is very specific to the hypervisor architecture.

Multiple virtualization layers can perform network isolation in different ways. VM layer is able to isolate programs network flows with firewalls. Hypervisor layer may restrict virtual interfaces to Virtual Local Area Network (VLAN) or specific switches, and physical equipments perform network isolation through legacy mechanisms. Also multiple organizations share out the same isolation rules to adopt a global behavior. Vendors and developers opt in delivering seamless solution easily deployable.

Elastic security involves resources allocation at different level and over multiple organization, being able to integrate new security resources dynamically.

End-to-end security requires confidentiality, integrity and availability at every layer crossed, regardless of the number or organization.

1.1.3 *Security properties*

IaaS infrastructure has to integrate three main features of cloud environments to address the problems selected in our approach (see the mapping between roadblocks and security properties in Table 1):

Multi-layering. A Cloud infrastructure is composed of many independent layers with their specific security mechanisms, while attacks may involve multiple layers. Also, reactions effectiveness against attacks can be more relevant. A specific layer can get a better global picture with information coming from other layers.

Multi-laterality. A Cloud may involve multiple organizations, with their own objectives, calling for flexible policies. Such policies require high-level expressiveness to abstract relations with specific equipment.

Roadblocks \ Properties	Multi-Layering	Multi-Laterality	Extensibility
Hypervisor security	✗	✓	✗
Network isolation	✓	✓	✓
Elastic security	✓	✓	✓
End-to-end security	✓	✗	✓

Table 1: Mapping Roadblocks to Properties

Extensibility. Clouds are increasingly evolving towards interoperability with other clouds. Thus, closed-world vision of security is inadequate. The integration of new security components have to be easy, straightforward and transparent.

1.2 RESEARCH OBJECTIVES

The objectives of this thesis are twofold: (1) propose and implement an end-to-end security architectural blueprint for cloud environments providing an integrated view of protection mechanisms; and (2) define within that architecture mechanisms enabling self-protection of the cloud infrastructure.

1.2.1 *A reference security architecture for cloud environments*

This thesis explores component-based designs to orchestrate and dynamically compose different security building blocks like hypervisors, hardware security elements (e.g., Trusted Platform Modules (TPMs)), network protections (firewalls, Intrusion Detection System (IDS)/Intrusion Prevention System (IPS), Virtual Private Networks (VPNs)), and secure storage, privacy-enhancing, or trust management mechanisms. Each component declares its guaranteed security properties using contracts, which are composed to derive the overall cloud security objectives. This end-to-end security architecture is validated through the realization of a prototype of secure cloud and several use case.

To reach the first objective, Bruneton et al. [41] demonstrated the viability of component-based design to build complex systems from heterogeneous building blocks and reach flexible security. We explore that approach to orchestrate and adapt security services in a cloud (e.g., as Web Services) to compose individual security services flexibly inside a unified security architecture. Security properties provided by individual security services are expressed as flexible contracts, e.g., Service-Level Agreements (SLAs), to derive overall security objectives guaranteed by the cloud infrastructure.

1.2.2 *Self-protected clouds*

Specify and implement self-protection mechanisms within the cloud is the second objective. We identify the components needed to realize one or more self-protection loops to make cloud security self-managed. This thesis also defines self-protection architecture. The identified security components are then implemented and integrated into the secure cloud prototype.

Design Principles \ Properties	Multi-Layering	Multi-Laterality	Extensibility
P1	✓	✓	✓
P2	✓	✗	✗
P3	✓	✓	✓
P4	✗	✗	✓

Table 2: Mapping Properties to Design Principles

To reach the second objective, IBM's autonomic computing approach [47] for self-managed security also proved its interest to build security infrastructures with minimal security administration overheads. It satisfies multiple security requirements, and react rapidly to detected threats: security parameters are autonomously negotiated with the environment to match the ambient estimated risks and achieve an optimal level of protection. A first generic component-based framework for self-protection has been defined [93]. The first part of this dissertation work study whether this framework is sufficient for self-management of cloud security, and define the necessary extensions for that purpose.

1.3 DESIGN PRINCIPLES

With the security properties and the objectives, we expose a set of design principles, referred as P_X or *principle*, for a self-protection cloud architecture (see Table 2):

[P1] Policy-based self-protection. The policy-driven paradigm has successfully demonstrated its power and generality to increase adaptability in self-management [193]. Therefore, a richer choice of security strategies can be supported in each phase of the control loop [111]. For instance, this thesis defines and enforces a wider range of detection and reaction policies. Model-based designs also generally result in cleaner, more extensible software architectures. More generally, policy automation contributes to a more unified management of security, with consistent enforcement across security silos. P1 thus addresses heterogeneity, multi-layering, and multi-laterality.

[P2] Cross-layer defense. Events detected in one layer trigger reactions in other layers. Conversely, a reaction may be launched based on events aggregated from several layers. This coordinated approach helps to bridge the semantic gap between layers who often are not aware of one another, each with their own monitoring mechanisms and representation for the monitored information of different levels of abstraction. The cross-layer approach also improves security by helping to capture the overall extent of an attack, often not limited to a single layer, and to better respond to it. P2 thus addresses notably the multi-layering challenge.

[P3] Multiple self-protection loops. A simple decision can generate several reactions, extending monitoring granularity. Multiple stages of decision are also possible. Different scope levels for security supervision may thus be defined and coordinated. Several degrees of optimality are chosen for the response: either local and fast, but of variable accuracy, or broader, with a higher-level of relevance due to more knowledge available, but slower [140]. Trade-offs with other properties

than security are also possible. P3 addresses the multi-laterality challenge by considering self-protection over multi-decisional and reaction paths.

[P4] Open architecture. The security architecture have to integrate easily and compose flexibly several types of detection and reaction strategies and mechanisms. The architecture should notably allow integrating simply heterogeneous off-the-shelf security components, to support the widest array possible of defense mechanisms to mitigate both known and unknown threats. This principle also enables to achieve better interoperability between security components across clouds. P4 essentially addresses the openness and heterogeneity challenges, along with the necessity to defend the system against multiple threats.

1.3.1 *Research environment*

Substantial prior work has attempted to build systems fulfilling one or several of those principles, in terms of policy management frameworks, self-protecting distributed systems, protection mechanisms for virtual machines and for the virtualization layer, or traditional Intrusion Detection and Prevention Systems (IDPSs) and anti-malware tools. Yet those solutions always seem to fall short of addressing one principle or more. For instance, despite quite extensive models, generic policy frameworks usually little address security or cloud environments. Most of the time, policies are not granular, and does not address multi-layered defense. Existing cloud protection mechanisms are either detection- or reaction-oriented, but rarely both. Similarly, they tackle cross-layering or have flexible policies, but not both. Integration with outside systems also remains difficult. Legacy IDPSs are more open, but often without a well-formalized adaptation model. Overall, the multiple self-protection loops addressed in P3 are almost always ignored.

1.3.2 *Our contributions*

To overcome those limits, this thesis presents an architecture and framework based on the previous design principles to build a self-defending IaaS infrastructures. Our solution called Virtual Environment Self-Protection Architecture (VESPA) regulates protection of IaaS resources through several coordinated autonomic computing security loops which monitor the different infrastructure layers. The result is a flexible approach for self-protection of IaaS resources.

This thesis details how we built:

VESPA a self-protection architecture for cloud infrastructures overcoming previous limitations. VESPA is policy-based, and regulates security at two levels, both within and across infrastructure layers. Flexible coordination between self-protection loops allows enforcing a rich spectrum of security strategies such as cross-layer detection and reaction. A multi-plane extensible architecture also enables simple integration of commodity detection and reaction components. Evaluation of a VESPA implementation shows that the design is applicable for effective and flexible self-protection of cloud infrastructures. However, some legacy components do not offer enough functionalities to build a secure cloud. Even the core virtualization component does not provide remediation facilities, that leads to our next contribution.

KungFuVisor a framework to build self-defending hypervisors using VESPA. Commodity hypervisors are able to detect attacks against their integrity, but lack the ability to repair themself. To answer this issue, the framework regulates hypervisor protection through several coordinated autonomic security loops which supervise different Virtual Machine Monitor (VMM) layers through well-defined hooks. Thus, interactions between a device driver and its VMM environment may be strictly monitored and controlled automatically. The result is a very flexible self-protection architecture, enabling to enforce dynamically a rich spectrum of remediation actions over different parts of the VMM, also facilitating defense strategy administration.

1.4 ORGANIZATION OF THE THESIS

This thesis is structured as follows:

Chapter 2 reviews **cloud computing security state-of-the-art** to compare academic and industrial solutions, to extract key points to match the thesis objectives. We describe Cloud Computing and the fundamentals of such architectures. Then, the thesis analyzes and compares the different hypervisors - that are the essence of virtualization - and cloud components. This chapter evaluate and classify their security. This chapter also gives an outline of Application Programming Interfaces (APIs) and protocols used by previous components to communicate, and their associated tools.

Chapter 3 describes the **VESPA autonomic framework model** that is the core of this thesis. This chapter describes the fundamental entities of the model, the policies and how they interact. This thesis analyzes then the threats to such architectures and give alternatives to prevent them. Finally this chapter details the prototype with a first implementation and its evaluation.

Chapter 4 analyzes framework object composition through **three use cases**. The first scenario describes the dynamic confinement of a virtual machine during a typical viral infection. The second scenario explains how to deploy policies over multiple IaaS stacks and the MobiCloud using VESPA. Finally, we adapt the solution to perform fuzzing and thus benchmark the performance of the VESPA framework.

Chapter 5 explores an **extension of the self-protecting architecture** to specific part of cloud components, **the hypervisor**. This chapter maps the VESPA architecture to hypervisor internals, to reuse previous work and highlight the potential of the framework. The new architecture brings self-protection to hypervisors, named **KungFuVisor**. We also depict attacks that threaten hypervisors, and classify them to analyze particular weak points. Then, this chapter concludes by describing and evaluating the implementation of the extension.

Chapter 6 concludes this thesis, with a final analysis of the objectives. The chapter details the limits of this study, and the future work describes some perspectives to extend this work.

2 CLOUD COMPUTING SECURITY: STATE-OF-THE-ART

Mais ne lisez pas, comme les enfants lisent, pour vous amuser, ni comme les ambitieux lisent, pour vous instruire. Non. Lisez pour vivre.

This chapter details the cloud computing concepts and the different approaches to enhance security. Section 2.1 presents the cloud computing paradigm and the different components. Section 2.2 details the core components of cloud infrastructures, the hypervisors. Section 2.3 discusses different approaches to secure the cloud and their limits. Finally, Section 2.4 summarizes remarks and analyzes.

2.1 CLOUD COMPUTING FOUNDATIONS

This section introduces fundamental IT definitions (Section 2.1.1) and abstractions (Section 2.1.2). Then this section describes cloud computing background (Section 2.1.3), and the variety of components to provide strong isolation (Section 2.1.4).

2.1.1 *Definitions*

Before entering cloud computing model and virtualization, we introduce essential definitions to describe a system. Those definitions are based on [124, 204, 75].

RESOURCE A program use software or hardware objects named resources. A resource can be a device or a logical controller, such as a hard drive, a memory or information.

INTERFACE Access primitives that query and modify resources, and respect sets of rules representing the interface definition. This is a language for users to interact with the resource. The set of rules defines limits related to the use of the resource.

PROGRAM The set of instructions changing machine state compose a program, and the associated activity is a task. In concrete terms, a program is a suit of instructions calling resources interfaces of the system.

2.1.2 *Resource abstraction*

The operating system (OS) abstracts a physical computer and its resources. The OS builds an overlay to separate the logical view from the hardware. This abstraction hides resources heterogeneity and provides a new view with unified interfaces. For example, Linux offers a file abstraction to access all system resources. A read on a file returns the data stored on the hard drive while a read on the network card receive packets.

An extension of the abstraction is virtualization. It is the simulation of a resource that does not physically exist. One or several lower level resources build this virtual resource. The virtualization is applied to low level resource as the CPU or the memory. The main goal is to provide seamless access to the physical resource with fine-grained control. We define a Virtual Machine (VM), by extension, when all physical resources are virtualized. As virtual resources, a VM does not exist but provide the same behavior as a physical machine, either on a physical machine or into an OS process.

This section details how the OS manage resources. First we review computer resources and their dependencies.

2.1.2.1 *Physical resources*

A computer contains several processors, a main memory unit, hard drives as second memory, network cards, controllers and Input/Output (IO) devices.

PROCESSOR The Central Processing Unit (CPU) fetches, decodes and executes assembly instructions of a program. It is the hearth of computing that coordinates all other components. Each constructor has its specific architecture but we find at least four components.

- The Arithmetic Logic Unit (ALU) handles arithmetic and logic integer operations. The addition is an arithmetic operation, while a comparison is a logic one.
- The Control Unit (CU) fetches, decodes and executes program instructions using the ALU if needed.
- The address and data buses link the components.
- The clock rhythms the CPU, and at every clock cycle the processor performs one or more operations.

The processor comes with a programmatic model to interact with and that is the base of commodity Operating Systems (OSs). This model usually contains four parts.

- The Instruction Set Architecture (ISA) is the list of instruction supported by the processor and associated behaviors. This is processor interface.
- The context of the processor details the state of the processor with general registers and special registers.
- Interruptions deliver synchronous and asynchronous events to the CPU. Launching an interruption suspend the actual CPU execution, trigger a specific processing and restore the execution. Contexts are saved and restored to fulfill the process seamlessly. This event-driven model enables asynchronous event delivery for devices (network cards, keyboard), and a generic interface for exceptions (divide by zero, segmentation faults). The CPU owns an Interrupt Descriptor Table (IDT) describing functions to execute on interrupt reception.
- Privileged modes. For security purposes, the processor can execute instructions in user mode with restricted instruction set or in supervisor mode with full access. User mode programs can use privileged services through system calls.

MAIN MEMORY The main memory is the internal Random Access Memory (RAM). This memory stores programs data and instructions temporarily. The CPU and the controllers interact with the RAM. The system bus interconnects the processor and the main memory. The cache memory is a small and fast memory component providing fast access to the latest data accessed. This mechanism causes coherency problems between cached information and the real one. The size varies and multiple cache layers exist providing granular memory cache. The L1 cache is extremely fast but contains hundreds lines at best, while L2 is slower and bigger, and now we see a large L3 cache to cache up to 12MB of information. The caches are varied, separating data and addresses. The Memory Management Unit (MMU) is a physical component traducing access to logical addresses toward physical address. The Translation Look-aside Buffer (TLB) table translates the addresses from the processor view to the memory view. The TLB delivers virtual memory through small units named pages. The page table describes the virtual memory, and the TLB is the cache of this table.

IO PERIPHERALS The IO peripherals enable communications with the outside, such as the user: keyboard, mouse, image scanners, graphic cards, or webcams. Each peripheral is under the control of a controller, linked through a bus (Universal Serial Bus (USB), Peripheral Component Interconnect (PCI) or system). The controller helps the processor and holds a series of commands to drive the peripheral autonomously. Then, an asynchronous interruption is triggered to inform the CPU that the action is done.

2.1.2.2 *System resources*

System resources are abstraction found on commodity OSs, from physical resources or built on abstract resources.

THREAD OF EXECUTION The CPU executes one instruction at a time, when an OS need multiple executions path at the same time. Thus, instructions and contexts are handled by a scheduler to multiplex tasks. This is a first virtualization of the CPU to provide simultaneous executions.

VIRTUAL MEMORY The virtual memory divides the logical memory used by programs from the physical memory. The MMU handles this virtualization. The logic addresses space is valid and accessible for a program from the address space. This address space is isolated and prevents modifications of colocated OS processes. The virtual memory segmentation enables memory policies to control program parts. For example, the code segment contains the instructions to execute, the data segment contains initialized data and stack segment contains the stack. Virtual addresses are represented by a segment number and an offset into this segment. The pagination provides a second abstraction to give a larger virtual memory. Thus, process can use more memory than the RAM size with a second memory unit as a backup, usually a hard drive. The last accessed data are stored into the RAM for fast access.

FILES The hard drive stores data permanently as files, which is a unified view of the second memory unit. Each file has a name, and the folder groups files.

DEVICE DRIVERS Device drivers are software in charge of managing controllers. The driver is aware of the controller interface and delivers a higher-level interface. Each controller needs a specific driver, which is why the Linux kernel code base contains 80% of drivers. However drivers deal with subtle sequences of commands, resulting in an error-prone environment. Section 5 details driver protection.

PROCESS A process abstracts a running program for user applications and system services. All resources needed to run the program are accessible in the process.

2.1.2.3 *Conclusion*

We defined the foundation of traditional IT systems, and we now introduce how these components are used in the cloud computing paradigm. Thus, the next section explains the terms related to cloud computing environments, based on the standard definition of the NIST institute.

2.1.3 *NIST definition of Cloud Computing*

Cloud computing is a novel paradigm characterizing the dynamic use of shared computing resources through virtualization. This cloud model is composed of five essential characteristics, three service models, and four deployment models [138] explained further.

2.1.3.1 *Essential characteristics*

ON-DEMAND SELF-SERVICE. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

BROAD NETWORK ACCESS. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).

RESOURCE POOLING. The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or data center). Examples of resources include storage, processing, memory, and network bandwidth.

RAPID ELASTICITY. Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

MEASURED SERVICE. Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts).

Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

2.1.3.2 *Service models*

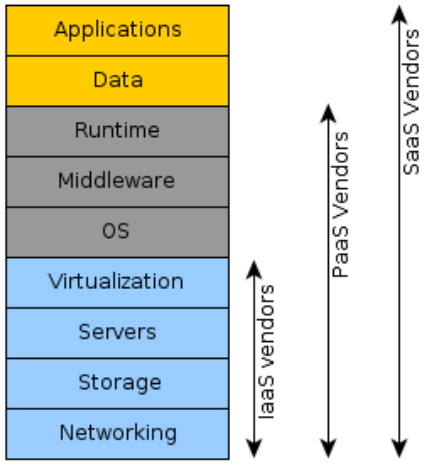


Figure 3: Cloud Service Models.

[SAAS] SOFTWARE-AS-A-SERVICE. The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

[PAAS] PLATFORM-AS-A-SERVICE. The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.

[IAAS] INFRASTRUCTURE-AS-A-SERVICE. The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

SERVICE MODEL	MAIN EXAMPLES
SaaS	*Amazon* Web Services, *Google* Apps, *HP* Cloud Services, *IBM* SmartCloud, *Microsoft* Office 365, Salesforce
PaaS	Cloud Foundry, *Heroku*, *Google* App Engine, *AppScale*, *Windows* Azure Cloud Services
IaaS	*Amazon* EC2, *Google* Compute Engine, *HP* Cloud, *Joyent*, *Rackspace*, *Windows* Azure Virtual Machines

Table 3: Notable Cloud Service Examples.

Figure 3 summarizes stacks used by vendors at the SaaS-, PaaS- and IaaS-level. Table 3 shows several examples of service providers. Our research aimed the IaaS layer, where virtualization of hardware resources is the key element.

2.1.3.3 *Deployment models*

PRIVATE CLOUD. The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.

COMMUNITY CLOUD. The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.

PUBLIC CLOUD. The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.

HYBRID CLOUD. The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds).

2.1.3.4 *Conclusion*

This section introduced the terms used in cloud computing environments. Now we map the cloud components previously defined with security components used to protect traditional infrastructure.

2.1.4 *Virtualization components*

In the cloud, pooled networking and computing resources may be seen from two different but complementary views. The *networking view* abstracts the network resources, i.e., the successive protocol layers that encapsulate the data to be transmitted in the

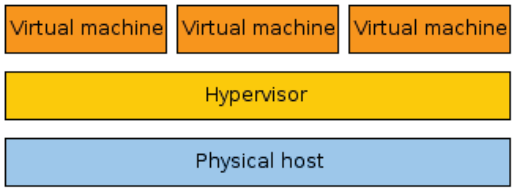


Figure 4: Cloud components.

cloud. Orthogonally, the *computing view* captures the computational and storage resources of each machine at different abstraction levels (software and hardware), e.g., processor, memory, devices. Ensuring end-to-end isolation requires a fine-grained control of information manipulated in each layer crossed along the data path.

To simplify, we consider three main layers in a IaaS infrastructure: *physical*, *OS* (hypervisor), and *application and/or middleware* (VM-level) (see Figure 4).

- **Virtual machine.** A VM, domU or guest, is the workload on top of the hypervisor. This machine believe that it is its own unit with dedicated resources. Thus, multiple VMs are consolidated on one single physical machine. The workload handles user application, administrator domains and virtual appliances.
- **Hypervisor.** The Virtual Machine Monitor [186] (VMM) is a component handling physical resource sharing amongst multiple concurrent virtual machines. To make an analogy with operating systems architecture, the hypervisor has similar role for VMs as the kernel for user spaces.
- **Host Machine.** The physical host manages physical resources and divide them between VMs. The CPUs, memory and devices are shared dynamically, meaning that a memory peak of one VM result in extended RAM allocation.

Multiple users sharing the same resources (multi-tenant) raise the question of isolation: How to deliver dedicated environment without impacting others ?

2.1.5 *Resources isolation*

Virtualization opens new attack vectors, as shown recently by many loopholes exploited in each layer. The main goal of these attacks is to bypass VM separation, or isolation, and gain access to private information. Compromising the hypervisor by a side-channel attack may leak information between VMs sharing resources [169]. Spoofing the virtualization layer is also possible as in the BluePill rootkit [174] to stealthily put a backdoor in place of the hypervisor. Another possibility is to fully bypass intermediate layer controls: the guest OS might directly access virtualized devices to improve performances and lower virtualization overhead. Similarly, the applicative layer could attempt to spoof Address Resolution Protocol (ARP) requests to break physical isolation (e.g., MAC filtering).

Thus, confidentiality requires in-depth defense at each level. Next, we provide a brief overview of some existing isolation mechanisms in each layer, summarized in Table 4.

Layer	Networking View	Computing View
Application (VM)	Application-level firewall: WAF... Virtual firewall: VShield App... SSL/TLS VPN	Antivirus: VShield EndPoint VM introspection [49]
Hypervisor	Virtual switch [50, 149, 136] System-level firewall: iptables, ebtables L2/L3 VPN IP overlay network [106]	Security API: [141] Security modules: [35, 176]
Physical	Dedicated network equipment: firewall (VShield Edge), physical switch, router VLAN L1 VPN Link overlay network [195]	MMU/IOMMU Hardware-assisted virtualization: Intel-VT [211], AMD-V [22], Nested paging [216], SR-IOV [67] VNICs [209]

Table 4: Some Solutions for Cloud Resource Isolation.

2.1.5.1 *Physical Layer*

NETWORKING VIEW At this level, network isolation relies on dedicated network equipment like switches, routers, and firewalls. Besides evident physical separation, IEEE 802.1Q-compliant VLANs enables to segregate virtual networks on the same physical infrastructure. It is also possible to create network overlays at the link layer [195] or above [106]. Firewalls can filter packets in a fine-grained manner using Access Control Lists (ACLs). They can be configured from a console via a serial port, or by accessing a tiny Web server directly integrated into the equipment.

COMPUTING VIEW The computing view is far more complex with hardware resource virtualization, a technology that allows a single physical equipment to offer seamlessly several resources abstractions to the OS, such as Network Interface Controllers (VNICs) [209], Input/Output Memory Management Units (IOMMUs) that map device-visible virtual addresses to physical ones, and special instruction sets to deliver virtualization in processors (Intel-VT [211], AMD-V [22]).

2.1.5.2 *Hypervisor Layer*

NETWORKING VIEW One level up, the hypervisor provides control through virtualized network interfaces and firewalls. The networking view is richer with kernel modules that extend physical switches into virtual ones. The two main competing solutions are Open vSwitch [149] and Cisco Nexus 1000v [50]. Both are similar, but Open vSwitch implements OpenFlow [136] to offer precise control of forwarding tables going well beyond ACL-based administration. Further control on communications is possible thanks to system-level firewalling solutions, e.g., iptables filtering rules for IP packets, ebtables rules for the link layer, or solutions like VMware VShield App.

COMPUTING VIEW In the computing view, resource isolation can be performed through hardware-assisted instructions or emulation. Type 1 hypervisors seem most suitable for fine-grained control over resources. Generic hypervisors are actively maintained, with both open source and commercial solutions. VMware ESX is a mostly closed platform, control remaining limited to a fixed set of APIs, and thus difficult to extend. The Kernel-based Virtual Machine (KVM) and Xen use Intel-VT or AMD-V for instruction processing, IOMMU to separate resources, and deliver a device driver

to manage virtualization hardware. KVM implements VMs as Linux processes, and therefore benefits from a wide variety of standard tools for process isolation and management (details in section 2.2.1). Security modules [35, 176] for isolation are available for Xen based on SELinux – thus directly supported by KVM with the advantage of built-in isolation of Linux kernel back end drivers. The libvirt library is also increasingly used to administer a wide range of hypervisors and modules. It provides a higher-level hypervisor-independent interface to control the virtualization environment. Security modules for this library are available, e.g., sVirt [141] providing Mandatory Access Control security to isolate guest OSes and specify access permissions.

2.1.5.3 *VM Layer*

The VM essentially relies on the hypervisor capabilities for computing and networking isolation. However, a growing number of virtual appliances are already available to filter network data (virtual firewalls such as VShield Zone/App from VMware), to monitor the VM security status (VM introspection [49]), or to isolate a group of compromised VMs by defining a quarantine zone (anti-virus suites). These solutions run into conventional user/group administration problems such as permission management to determine the domains in which users, applications, and devices can be added. Some of those solutions provide little or no explicit interface to manage remotely the other VMs. Moreover, in the guest OS, security control usually remains limited to user-land, while kernel modules provide richer and stronger isolation.

2.1.5.4 *Conclusion*

Overall, those solutions suffer from three main limitations:

1. **Heterogeneity.** Available mechanisms are highly heterogeneous, lacking of an overall architectural vision regarding their orchestration into an integrated security infrastructure.
2. **Scalability/Maintainability.** The different security configurations of previously underlined mechanisms (e.g., VM-level firewall rules and physical-level ones) induce scalability and maintainability challenges: even if a cloud environment may be tuned to meet specific needs, nested dependencies between views and layers can become appalling and virtually impossible to solve, excluding the "by hand" approach to security management.
3. **Dynamicity.** The extremely dynamic character of the cloud (e.g., with VM live migration between physical machines), and the short response times required to activate system defenses efficiently, make the problem even more complex.

A **flexible**, **dynamic**, and **automated security management** of cloud isolation mechanisms is thus clearly lacking today. Furthermore, an in-depth description of the new hypervisor mechanisms follows to identify new challenges raised by the virtualization layer.

2.2 HYPERVISORS

This section describes hypervisors and their security. First we present the architecture of widely used hypervisors, with a comparison of their specificities. Second we skim solutions that leverage hypervisor security through a variety of approaches.

2.2.1 *Virtualization architectures*

Virtualization is a technology to abstract physical resources to users. Even if this layer was introduced to compensate problems inherent to heterogeneous physical architecture, such as privileged mode, it is now an essential component of computer architecture. Virtualization solutions provide benefits in costs, efficiency and security [97]. Achieving such benefits is a difficult task, and one has to carefully consider reliability towards virtualization layer. The design and the placement of the layer are crucial, to select the more appropriate semantic view. Indeed, control is finer while closer to the bare-metal, but is highly error-prone. Conversely, reusing building blocks such as memory components for convenience results in coarse-grained control. Thus, virtualization is available at different level depending of the needs, as detailed into Section 2.2.2. Figure 5 gives an overview of physical machine consolidation.

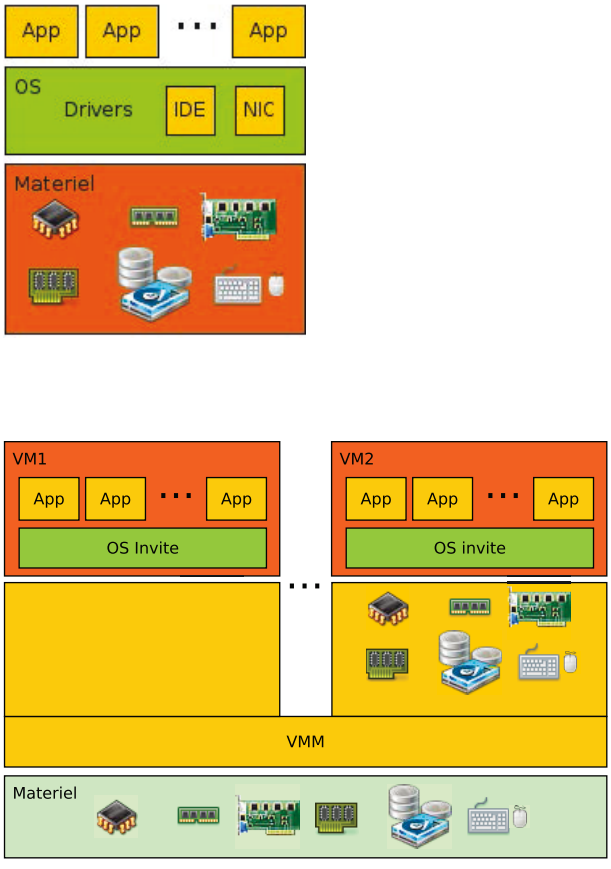


Figure 5: (a) Without VM; (b) With VM .

Type I The hypervisor, or virtual machine monitor, is the main component of the virtualization layer. The Type I virtualization, also named native virtualization, let the VMM runs directly on the physical resources. Virtual machines are two levels above physical resources.

This kind of virtualization provides very low overhead as the hypervisor is cling to the bare metal. However, this low-level access requires access control to resources and pretty much every security measure usually performed by commodity kernels. The most deployed Type I hypervisors are Xen, VMWare ESX and Hyper-V.

Type II The Guest OS/Host OS virtualization is the most used solution on average computers, also named Type II or hosted virtualization. The existing operating system (Host OS) runs a third-party virtualization software, and share existing resources among several guest operating systems. Each guest (Guest OS) consist of a virtual storage image file and a virtual hardware description file. The separate VMM application handles VM execution and distribution of shared resources, e.g. the network card. The VMs are not aware of the network virtualization, and the hypervisor provide strong isolation to deliver different network to VMs. Virtual machines are three levels above physical resources.

This kind of virtualization benefits from the operating system facilities: device drivers, processes isolation and privileges separation. Nevertheless, it inherits operating systems flaws and lower security in an error-prone environment. Notwithstanding, disk I/O are strongly impacted and provide uncomfortable experiences. Other operations are close to native speed. Two softwares lead this technology for desktops. VMware Player is a free closed source solution from VMware, and VirtualBox from Oracle. Servers running under Linux inherit the KVM hypervisor as a kernel module to load.

2.2.2 *Virtualization mechanisms*

2.2.2.1 *Emulation*

The emulation was the first virtualization mechanism to handle and interpret assembly code. The software mimic a specific set of physical resources, and the guest is thus able to run seamlessly on the system. This technique is often used to play console video games on personal computers. Guest assembly code is interpreted and transformed into host instructions. Figure 6 details the emulation process for an ARM instruction on Intel x86 architecture.

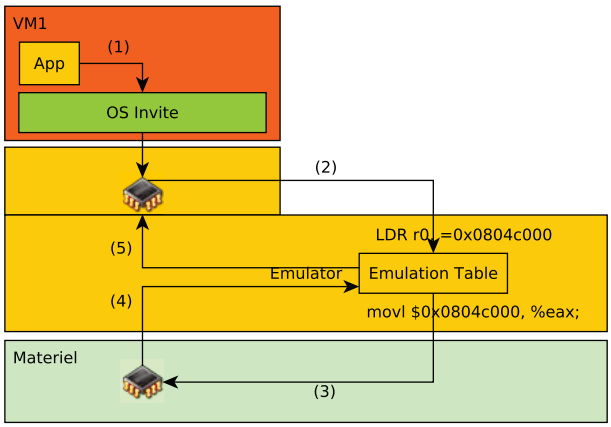


Figure 6: ARM on Intel instruction emulation example .

The process, named binary translation, is heavy and slows guest execution by a factor 100 to 1000. The obvious concept of static binary translation consists of translating all of the guest code into host code without running it. Some peculiarities imply that some basic blocks are referenced by indirect branches, thus only known at run-time. Dynamic binary translation solves the caveat with on-the-fly emulation. Conventional emulators fetch, decode and execute instructions in a loop, thus implying major overhead. Process optimization relies on the basic block (BB) granularity and that the vast

FEATURE	BOCHS	QEMU	VMWARE PLAYER
Guest architecture	x86	x86, ARM, SPARC, PowerPC	x86
Emulation	Interpretation	Dynamic binary translation	Dynamic binary translation
Disk Image Format	All	Raw, QCOW2	Raw, VMDK

Table 5: Main emulators comparison.

majority of execution is limited to few BB. The memoization process then keep the translated representation to speed up further call to this code sequence. Intel Pentium 4 processors, with the NetBurst microarchitecture [123], has an execution cache to store up to 12K of previously translated operations.

SECURITY The main security bottleneck is the respect of architecture manuals into the emulation tables. Several instructions have side effects, complicating the emulation process. However all actions for a specific instruction are gathered at a fixed place into the emulation table, simplifying tests and security analysis.

TECHNOLOGIES Bochs [129] is an open source emulator oriented towards x86 and x86_64 platforms. Instructions are interpreted to stay as close as possible to the expected behavior, with no side effect. Despite low performances, it is a common tool to analyze malwares in a transparent environment [187]. QEMU [32] is a free, open source emulator which support a large number of guest architectures (x86, x86_64, MIPS, PowerPC, ARM, SPARC, m68k). Dynamic binary translation offers significant performance improvements over Bochs interpretation, and is thus the most common emulation software nowadays. Our tests depict Bochs as more efficient for OS debugging purposes, while QEMU performs better in pure execution speed.

2.2.2.2 *Partial virtualization*

The partial virtualization approach simulates several classes of physical devices. For example, memory management, where OS uses partial virtualization to segment physical memory into several isolated address spaces. Processes can use the same virtual memory addresses, as the OS traduce them into different physical addresses. Thus, it enables resources sharing above a common OS and provides some kind of process isolation. However, it is not enough to deliver isolated instances of OSes. Windows NT4 and Linux are both using this virtual addressing mechanism to partition applications and enhance global system security. The amount of virtualization mechanisms is flexible: one can offload some tasks toward hardware. Thereby a hard drive can be dedicated to a VM, and all further accesses will be handled by the VM. This pass-through approach is applicable to the vast majority of host devices.

SECURITY The hypervisor cannot perform isolation checks anymore and the VM is responsible of resource management and security. Performances are usually better as the VM will use dedicated drivers to access the device and not the generic hypervisor device driver.

TECHNOLOGIES While some hardware deliver partial virtualization on-demand, such as the address space virtualization enabled on all current computers, it is hard to detail partial virtualization technologies. We argue that all components building up computers represent potential partial virtualization technologies.

2.2.2.3 *Operating system-level virtualization*

The OS also performs virtualization at the system call interface to leverage performances, with isolated environments sharing the kernel. This approach is often referred as **lightweight virtualization**. As seen before, processes memories are isolated by the close collaboration of the MMU and the kernel. Each process memory space is abstracted to give the illusion of a large contiguous address space, while the kernel ensures that no other process can be altered. This feature translates the RAM address space, or physical addresses, to logical address space. It is also applicable at the file-system level with reduced access to the current one (chroot). An HTTP server delivers static pages from a single directory, thus giving an abstract view of the file-system is sufficient for service and prevent further exploitation. Containers are the instances of flexible user-space isolated environments. All containers share the same kernel, thus offering little to no performance overhead. However, flexibility and security are directly impacted. The kernel delivers the same OS as the host, and a single security failure into the kernel compromise all guests. The attack surface to perform privilege escalation is large, ranging from subtle partition mounting [134] to unsafe capabilities [132], including legacy kernel one.

SECURITY While chroot is not a security mechanism, several other software (see Table 7) offer fine-grained isolation of file-system, network, memory and root access. The *cgroups*, *user name spaces* and *capabilities* Linux mechanisms manage resource allocation, isolation and root privileges reduction respectively. To provide a secure execution environment among users, secure computing mode (seccomp [15]) leverages system calls interface and deliver reduced kernel interface. The seccomp mechanism is a one-way isolation mechanism restricting system calls to read(), write(), sigreturn() and exit() with opened file descriptors. Every other system calls result in a process termination with a SIGKILL. The restriction performs control on computing resources, but handles neither memory issues nor file-system access control. The setuid sandbox mechanism switch uid and gid enforce restrictions on debugging other processes, sending signals and file-system access control. Both seccomp and setuid sandbox are common to isolate browsers threads, especially chrome. The SELinux sandbox enforces MAC policies to process thus controlling available system calls, inheritance and temporary space access. The underlying technology use Multiple Category System (MCS) or Multi-Level Security (MLS) to label files and delegate control to the SELinux MAC [2].

Mandatory Access Control is available both inside the kernel [185, 178, 92, 31] with Linux Security Module (LSM), and outside the kernel [189, 105] to leverage vanilla kernel Trusted Computing Base (TCB). They are compared on Table 6

Security Enhanced Linux (SELinux [185]) inherits the Flask security architecture to provide a flexible non-discretionary access control. The clean separation between policy enforcement, inside the kernel through Linux Security Module (LSM), and policy decision-making, encapsulated in another component, allows a low overhead and leverage modularity. Processes and system objects (e.g. sockets, files) are assigned a context with a role, a user and a domain. Administrator-defined policies describe the

Technology \ Properties	LSM	Complexity	Cohabitation
SELinux	✓	✓	✓
SMACK	✓	✗	✓
TOMOYO	✓	✗	✓
AppArmor	✓	✓	✓
Grsec	✗	✓	✓
RSBAC	✗	✗	✓

Table 6: OS-level security

possible transitions of the user, and how processes can interact with system objects. Nevertheless, this fine-grained mechanism require complex administration to enforce conform behavior.

Simplified Mandatory Access Control Kernel (Smack [178]) tackles the complexity problem by leaving out the role based access control and type enforcement. Thus, tasks and kernel objects are labelled and controlled upon access to allow execution while matching. It is a more permissive mechanism that cannot fulfill SELinux completeness, but simpler to setup and to administrate through extended file-system attribute. Several commands mimic conventional UNIX program to access files with facilities to set and modify labels.

TOMOYO [92] is a LSM to perform MAC on Linux, and was integrated into the mainline version 2.6.30. Programs are profiled automatically to establish isolation boundaries, unlike previous label-based security mechanisms. It is able to provide SELinux security level with a lighter administration, and support AppArmor [31] cohabitation for multi-layered isolation.

TECHNOLOGIES OpenVZ [116] is a patched Linux kernel providing subsystem creation, isolation and resource management. Containers have their own set of files, users, groups, process tree, network, devices and Inter-Process Communications (IPCs). Underlying resources are shared according to administrator-defined limits for the scheduler and hard drives. Furthermore, virtualization facilities such as memory snapshots and live migration among physical servers are natively supported. The overhead is minimal with near-native speed. Virtuozzo [198] is the commercial counterpart of OpenVZ, with Windows support.

LXC [115] combine previous mechanisms to provide containers close to OpenVZ. Virtual file-systems are handled by chroot, resource management by *cgroups* [162] and a modified kernel interface to intercept system calls. Checkpointing and conventional administration tasks (excepted live migration) are supported to provide a lightweight virtualization solution. The LXC architecture allows developers to create even lighter containers by using only some namespaces. The *shared subtree namespace* [56] provides on demand isolated copies of the current file system, with several files shared among users for performance reasons. The *pid namespace* [117] allows multiple processes with the same Process IDentifier (PID) in different environment. It is an essential feature for live migration where isolated environments have their own process PIDs that cannot be modified. The *pid* structure is thus modified to integrate a pointer towards a *upid*, a new structure to represent the PID into a namespace. To some extent, the network has to do the same kind of isolation to provide seamless network port usage (e.g. multiple

SOLUTION	OS	FS ISOLATION	ROOT ISOLATION	NETWORK ISOLATION
chroot	Linux	✓	-	
LXC	Linux	✓	-	✓
Linux VServer	Linux	✓	✓	✓
OpenVZ	Linux	✓	✓	✓
Parallels Virtuozzo	Linux, Windows	✓	✓	✓
Jail	FreeBSD	✓		✓
Sandboxie	Windows	✓		✓
Returnil	Windows	-		
iCore Virtual Accounts	Windows XP	✓	✓	✓
Bufferzone	Windows			

Table 7: Kernel level isolation mechanisms .

HTTP servers). The network isolation is handled by the *net namespace* [57], introduced by adding a new *net_ns* field into the *net_device* kernel structure. Such implementations involve control of the L2 OSI stack level for low-level network granularity, extended to L3 to prevent user from creating new interfaces and grab L2 information. Thus each *net namespace* brings its own network interfaces, IP addresses, routing tables and iptables rules. The *ipc namespace* leverages System V IPC mechanism with on-demand private IPC namespace that is not shared with any other process. It is required to perform IPC isolation on legacy softwares using IPC variables derived from inodes. The *uts namespace* controls the get/sethostname syscalls to adapt domain information and provide compatibility with tools relying on those information (e.g. sudo). The *user namespace* leverages the *pid namespace* to users, with a new root user created for each namespace.

2.2.2.4 *Full and paravirtualization*

A level below, system-wide virtualization delivers a complete VM with dedicated virtual devices, virtual processors and virtual memory. The full-virtualization challenge was formalized [164] to define a VMM capable of virtualizing a full set of hardware resources: (1) Equivalence/Fidelity, (2) Resource control/Safety and (3) Efficiency/Performances. From those principles may in turn be derived virtualization theorems, explaining that "an effective VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions". Unfortunately some architectures do not conform [19], e.g. with Intel x86 providing 17 sensitive instructions (e.g. *popf*) accessible from unprivileged state. Two counter measures [28, 217] enabled full virtualization on the Intel x86 architecture.

XEN Xen [28] has been the software to perform a breakthrough into virtualization architecture and performance, with a strong cooperation between the hypervisor and guests, also called paravirtualization. Initially developed by Citrix, a strong community (e.g. Google, IBM) continually improves the hypervisor. The Xen segmented model delivers components with specific roles to apply the least privilege principle.

An administrative VM (domo) manages network, I/O and memory interactions of guest VMs (domU). To illustrate, the domo and the underlying hypervisor are close to the third-party application in the hosted virtualization (VM Player and VirtualBox). The domo is a custom Linux kernel aware of the underlying virtualization, while domUs run in full virtualization mode. The latter provides an entire system (Hard drive, NIC, CPUs) and thus run unmodified guest OSes. Sensitive system calls are trapped and modified to be compatible with the Xen hypervisor, and thus respect [164]. Nowadays, the Intel x86_64 provides hardware facilities to leverage virtualization, supported since Xen 3.0. While orchestrating resources among virtual machines, Xen does not provide access control facilities. The Xen Security Module (XSM), derived from LSM, handle third-party security components, e.g. to restrict access to resources with MAC or DAC. The Flux Advanced Security Kernel (Flask) XSM reuses the SELinux rule syntax to describe policies to apply. The sHype XSM from IBM, adapted from the original rHype hypervisor, is oriented toward large adoption rather than critical systems. The code is not intrusive (about 2600 lines of code) and is built with scalability and performances (1 percent overhead) in mind. The Xen adaptation handles Chinese Wall and Simple Type Enforcement to provide a fine-grained control.

The VM security inherits physical facilities to outsource cryptographic functions [159], enhance the non-executable pages protection [175], control hypercalls [224] or even protect critical structures (IDT, hypercall tables and exception tables). However those protections are neither available in the main development branch, nor in the public domain. VM introspection [63, 156, 155] provides low-level access to analyze VM state from the hypervisor, detect VM intrusions [85], forensic memory analysis [191, 64], real-time file protection [238] or malware analysis [62, 107]. Equally important, the network security is twofold. First, the domo firewalls network streams with facilities integrated in Linux: ebtables, iptables/netfilter or pf. Even more complete firewalling solutions, such as Shorewall or L7-filter, can be deployed seamlessly into the domo and enhance networking security. Second, the modular architecture allows easy application of the least privilege principle.

The Xen attack surface, with an attacker controlling one VM with network access, is depicted in [68]. For the network path, a bug into the hardware driver, the bridging component or in the back-end of the split driver leads to the control of domo and thus the whole system. The Driver Domain is an unprivileged VM to move those sensitive components, without interactions with the domo. Nevertheless, the attacker is still able to access the network adapter and the front part of domain sharing the split driver. The tool stack to build VM on-demand parses file system and menu, and can be corrupted with subtle format specification [233]. Alike Driver Domains, pvgrub acts as a small OS to delegate the parsing towards domU, making parsing exploitation meaningless. Underlying device emulation is marred by the same defect. *Stub domains* are small services VM running one application, ideal to isolate the Qemu emulation program and thus confine the exploitation. To a further extent, several solutions aimed at reducing the TCB [142, 172, 54] through the disaggregation of the domo.

VMWARE ESX/ESXI VMware ESX and ESXi are the Type I virtualization softwares of VMware. ESXi is free closed source software which has very special hardware requirements beyond those of ESX. They share the same *vmkernel*, but ESXi uses a smaller footprint, thus with a faster boot time at the expense of the Service Console. Anyhow, ESX is end-of-life software and have to be upgraded to ESXi. There is an Embedded Edition which supports booting from SAN, scriptable installations and Ac-

tive Directory (AD). The *VMkernel* micro kernel implements a remote command line interface (CLI) and a common information model (CIM) to manage the configuration and the hardware. New modules can be loaded to the kernel, assuming they have been signed by VMware. The kernel scans the memory to detect integrity violations such as buffer overflows. The ESXi attack surface is hard to estimate as there is no source code. Virtualized I/O are handled by the VMkernel, while passthrough I/O can be selected on a per-VM basis. The VWware *hypercalls* increase the surface. However, security facilities to monitor and control CPU, memory and network are isolated into *VMsafe* VM. The VMkernel allocates resources and manage memory, with a systematic wipe of reallocated pages to prevent information leaks. Likewise, the IOMMU enhance memory isolation and address Direct Memory Access (DMA) attacks. Optimized memory operations, such as *transparent page sharing*, *ballooning* or compression, require close attention to respect the isolation. Memory pages are shared to avoid multiple allocation of the same object, which is particularly effective while running the same OS multiple times. However, missing a single write operation, e.g. under heavy load, result in information leaks. ESXi combines emulation, paravirtualization and hardware assisted virtualization to reap benefits from each technology. Efficiency and security of binary translation compared to the first generation of hardware virtualization, paravirtualization with *VMWare Tools* for a better experience and support of the 64b guest architecture. Recent versions also support VT-x and AMD-v for both the 32b and 64b guest architectures, and nested paging (Rapid Virtualization Indexing (RVI) and Extended Page Table (EPT)). The memory virtualization has hardware assistance through Intel EPTs and AMD RVI. This extra indirection layer replaces software MMU, thus improving translation time by 42% [218]. Likewise IOMMU is available to enhance resolution for I/O communication at the hardware level.

MICROSOFT HYPER-V Microsoft Hyper-V [214] appeared with the Windows 2008 OS, and is the first hypervisor based on the Windows OS. While ESX and Xen are Linux oriented, users have to be at ease with UNIX or Linux environments and commands. Hyper-V administration is done through conventional Windows GUI.

The Microsoft Hyper-V solution is a bare-metal hypervisor with a microkernel architecture [139]. Drivers are not implemented into the hypervisor, but offloaded into a less privileged ring. The hypervisor is divided in two blocks. Essentials virtualization functions such as virtualization service clients (VSC) and OS kernel are in ring 0. The root partition gather portions of traditional hypervisors offloaded to build a micro hypervisor. Virtualization stack containing VM worker processes is in ring 3 (root partition), the same level as guest applications. The TCB is composed of: root partition excluding VM worker processes, and the windows hypervisor.

The current Hyper-V hypervisor focuses on strong isolation between partitions and the protection of both integrity and confidentiality of guest data. Separation is performed through unique resource pool per guest, separate worker processes and guest-to-parent communications over unique channels. The non-interference is guaranteed by protected computations, no communications between guests through VM interfaces are allowed. However some attacks are still possible (inference attacks, covert channels) and availability is not guaranteed.

KVM The Kernel-based Virtual Machine [119] (KVM) is an open source project started in 2008 which was quickly adopted by the Linux kernel 2.6.20 branch. Thus, it is the main virtualization solution of Linux-based distributions (e.g. Arch Linux,

Ubuntu, Fedora). The hypervisor is built up by two components in the host OS: a driver and an emulator. The hypervisor module transforms the usual linux kernel into a real hypervisor by inserting a kernel driver. VMs are considered as host processes and inherit isolation mechanisms and scheduling optimization of the host kernel. The QEMU program delivers virtual devices to VM, but the CPU and the memory can be handled by hardware through virtualization facilities (Intel Virtual-Machine Extensions (VMX) and AMD Secure Virtual Machine (SVM)). Software-only virtualization is always available as a fall back with QEMU handling all the emulation.

KVM is secured by conventional Linux protection mechanisms. The MMU protects physical memory by controlling a virtual address space for each VM. The kernel-level security mechanisms detailed in Section 2.2.2.3 also apply to enhance OS security.

MICROKERNEL APPROACH Eventually, monolithic hypervisor is a virtualization solution simpler than a conventional kernel but still complex. The driver model is implemented to enhance hypervisor capabilities to deliver alternative devices to virtual machines. The microkernel hypervisor uses simple partitioning functionality to leverage reliability and minimize TCB, and are explained in details into 2.2.7.1.

2.2.3 *Security testing*

2.2.3.1 *Statistical distribution of hypervisors vulnerabilities over time*

Figure 7 depicts vulnerabilities found on the NIST National Vulnerability Database (NVD) for KVM, Xen, ESX and Hyper-V over time. The score represents vulnerability criticality, 10 being the most. The area underlines the number of hypervisor versions impacted, a small point designing one specific version, while a larger one means that tens of versions are vulnerable. Finally, the color reflects attack complexity: green for easy, brown for medium and blue for hard. We can see that more vulnerabilities are found as the time goes by for all hypervisors. The impact and complexity of attacks become more important.

VULNERABILITY CLASSES The hypervisor attack vectors [161] are grouped in 4 categories. The first category gathers vectors related to virtualized hardware with Virtual CPUs [13], Symmetric MultiProcessing (SMP) [11], MMU [9], Interrupt and Timer Mechanisms [10], I/O and Networking [14], and Paravirtualized I/O [5]. The second category groups hypervisor-VM operations, such as VM Exits [12] and Hypercalls [8]. The third category contains interfaces needed to manage VM [4] and their associated management softwares [6]. And the fourth category handles hypervisor modularity and the variety of extension to leverage the inner security [7].

Table 8 depicts Common Vulnerabilities and Exposures (CVE) repartition for the KVM hypervisor among the categories. The majority of attacks affect the first category, with 15 related to device emulation (~40% of all the attacks). Therefore hypervisors have to secure the device drivers for enhanced security.

2.2.4 *Hypervisor code analysis*

The hypervisor evaluation, similar to application testing, is done through the static analysis of the code or by dynamic debugging.

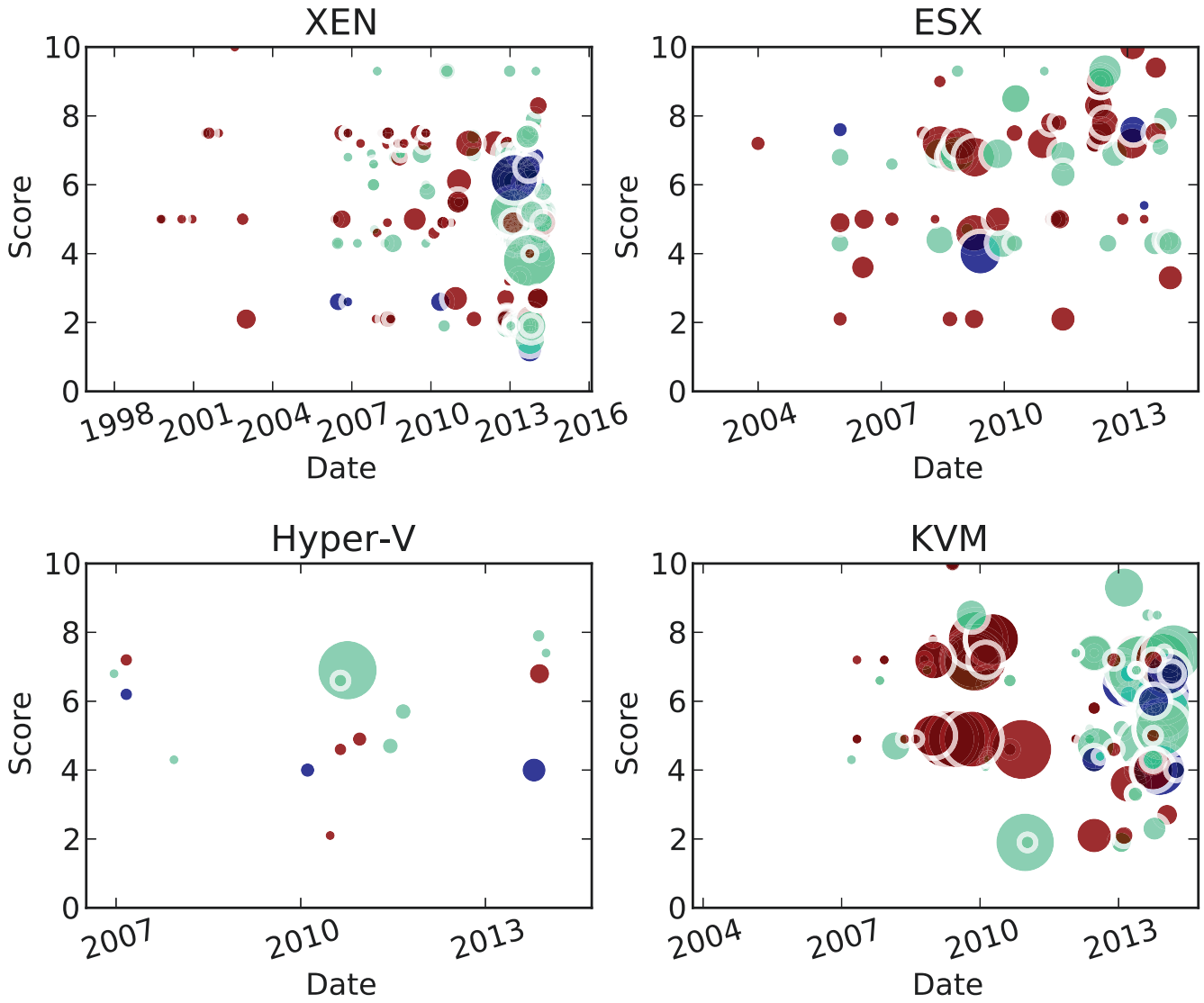


Figure 7: Commodity hypervisor vulnerabilities.

ATTACK VECTOR	KVM
Virtual CPUs	8
SMP	3
Soft MMU	2
Interrupt and Timer Mechanisms	4
I/O and Networking	10
Paravirtualized I/O	5
VM Exits	2
Hypercalls	1
VM Management	2
Remote Management Software	1
Hypervisor add-ons	0
Total	38

Table 8: KVM CVE repartition (from [161])

STATIC Static analysis of open-source hypervisors underlines potential flaws in the code [38].

DYNAMIC Black box testing performs fuzzing on specific attack surfaces to extract unexpected outputs. Instructions fetched and executed by the CPU are straightforward to test: sending random instructions and comparing results between bare-metal and virtualized environments. While the differences usually characterize an environment and thus provide virtualization detection [152], some lead to privilege escalation in a generic way [231]. The communication with components linked with northbridge and southbridge is an inexhaustible source of experimentation. Timing attacks related to TLB resolution among VMEXITS discriminate memory accesses, ranging from simple flush detection for virtualization detection [173] to confidential information extraction [169, 236] – keystrokes and private certificates. Random inputs against virtual devices and their drivers therefore uncover a number of vulnerabilities in multiple hypervisor solutions [150].

2.2.4.1 *Conclusion*

We will use fuzzing to test our solution applied to hypervisors, showing the extensibility and measuring maximum throughput (Section 4.3).

2.2.5 *Virtualization peculiarities*

Another research track aims at detecting discrepancies specific to virtualized environments to evaluate virtualization transparency.

2.2.5.1 *Virtualization detection*

CATEGORIES Virtual machine detection can be formalized in three categories:

Logical discrepancies such as bad emulation and devices specific to hypervisors.

Resources discrepancies such as TLB misses.

Timing discrepancies with attacks based on local timers and remote timers.

LOGICAL DIFFERENCES J. Rutkowska initialized the virtual environment detection playground in 2004 with Redpill [173]. This detection is based on the address of the IDT. The idea is that each physical processor have one and only one IDT. Thus hypervisors have to create the IDT of virtual processors to different locations. Physical IDT is located near the address 0x80000000, when virtual ones are above 0xc0000000. While it was true at this very moment, this technique is obsolete and we were not able to detect virtual environments based on the address of the IDT.

P. Ferrie identifies several techniques to detect virtual machine emulators [80, 79]. He is able to detect emulation discrepancies, such as wrong emulation of instructions. Specific hypercalls related to VMTools (accelerated graphics, drag and drop) are also tested, as they are very specific and does not respect usual behaviors. A typical attack is to use an instruction over the authorized size (say 15B on Intel), which raises a fault on bare metal while completely ignored on Qemu. D. Quist created a tool to detect virtualization, vm-detect [166].

The conclusion underlines that no virtual system is completely hidden, nevertheless malware will have to infect virtual environment infections as more and more system are virtualized [87]. T. Raffetserder also describes discrepancies, but from bad emulation instructions marked as "won't fix" [167] from intel documentations and update specifications. CPU caches are modified to leverage timing attacks, with more precision and accuracy.

RESOURCES DIFFERENCES X. Chen creates a taxonomy of different detection techniques, and introduces network based detection [46]. The analysis of TCP Round Trip Time (RTT) showed specific behavior for both bare metal and virtualized environments. There is also a proposition to fake a virtual environment and thus evade malware infection.

DIFFERENTIAL ANALYSIS R. Palaeri automates the creation of detection techniques by fuzzing CPU and compares registers [153], finding more than 20 000 differences in few hours. B. Lau automatically compares the behavior of malwares through differential analysis [128]. The result highlights that less than 1 percent of malwares try to detect virtual environments.

M. Lindorfer extends the malware and detection taxonomy [131] by providing new detection techniques. G. Pek exposes how to detect virtual environment specifically created to be stealthy with an implementation called nether [158]. This paper presents new techniques to detect VT instructions, with a classification of analyzers and the conclusion that transparency is not achievable.

2.2.5.2 *Seamless virtualization*

On the other side, A. Dynaburg formalizes anti-detection mechanisms and how to use available technologies (i.e. VT-*) to create a transparent virtual environment [62].

Ether, the name of the tool, is able to dynamically unpack malwares. Some architectural defects are analyzed, such as the Intel oxfoof bug, which represents unexpected behavior from the processor specification but that is present on all processors. Counter measures to escape detection are analyzed and explained.

PROTECTING RESOURCES A. Nguyen provides an architecture to virtualize only vital components and use passthrough for the rest [145]. Instructions consuming CPU cycles, such as VMExits, are no more caught and decrease hypervisor footprint. The main drawback is that this system is only able to virtualize one VM, losing one of the main benefit of this technology. E. Keller follows the same idea, creating a hypervisor with a very low attack surface and hard to detect [201]. D. Kirat introduces physical virtualization [114]. Many virtualization concepts are applied to bare-metal, such as snapshots with volatile memory based on RAID mirroring. Registers and interrupts are saved when OS boots, and are restored when requested. Each restore is taking about 4 seconds. C. Thompson analyzes MAVMM and tries to patch defects [207]. TLB flushing can be avoided with new VT instructions, while timing attacks are still a major issue. The major contribution is the code base freely available. D. Perez examines hypervisors entry points to measure the attack surface [160]. Timing attacks using instruction handled by the hypervisor (RDTSC and CPUID) are given as examples.

2.2.5.3 *Conclusion*

From those analyzes of virtualization peculiarities, we extract rules to feed our decision engine and detect when a malware tries to adopt a different behavior than bare metal.

2.2.6 *Protections*

Multiple approaches have been studied to solve previous problems through execution flow, memory management and network.

2.2.6.1 *Control flow isolation*

The attack surface varies as a function of the virtualization mechanism, and implies different protection mechanisms.

The emulation (Section 2.2.2.1) resides on an instruction analysis engine which fetches and executes callback function registered to handle them. Here the attack surface gathers the engine and callbacks. These, essential for non-hardware assisted hypervisors, are error-prone but provide fine-grained control of edge effects. However, no actions inside the VM can affect physical devices as all parts are emulated. The engine is isolated by Host OS protection mechanisms (MMU) to prevent privilege escalation when breaking out of the hypervisor.

The system-level virtualization (Section 2.2.2.3) shares the Host OS kernel between all guests, and thus inherits a wider attack surface. This is the compromise chosen by this solution, where component pooling results in higher performances. Unfortunately, Linux security is bypassed several times a year [17, 66].

Hardware-assisted virtualization offload instruction analyzes to the CPU, like the system-level virtualization. Isolation is enforced by design, with Virtual Machine Control Structure (VMCS) internal verifications. Thus, hypervisor defines only contexts,

registers and bit-masks to regain control. The hypervisor TCB is lighter, reducing greatly chances of vulnerable code. However, closed-source microprocessor code can hide side-effects as explained into 2.2.5. Furthermore, the attack surfaces depend on hardware-assisted mechanisms completeness. For example, virtual memory and devices may be offloaded to the processor, or completely emulated. Once again, there is a trade-off between control and performances. New CPU architectures, such as Intel Haswell [104], tend to lower hardware assistance cost and become reference components of virtualization.

Paravirtualization can emulate instructions or use hardware facilities, and inherits the previous problems highlighted. Moreover, the hypercalls specific to this technology provides direct access to hypervisor internals. Components partitioning is essential to avoid full access to the administration VM (named domo) during an attack.

2.2.6.2 *Hypervisor disaggregation*

Today, every hypervisor has an architecture where components are more or less isolated. The original monolithic structure is now disaggregated in tight building blocks. The components privileges have to be limited to normal behaviors and vital functionalities. We have seen how Xen build stub domain micro VMs to offload services and reduce TCB 2.2.2.4. The same techniques can be applied to other hypervisors, the difficulty depending on the underlying architecture.

2.2.6.3 *Memory management*

To prevent DoSs attacks and optimize system performances, hypervisors attempt to manage memory in several ways [219].

SHARING PAGES A physical machine hosts the same OSes, with the same kernel, and more generally the same data. It then uses sharing pages techniques to enhance performances. The VMs memories are analyzed randomly to aggregate identical data without creating load on the host. The main algorithm calculates the hash of pages, then compares them. When two memory pages are the same, the hypervisor create a read-only page and copy data on it. A standard copy-on-write mechanism will duplicate the page seamlessly when a VM modifies it. The access launches an exception caught by the hypervisor. This technology is name Transparent Page Sharing for VMware and Kernel same-page merging for KVM.

BALLOONING The hypervisor allocates parts of host memory to the VM. At execution, the hypervisor gives this memory to the VM without any possibility of further safe modification, as there is no semantic view of the "inside". A solution is to integrate an agent inside the VM to guarantee the communication with the hypervisor. The agent gathers free pages of the VM to give them back to the hypervisor when out of memory.

MEMORY COMPRESSION The compression mechanism happens when VM memory is filled. The traditional swapping mechanism pours memory pages to the hard drive and thus extends RAM but with a huge performance cost. Accesses are up to 100 times slower. As an alternative, memory is packed into a compression cache. Access times are decreased with only a decompression of the page into the RAM.

HYPERVISOR SWAPPING As traditional OSes, the hypervisor has a swap zone to migrate VM and free space. This approach is simpler than ballooning but has a strong impact on access times. Moreover, new problems appear such as page swaps and the double-swap when the whole system is out of memory.

STOPPING VIRTUAL MACHINES Finally, the hypervisor has a stop feature to kill VM when memory usage is critical. This is more a fall-back solution.

2.2.6.4 *Network technologies*

Hypervisors offer network features at different OSI layers. At the second level, the hypervisor handle VM frames to make them believe that they are alone. At the third level, the hypervisor is a gateway and translates addresses. VMs are on the same subnet, and cannot be seen on the physical network. There are also distributed virtual switches offering a wide range of features close to real ones, such as VLAN, spanning tree or network diagnostic.

NETWORK FLOWS ISOLATION Equipment available into the VM always appear as a dummy network interface. However, the latter hides efficient mechanisms to isolate network flows. The simplest is to recreate a new network stack and to entrust it to the hypervisor. The hypervisor then translates data and transfers it to the physical device, or send data to the virtual switch. There is also the possibility to assign a physical network card to a VM, and pass-through the hypervisor. Thus, the hypervisor loses all control of the device, but the VM has a network card with all functionalities. Finally, the VM can help the hypervisor with a specific driver, allowing better interpretation and delivering high speed.

2.2.6.5 *Conclusion*

We classify those memory management mechanisms as available reactions to apply when dealing with a VM. We also have points to watch and finer control of the security vs performance cursor.

2.2.7 *Architectures*

The inherent code base needed to perform virtualization is tremendous. Thus, aside from well-known hypervisors, new security architectures explored different leads to reduce intrinsic complexity, handle embedded constraints and virtualize them.

2.2.7.1 *Micro hypervisors*

The hypervisor is considered trustworthy and is the key of a secure virtual infrastructure. With arbitrary privileges on VM (inspect/modify states, creation/destruction...), the hypervisor can endanger their confidentiality, integrity and availability. New attacks aiming at the hypervisor represent a technological breakthrough and open a new research era. As seen into 2.2.3.1, attack vectors come from device drivers: network card, power management, graphic acceleration... These are named bounce attacks. A buggy hypervisor driver code is exploited inside a VM, a malicious code is injected to break VMs isolation and backdoor other components.

To prevent these attacks, two approaches have been designed to reduce the size and the complexity of the TCB. Two components are particularly sensitive and have to be protected, the hypervisor core and the administration VM (domo). The first approach hardens the TCB, while the second minimize it.

TCB HARDENING Trusted computing architectures [25] deliver strong integrity of the hypervisor and components through TPM attestation. Otherwise, language techniques [203] protect hypervisor control flow to guarantee executed code authenticity. Thus, one can detect buggy code and malicious code. Finally, sandboxing confines malicious code by hooking device drivers at multiple levels: hardware [181], hypervisor [133], user space [83].

Driver protection domains, such as Nooks [197], decrease buggy code impact on the hypervisor for less that 100 Line Of Codes (LoC). However, it does not protect against malicious code and does not provide remediation. Thus, some approaches offload drivers to user space and constrain their possibilities. The micro driver approach showed a high performance cost.

Another field is to provide memory integrity of the hypervisor, i.e., the fact that the hypervisor can not be modified by the software running at a lower privilege level.

TCB MINIMIZATION Another research axis create new architectures rather than patching weaker ones. The approach is to adapt operating system research evolution (micro kernels, exo-kernels) to hypervisors. The code is divided in two parts, critical and non-critical. The objective is to minimize critical code size by dropping non-essential services. Device drivers can be virtualized, e.g. Hyperwall [199], to enhance isolation without code modification.

A more intrusive approach, DeHype [232], even divide the critical code in two parts. A lightweight KVM is offloaded into the user space, and a minimal hypervisor, HypeLet, replace the original KVM with only critical components. We can see a division of functionalities which is eligible to formal proof. These are named micro hypervisors, or microvisors, and the extreme ones are NoHype or BareBox [200, 114], where all communications are pass-through and the virtualization layer almost disappears.

An interesting architecture, the Nova [192] microvisor divides the hypervisor: the micro hypervisor (with less than 9000 LoC), and virtualized components in user space (named VMM here). The microvisor handles some essential functionalities (MMU, IPC). VMMs handle hypercalls interface or address spaces, and can be recovered dynamically if compromised. Device drivers and applications run out of the hypervisor. However the Nova micro-hypervisor verification project [205] aims only at low-level properties of the code, such as memory and hardware safety and termination, and does not consider virtualization correctness at all [206]. Deploying such architectures implies a lot of new components and offers only few services compared to commodity hypervisors. Also, new versions require new patches, and infrastructure is highly dependent on developers. These new architectures still have small communities.

A second approach, as seen into section 2.2.2.4, disaggregates the domo. For example, Min-V [144] strips down the Hyper-V hypervisor with least used device drivers. The attack surface is then decreased, and tests highlighted that several attack classes are prevented. To a further extent, hypervisor verification tools show isolation properties for a minimalistic model of a hypervisor running on a simplified hardware without MMU and TLBs [30]. Alkassar outline a virtualization correctness proof of a simple hypervisor for a single-core RISC machine with a single level address transla-

TECHNOLOGY	POLICIES	DISAGREGATION	VMI
Xen	Flask, sHype	Qubes, XOAR, HyperSafe	Virtuoso, XenAccess, LibVMI, VMIFMA, VMWall, VRFPS, Ether, VMWatcher
KVM	SELinux	sVirt, DeHype, HyperLock	libvmi
Hyper-V		Min-V	CXPInspector

Table 9: Hypervisor isolation mechanisms.

tion but without a TLB. The result of the verification is a simulation proof, carried out in Microsoft's VCC verifier [53]. It was the preamble of the complete verification of the Hyper-V hypervisor including virtualization correctness [130].

2.2.7.2 *Nested virtualization*

The effective protection may be impossible if the complexity or the infection is too high. Other approaches try to protect the infrastructures even if the hypervisor is compromised. Referred as Hypervisor-secure virtualization (HSV), the hypervisor is not in the TCB. An underlying layer, software or hardware, is in charge of the security. Resource management and security are divided unlike commodity VMM: the hypervisor can launch, stop and schedule VMs, but cannot influence pagination, secured by the security layer.

Nested virtualization performs software HSV. The hypervisor to protect (L1), is virtualized by a minimal secure hypervisor (L0), protected by the hardware. The different virtualization layers can coexist on the same computer by relying on hardware-assisted extensions Intel VT-x: L0 is running as the host, while VMs and L1 are guests. Memory virtualization also depends on hardware capabilities. Emulating paging or using EPT, page tables are merged to avoid an extra indirection level. The CloudVisor [235] architecture enhances security through this paradigm. The compromised L1 hypervisor manage resources, but an extra verification is performed at L0 layer to prevent unauthorized page modification. As Overshadow [45] for OS, pages are also encrypted to prevent L1 modifications. This approach is a promising next step, as L0 abstracts hardware, and is a first solution to homogenize multiple IaaS infrastructures. The VMs are able to run independently of the IaaS provider [230].

2.2.7.3 *Conclusion*

The virtualization layer architecture differs for each hypervisor software. Implementation choices of components that deliver internal functionalities, such as virtual devices mapping or virtual MMU, impact hypervisor behavior. While there is no ideal architecture [98], the hypervisor has to be selected wisely to be in tune with the above virtual machine.

2.2.8 *Layer security mechanisms*

Section 2.1.4 presented several approaches to secure each layer, which are detailed here.

LAYER	METHODOLOGY
Secure virtual machine	Harden guest machine as secure hardware
	Control allocation of physical resources
	Integrity validation, signature checking, virtual encryption
	Monitor change management procedures
	Secure time synchronization
Secure management interfaces	Minimize attack surface
	Strong authentication, encrypted communications
	Least privilege user access
	Log and audit event
	Secure local and remote hypervisor management interfaces
Secure hypervisor	Use attestation and integrity checks
	Patch and update attestation records
	Use care with resource allocation to VMs
	Monitor hypervisor for sign of compromise
Secure host operating system	Use attestation for verification
	OS hardening
	Implement standard network security measures
Secure Hardware	Use attestation for verification
	Control physical access
	Use BIOS passwords
	Remove unnecessary hardware

Table 10: Platform hardening methodology. (from [157])

VM introspection (VMI [85]) leverages hypervisor capabilities and context to supervise VM behaviors. Whether inside the VM with an agent, into a third party appliance, into the hypervisor or even a mix, every introspection solution has its very own features. In-VM monitoring [180] benefits from the VM context, thus allowing a straightforward analysis of behaviors. However, the monitoring component has an in-VM footprint and has to be protected from malicious programs. Security appliances [101] are isolated in another VM to enhance isolation, and cooperate with the hypervisor to analyze the target VM. Those extra detours decrease the reactivity time with twice more VMEXITs.

Several challenges slow-down the adoption of these systems. A legitimate user inserts a USB key into the computer, modifying the IDT and thus the kernel. The hypervisor analyzes root cause and has the daunting task of deciding if this action is legitimate. Furthermore, hypervisor is able to hook specific functions, such as the interface with I/O devices to analyze a data block, but is not aware of the context. Data structures associated to those blocks have to be rebuilt from the virtual memory.

2.2.8.1 *Hardening recommendations*

Recommendations to harden virtual systems are detailed in [208] and summarized in [157] (see Table 10). It is a minimal checklist to respect to ensure a secure cloud computing environment.

The security mechanisms previously seen are mapped to the methodology in Table 11. They are enablers to setup the checklist and enhance the current security level.

Layer	Approaches/Mechanisms				
	Min TCB		Harden TCB	Access control	Attestation
	Disaggregation	**New architecture**			
Secure virtual machine	MicroDrivers [83], Overshadow [45]	Full virtualization, Emulation	GrSecurity [189], OpenWall, WX̂, ASCII Armor, PAX, Exec shield, ASLR, PIE	Terra [86], Proxos [202], SELinux [185], Smack [178], TOMOYO [92], AppArmor [31]	vTPM [159]
Secure management interfaces	cgroups	Entropy	PolicyKit, sVirt [141]	SSH	TLS
Secure hypervisor	Hyperwall [199], DeHype [232], SSC [42], Min-V [144], XOAR [54], Qubes [172], HyperLock [228], HyperSafe [227]	NoHype [200], Barebox [114], MAVMM [145], TrustVisor [135], XMHF [213], Lockdown [212], SecVisor [179], BitVisor [182], sHype [177]	HyperSentry [25], LXFI [133], HyperGuard [175], HyperCheck [225], RandHyp [224]	CloudSec [101]	Nova [205, 206], L4 [120], iKernel [203]
Secure host operating system	Type I hypervisors, CloudVisor [235], Turtles [33]	Micro hypervisors	GrSecurity [189], OpenWall, WX̂, ASCII Armor, PAX, Exec shield, ASLR, PIE	SELinux [185], Smack [178], TOMOYO [92], AppArmor [31], Gradm	
Secure Hardware		MPX [103], SGX [102], NX, SMEP, SMAP	SMM, MultiHype [181], SICE [26]		SMM, TPM, TXT

Table 11: Mapping hypervisor security mechanisms to methodology.

The classification is based on the different features highlighted throughout this chapter.

The new architecture approach to minimize TCB for secure host operating system is empty. This is because the hypervisor and the OS are mixed up by design. New architectures tries to outperform by designing a virtualization-aware OS, and the Secure hypervisor cell is applicable to protect the host OS layer.

2.2.9 *Conclusion*

While providing compromise detection at several levels, there is still low remediation to rollback kernel to a clean state. An promising approach to set up these mechanisms is the autonomic computing paradigm.

2.3 AUTONOMIC COMPUTING

We describe the autonomic approach and define specific terms [113]. Systems following this approach can manage themselves using high-level objectives defined by their administrators.

2.3.1 *Definition*

An autonomic computing system is identified through 8 key elements defined into [112]:

1. To be autonomic, a computing system needs to "know itself" – and comprise components that also possess a system identity.
2. An autonomic computing system must configure itself under varying and unpredictable conditions.
3. An autonomic computing system never settles for the status quo – it always looks for ways to optimize its workings.
4. An autonomic computing system must perform something akin to healing – it must be able to recover from routine and extraordinary events that might cause some of its parts to malfunction.
5. A virtual world is no less dangerous than the physical one, so an autonomic computing system must be an expert in self-protection.
6. An autonomic computing system knows its environments and the context surrounding its activity, and acts accordingly.
7. An autonomic computing system cannot exist in a hermetic environment.
8. Perhaps most critical for the user, an autonomic computing system will anticipate the optimized resources needed while keeping its complexity.

An autonomic computing system features 4 main properties detailed further:

Self-configuring For automatic configuration of infrastructure components.

Self-healing For automatic recovery when system is in a faulty state.

Self-optimizing For automatic adaptation of resources toward better efficiency.

Self-protecting For automatic protection against attacks.

2.3.2 *Self-configuration*

Autonomic systems will configure themselves automatically in accordance with high-level policies – representing business-level objectives, for example – that specify what is desired, not how it is to be accomplished. When a component is introduced, it will incorporate itself seamlessly, and the rest of the system will adapt to its presence – much like a new cell in the body or a new person in a population.

2.3.3 *Self-optimization*

Autonomic systems continually seek ways to improve their operations, identifying and seizing opportunities to make themselves more efficient in performance or cost. Just as muscles become stronger through exercise, and the brain modifies its circuitry during learning, autonomic systems will monitor, experiment with, tune their own parameters and will learn to make appropriate choices about keeping functions or outsourcing them. They will proactively seek to upgrade their function by finding, verifying, and applying the latest updates.

2.3.4 *Self-healing*

Autonomic computing systems will detect, diagnose, and repair localized problems resulting from bugs or failures in software and hardware. Using knowledge about system configuration, a problem-diagnosis component (based on a Bayesian network, for example) analyzes information from log files, possibly supplemented with data from additional monitors that it has requested. The system then matches the diagnosis against known software patches (or alert a human programmer if there are none), installs the appropriate patch, and retests.

2.3.5 *Self-protection*

Autonomic systems will be self-protecting in two senses. They will defend the system as a whole against large-scale, correlated problems arising from malicious attacks or cascading failures that remain uncorrected by self-healing measures. They also anticipate problems based on early reports from sensors and take steps to avoid or mitigate them.

We provide further analyze of self-protection to argument design principles fulfilling. Note that the following sub sections can be related to other aspects of autonomic computing, such as self-configuration with policies, but the self-protection is our main target in this thesis.

2.3.5.1 *Policy frameworks*

Several generic policy management frameworks [58, 210, 72, 93] have been proposed to automate system and network adaptations to respond to context changes. They are generally built around well-documented, extensible information models. A few of them [58] address security for large organizations with traditional IT systems. However, policy-driven security automation remains at a very early stage for the cloud [147, 94] – for instance, by applying ideas from Model-Driven software development for the PaaS layer [147]. Few frameworks address multi-layered defense or multiple loops. A notable exception is the FOCALE autonomic architecture [72, 194], which shares many similarities with our work. FOCALE features a mediation layer with agents which operate by model translation to normalize formats of monitored data or reaction commands. FOCALE also allows multiple loop patterns, but with a more elaborate adaptation model: an outer management loop enables context-aware adaptation of each component of a main control loop. Although FOCALE seemed suitable for Beyond 3G network deployment, it appears nonetheless quite complex to implement, and does not seem to have been applied to the cloud.

2.3.5.2 *Self-protection frameworks*

A number of self-protecting systems [121, 20, 60, 184] have also been investigated, mainly to mitigate network threats[44]. These systems contain both detection and reaction mechanisms. The overall assessment is broadly similar to policy frameworks, although some projects investigated cross-layer security [121], support multiple detection and reaction algorithms [121, 20, 60], or have an explicit policy orientation [184]. Our work complements RootSense which defines layered security architecture. However, while the RootSense design is very much oriented towards detection, reaction

being only the last stage, our solution design is more balanced between detection and reaction, context aggregation and reaction policy refinement being performed symmetrically by dedicated agents. Moreover, RootSense targets single hosts rather than clustered architectures. The self-protection architecture defined in the Selfware project [60] is another closely related work. It targets PaaS rather than IaaS environments, and defines a 2-level architecture – a component-based system representation being used to detect and react over managed resources. While introspection and control interfaces directly derive from the use of a component-based model, the security mechanisms for detection and reaction remain implicit. In our solution, they appear as an explicit security management plane, offering increased flexibility to include third-party security components.

2.3.5.3 *Protecting VMs*

Virtual machine introspection [85] sparked a whole steam of research [143, 107, 170, 100] to use the capabilities of the hypervisor (Virtual Machine Monitor or VMM) to monitor VM behaviors. Different alternatives where proposed to place the monitoring component: embedded in the VM, in the VMM, or in an "out-of-VM" appliance. A few systems attempted to tackle the well-known "semantic" gap issue by comparing monitoring information gathered from different layers [107, 170, 100]. These efforts were mostly focused on detection with almost none, or very simple remediation policies (restart, kill a VM) [100]. Some systems [86] based on a trusted VMM also allowed verification of flexible integrity policies. Overall, the corresponding architectures generally proved difficult to be compatible with legacy anti-malware software. A number of reaction mechanisms were also proposed, mainly in terms of firewalls [177, 171] or self-recovery mechanisms after an intrusion [89]. But few of them have self-protecting features or flexible security policies.

2.3.5.4 *Hypervisor defense*

One layer below, a variety of techniques were proposed to protect the VMM from subversion, with special attention to buggy or malicious device drivers which enable kernel exploitation due to poor confinement. One can mention trusted computing architectures [25] providing strong VMM code integrity guarantees, sandboxing by controlling communications between driver and device [197], kernel [133], or user space [83], virtualization [203, 33] and new component-based VMM security architectures [192, 54] to strengthen isolation, or language-based techniques [227] to detect safety violations. However, solutions remain limited either to pure integrity detection [25, 227] or simple containment [133, 83, 203], proposing no actions to sanitize the kernel. Security policies are also often not well separated from the interception mechanisms themselves, making them hard-to-manage. Security mechanisms usually require extensive code rewriting, making them hard to apply to legacy hypervisors. Overall, VMM self-protection is still a widely unchartered area [222].

2.3.5.5 *Detecting intrusions and malware*

Finally, generic IDPS and anti-malware tools [27, 226] may also be viewed as a form of self-protecting system, with an extensive number of techniques to detect both known and unknown attacks [154]. Those systems are usually based on a single control loop,

Systems	Principle fulfilled?			
	P1	P2	P3	P4
Policy frameworks	yes	no	no	yes
Self-protection systems	no	no	no	some
Protecting VMs	some	yes	no	no
Hypervisor defense	no	a few	no	no
IDPS/anti-malware techniques	no	no	a few	yes

Table 12: Principle coverage by some existing classes of systems .

with a few attempts at cross-layering[226] to detect elusive malwares. Their architecture is usually more open to allow selection and composition of several detection algorithms to improve accuracy. However, in most cases, they have been little applied to the cloud.

2.3.6 *Policy models*

Access Control (AC) models are very widespread, declined in several categories and often referred in terms of subjects and objects. A *subject* is "a computer system entity that can initiate requests to perform an operation or series of operations on objects". An *object* is "an abstract concept that is useful for purposes of generically modelling AC approaches and their mechanisms" [78]. Discretionary Access Control (DAC) manage rights of an object for given subjects. The Lampson model defines a generic access matrix [126] to express security policies, with one row per subject and one column per object. Although the matrix illustrates all possible interactions, only few intersections are populated, and thus inefficient in terms of memory. Actual UNIX AC implements access control lists, capabilities and protection bits to optimize the DAC. These have drawbacks, such as it does not handle data access for particular subjects. The Bell - La Paluda model [127] implements hierarchical access control to meet military classification model with security levels, to bring mandatory access control (MAC). While classic laws are *No Read Up, No Write Down*, the Biba integrity model [36] perform the opposite to ensure no compromise of lower level integrity. These models are simplistic and hard to use effectively on conventional OSes.

Newer AC models are more suitable for OSes. Domain-Type Enforcement (DTE) enhance the TE model with the association of subjects to domains, and objects to types. A domain-domain access control table (DDAT) specifies how domain can interact, while a domain-type access control table (DTAT) defines the allowed interactions between domains and types. They are equivalent with the Lampson generic access matrix, but with a reduced amount of entries by grouping entities. It is used by firewalls [148] and OS [118]. The DTEL high-level language is available to specify DTE security policies. Also, the Role-Based Access Control (RBAC) [77] explores roles and privileges to fit OSes environments. Users are tied to roles, and how which operations are available to the users. Extensions have been proposed to extend RBAC with organisational features [109]. Organization-Based Access Control (OrBAC) is a generic access control model that abstracts subject, action and object with a role, an activity and a view. The role (resp. activity, view) groups subjects (resp. actions, objects) obeying the same security rule. The latter is defined by an organization, which bound policies. The OrBAC

API [16] exposes clean interfaces to create and edit the underlying engine. The MotOrBAC [24] security policy editor 2 is a GUI that allows integrated OrBAC management and specification. Extensions of the OrBAC model address privacy [151], collaborative systems [69] or even critical infrastructures [108].

Our solution have to be policy-driven to simplify administration and leverage management. Thus, we consider the policy component as an adaptable module to plug seamlessly into the security architecture. Our goal is an OrBAC module with the associated API to empower the administration with abstract policies. The separation with the implementation suits our needs to handle mushrooming technologies. Into the cloud context, organisations can be multiple IaaS providers and be the foundation for a multi-provider solution, with a security architecture supporting Amazon, Google and Microsoft.

2.4 CONCLUSION

We have seen cloud computing concepts and approaches related to security enhancement in such environments, through legacy mechanisms and novel approaches, such as the autonomic computing introduced by J. Kephart [113].

3 VIRTUAL ENVIRONMENT SELF-PROTECTION ARCHITECTURE

Learn the rules like a pro, so you can break them like an artist.

- Pablo Picasso

This chapter presents our main contribution, the **Virtual Environment Self-Protection Architecture** (VESPA).

First, we explain the concept of autonomic computing adapted to cloud security and how a self-protection framework can handle issues detailed in Section 2.1. Then we review the different classes of attacks against the cloud in Section 3.2. After what we detail our model through design principles, respecting our objectives defined in Section 1.2, and define a model in Section 3.3. The framework concept refines the model in Section 3.4 and shows how to perform self-protection. Section 3.5 exposes two different implementations of the framework, followed by a straightforward evaluation in Section 3.6.

3.1 CONCEPT

VESPA is an autonomic framework with respect to key elements presented in Section 2.3. With virtualization in mind, we designed an architecture to benefit from the inherent layered virtualization model. From this architecture we built a flexible framework with a hierarchy of components, enabling policy derivation for easy administration. The result is a toolkit empowering IaaS infrastructure supervision to link available security components.

3.2 THREAT MODEL

The IaaS infrastructure attacks are classified in 3 categories detailed further: (1) compute, related to resources such as CPU, RAM or devices; (2) network, related to resources such as virtual switches and network devices; and (3) storage, related to resources such as virtual hard drives and dedicated equipments to gather files.

3.2.1 *Compute threats*

Cloud resources are not spared by virus infection, such as Zeus and other trendy malwares. However, we want to preserve tenant isolation regardless of colocated workloads. Then the VM hardware allocation have to be preserved. A single VM is assigned to a specific number of CPU, RAM and devices. Attacks showed how to overstep those limits and compromise colocated VM confidentiality, integrity and availability.

Furthermore, we have seen how virtualization interferes with conventional memory management to consolidate physical machine workload. Those mechanisms have to be under close surveillance as they add potential traps to the underlying hypervisor.

Indeed some agents inside the VM inform the hypervisor of the memory usage, and create another attack vector for illegitimate access.

3.2.2 *Network threats*

Networking resources allocated in the cloud support a large number of virtual machines with low impact. Thus attackers are using the huge bandwidth to attack other services and threaten availability. Denial of Service is often distributed for even more impact. The control can be intentional, for Crimeware-as-a-Service, or under unexpected control, such as botnets.

3.2.3 *Storage threats*

A central storage usually gather VM virtual hard drives as files on the same physical machine. This is an opportunity for an attacker to gain access to data and perform cold-boot attacks (see Bastion [43]).

3.3 MODEL

We now present our self-protection model. After discussing the VESPA design principles, we describe its layered architecture that features: a security plane containing off-the-shelf components for fine-grained security supervision (monitoring and control) over IaaS resources; an agent-based mediation plane abstracting away security component heterogeneity, and enabling granular levels of security supervision; and an autonomic management plane realizing two levels of self-protection, both within layers and across layers. After a broad architecture overview, we describe the structure of each plane in detail. We then explain how their components can be put together to realize the different self-protection loops.

3.3.1 *Design Principles*

The VESPA design is built on four guiding principles already presented in the introduction 1.3:

Policy-based self-protection. Our solution have to provide effortless administration facilities to be effectively adopted and deployed.

Cross-layer defense. Our solution integrate each of them to leverage defense granularity. Thus, we have communications both inside a layer, and between layers. Such interactions are non trivial as layer semantic usually diverge (i.e. the hypervisor only see CPU instructions, and not processes). Detections and reactions should not be performed within a single software layer, but may also span several layers.

Multiple self-protection loops. Events collected from one layer can trigger reactions on other layers, improving infrastructure security. Thus, intra layer events can create inter layer events. It leverage the flexibility of supervision perimeter. Our solution creates and handles such events to provide a new level of flexibility for administrators.

Open architecture. Multiple detection and reaction strategies and mechanisms - notable heterogeneous off-the-shelf security components - should be easily integrated in the architecture, to mitigate both known and unknown threats.

3.3.2 *VESPA System Architecture*

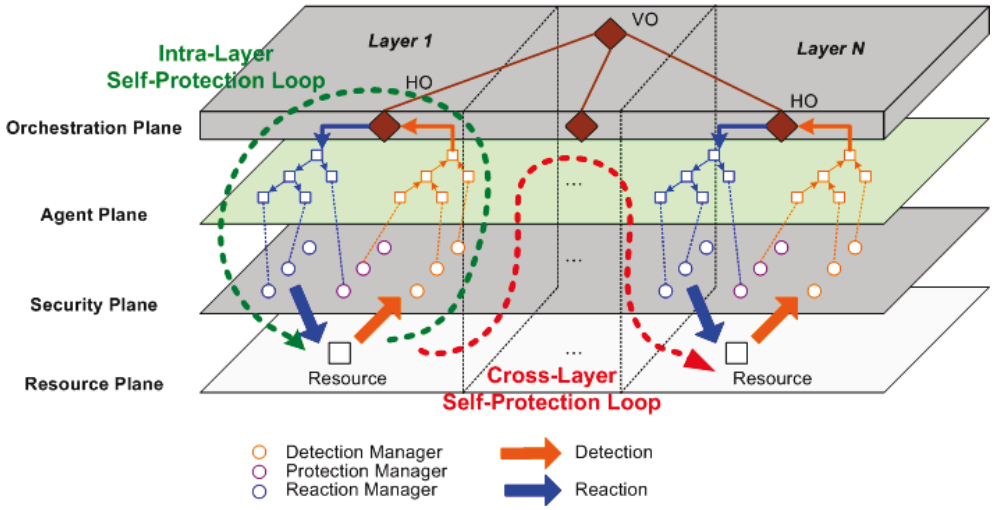


Figure 8: VESPA Self-Protection Architecture.

A IaaS infrastructure groups resources into *layers* according to the virtualization level. VESPA considers security management orthogonally to layers, self-protection being achieved through a set of autonomic loops operating over a number of components organized into four distinct *planes*, as shown in Figure 8.

At the bottom, a *Resource Plane* contains the IaaS *resources* to be monitored and protected, i.e., managed elements. Over it, a *Security plane* contains commodity detection and reaction components that deliver security services such as resource behavior and/or state monitoring (e.g., an IDS component), or reaction and/or resource state and behavior (e.g., a firewall component). These components are the sensors and actuators of traditional autonomic security architectures [47]. Their APIs are typically vendor-specific.

The next plane, the *Agent Plane*, abstracts away security component heterogeneity by defining a mediation layer between the security services and decision-making elements. This plane is built from two hierarchies of *agents*, one for detection, and another for reaction. Agents have two main roles. First, leaf agents are adapters between the VESPA framework and the security components, used to translate vendor-specific APIs into a normalized format both for detection and reaction. Thus, they enable to plug-in third party security components within the framework. Higher-level agents are in charge either of alert correlation or of reaction policy refinement. Thus agents enable a granular level of security supervision over underlying resources.

The topmost plane, the *Orchestration Plane* contains the decision-making logic. It is composed of two types of autonomic managers (called *orchestrators* in VESPA): *Horizontal Orchestrators (HOs)* that perform layer-level security adaptation; and *Vertical Orchestrator (VOs)* in charge of cross-layer security management.

3.3.3 *VESPA Model*

In this section, we now present the design of each VESPA plane in terms of Unified Modeling Language (UML). With such definitions and abstractions, developers are able to derive the model to many languages fitting their needs.

3.3.3.1 *Resource Model*

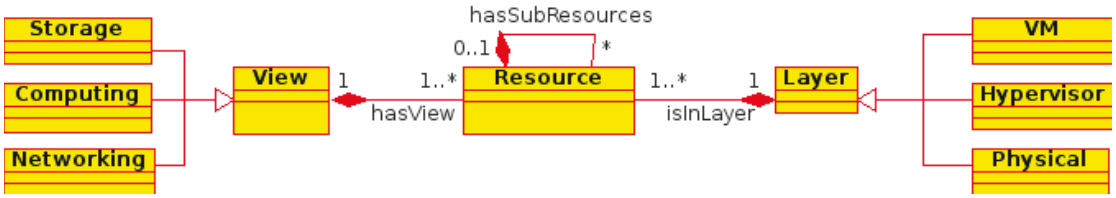


Figure 9: Resource Model.

IaaS *resources* are categorized according to two orthogonal criteria (see Figure 9). A *layer* defines the location of the resource in a cloud stack. Current IaaS stacks are built as a *physical machine* running an *hypervisor*, in turn executing *virtual machines (VMs)*. Three separate layers are thus clearly identified. The *view* abstraction captures a broad class of resources: *computing*, *networking* or *storage*.

In a typical IaaS stack, the interplay between layers and views may be summarized as follows. The physical layer provides raw computing, communication, or storage facilities to other infrastructure components. Typical members of each view are respectively: CPU, memory, and graphic cards; commodity network equipments and interconnects; and storage devices connected to the network or to a PCI slot. Above, the hypervisor multiplexes and isolates physical resources providing them as a virtualized device abstraction to VMs. View members then become respectively: hypervisor virtual CPU, memories and devices; virtualized network equipments such as routers, switches and firewalls; and virtualized storage accessible as dedicated devices. At the top, VMs have their own resources just like any operating system (guest OS), relying on the hypervisor for device emulation or access.

3.3.3.2 *Security Model*

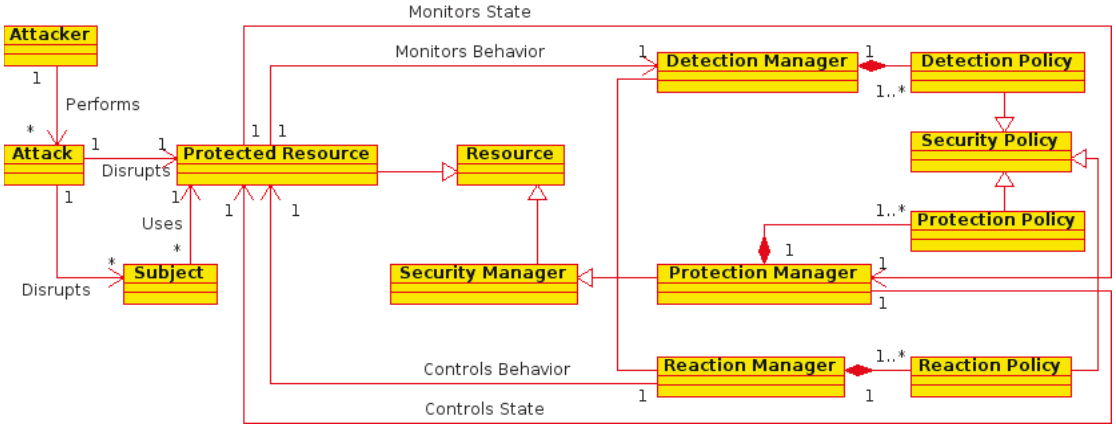


Figure 10: Security Model.

VESPA protects the critical assets of the infrastructure against attacks, called *Protected Resources (PR)* (see Figure 10). Attacks may corrupt a PR, or disrupt the subject which is using it.

In VESPA, some of the main considered threats target *the VM layer*: a malicious VM fools the IaaS VM placement strategy to become co-located on the same physical server as the target VM. A side-channel attack breaking VMM isolation may then be used to steal / corrupt information from the infiltrated VM. Another variant may be to contaminate the VM guest OS, e.g., with a virus spreading through network, IPCs, or file system. Results can range from unexpected network traffic, arbitrary code execution in user or kernel mode, to privilege escalation. Traditional network security threats are also to be considered between VMs, e.g., traffic snooping, VM MAC/IP address spoofing, or VLAN hopping.

However, more potents attacks on the hypervisor layer are also relevant. A VM escapes from hypervisor isolation enforcement to take full control of the virtualization layer. Possible attack vectors include misconfigurations, or malicious/poorly confined device drivers in the hypervisor. Possible next steps include compromising hypervisor integrity, installing rootkits, or launching an attack against another VM.

Attacks against the physical layer such as DMA attacks on device, compromise of the SMM CPU mode, or traditional threats on the physical network are also considered for protection. In VESPA, PRs are under the supervision of a *Security Manager (SM)*. This means: (1) monitoring resource behavior through a *Detection Manager (DM)*, e.g., an Intrusion Detection System (IDS); (2) modifying resource behavior through a *Reaction Manager (RM)*, e.g., a firewall; or (3) monitoring and modifying the resource internal state with a *Protection Manager (PM)*, e.g., a file system integrity and intrusion recovery manager [89]. Those security components are typically off-the shelf, accessible only via vendor-specific security APIs. The behaviors of all SMs are governed by *security policies*.

3.3.3.3 *Agent Model*

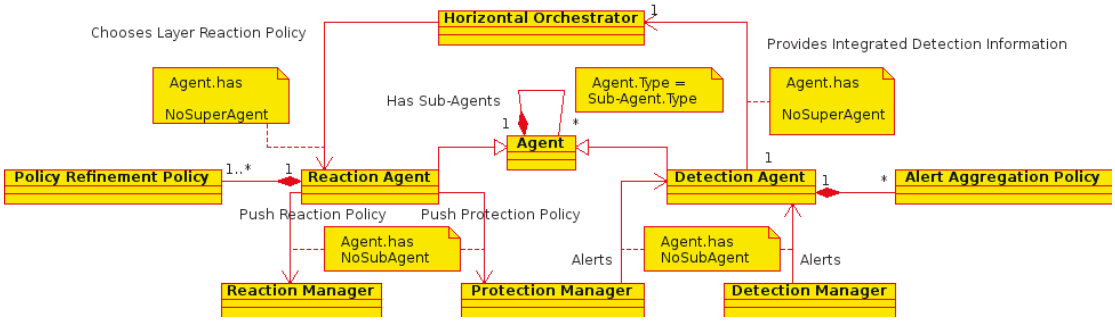


Figure 11: Agent Model.

The agent plane plays the role of a mediation layer between off-the-shelf security components (SMs) in the security plane and decision-making in the orchestration plane, both for detection and reaction phases (see Figure 11). Agents perform security context aggregation from low-level security events gathered from detection mechanisms located in the security plane to a high-level risk assessment able to guide the decision process. Conversely, they also realize reaction policy refinement from the high-level response chosen after the decision to low-level policies which can be enforced by the reaction mechanisms of the security plane. Agents are thus naturally

organized in a hierarchical structure, root agents (resp. leaf agents) capturing high-level (resp. low-level) analysis and response. Two separate agent hierarchies are defined, one for detection, and another for reaction. Agent behaviors are governed by transformation policies, both for alert correlation and reaction policy refinement.

In line with the open architecture principle (P4), the agent plane also aims to enable to plug-in third-party detection or reaction components within the VESPA framework. Leaf agents may be viewed as resource API adapters between VESPA and such components: they perform the translation between vendor-specific interfaces of external security components and a normalized representation for detected events and response actions.

Detection is performed as follows: a DM or PM notifies its associated *Detection Agent (DA)* of security-sensitive events. Each DA then applies an *Alert Aggregation Policy* to correlate collected information from sub-agents before sending them to its parent agent. When reaching the root detection agent, security context information is transmitted to the orchestration plane through the *Horizontal Orchestrator (HO)*.

The reaction process is symmetric: after choosing to enforce a layer-specific reaction policy, the HO, sends that policy to the root *Reaction Agent (RA)*. Each RA will apply a *Policy Refinement Policy* to fine-tune the chosen adapted response, before sending it to chosen sub-agents for enforcement. When reaching a leaf reaction agent, reaction and/or protection policies are pushed towards the corresponding managers in the security plane.

3.3.3.4 *Orchestration Model*

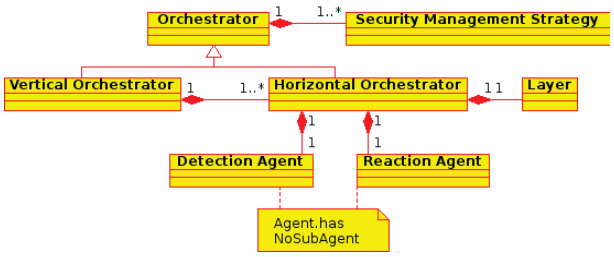


Figure 12: Orchestration Model.

The decision-making logic is contained in the orchestration plane, and is split between two types of *orchestrators*, as shown in Figure 12. Each IaaS layer contains a *Horizontal Orchestrator (HO)* providing a layer view of security management. The HO is a simple autonomic security manager performing a reflex, local response to threats targeted at layer resources. The HO gathers the overall layer security context information from the root DA. The HO *Security Management Strategy* allows it to choose the best layer-level reaction policy, which is then dispatched to the root RA for enforcement.

The HO may also apply decisions coming from a *Vertical Orchestrator (VO)*, an overall autonomic manager that realizes higher-level, wider spectrum security reactions. The VO coordinates layer-level decisions to provide a consistent, cross-layer response to detected threats. Based upon layer-level information collected from the relevant HOs, the VO builds a high-level knowledge base of overall infrastructure resource states and alerts. The VO *Security Management Strategy* contains administrator-defined policies on alert feedbacks to trigger or not a cross-layer response. It also allows the

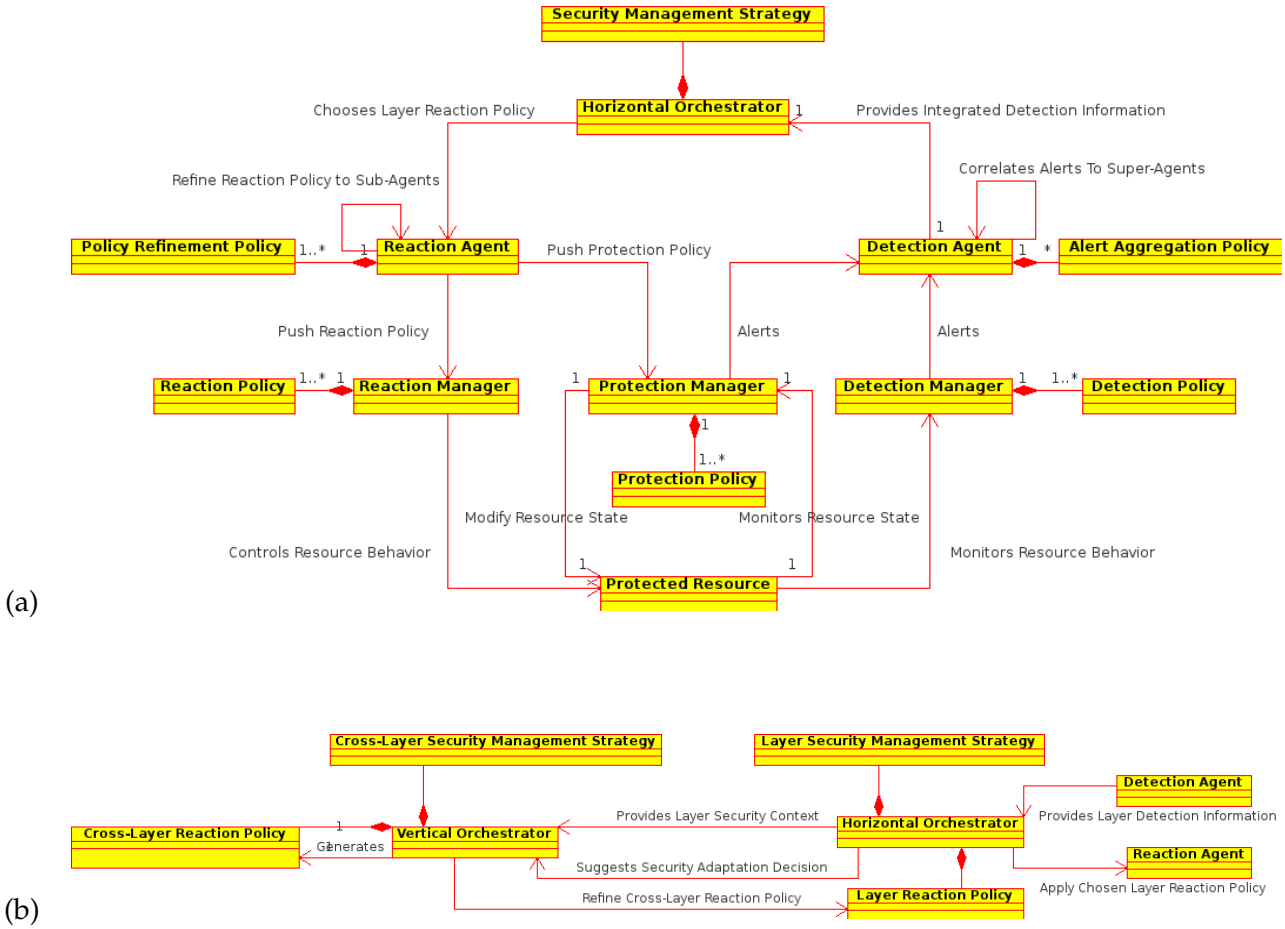


Figure 13: (a) Intra-Layer Loop; (b) Cross-Layer Loop .

VO to chooses the overall reaction policy, which is then pushed down to the relevant layers for enforcement by the corresponding HOs.

3.3.3.5 *Policy Model*

Several types of policies are used in VESPA. *Monitoring policies* define how to collect, filter, and correlate alerts from sensors. *Refinement policies* express how to narrow a generic wide-scope response down to a small set of actions understandable by a specific reaction mechanism. *Security management strategies* govern decision-making, defining which reaction policy to apply or generate in a given security context. Finally, *reaction policies* specify how to modify the behavior or state of the resource accordingly.

Our framework considers policies as a strong asset. However, a wide range of policy models and languages have been proposed as for instance for decision [111] or reaction [59]. Choosing a policy model which is too specific could prove too restrictive, severely limiting framework applicability and extensibility, especially for cloud environments, highly heterogeneous by definition. Therefore, instead of developing a new policy model, we chose to base VESPA policies on an existing model such as [72] handling a large panel of policies, notably ECA rules.

3.3.4 *A Two-Level Self-Protection Approach*

The VESPA design offers the possibility of composing multiple self-protection loops. The previous sections described separately the necessary building blocks and interfaces. We now describe how to put them together to realize two levels of self-protection for a IaaS infrastructure: (1) at layer-level; and (2) cross-layer.

3.3.4.1 *Intra-Layer Self-Protection*

The layer-level loop works as shown in Figure 13a. The behavior and state of each PR can be monitored and modified. When a threat is detected by a DM and/or PM, the associated agent is directly informed. Collected monitoring information is then aggregated and correlated according to defined detection policies and transmitted to super-agents. The process is repeated until reaching the root agent. The overall security context information is then transmitted to the orchestration plane, where the HO takes a security adaptation decision, and selects a reaction policy sent back to the agent plane for enforcement. Reaction policies are refined going down the agent hierarchy until reaching the leaf agents, which push policies to RMs and/or PMs, modifying behavior and resource internal state. Note that this type of loop realizes a reflex response, decision remaining lightweight to cut down the overall reaction time.

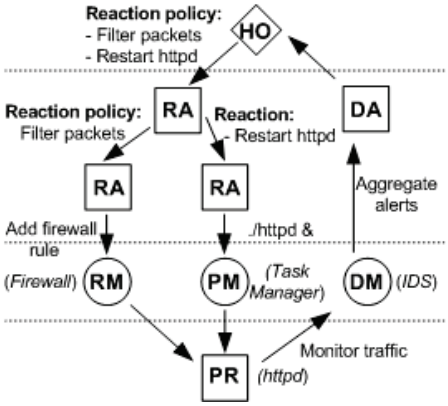


Figure 14: Web Server Flooding Self-Protection.

EXAMPLE To illustrate, we consider as PR, a VM-hosted Web server to be protected against flooding attacks. Some relevant security mechanisms might be an IDS (DM) for traffic monitoring, a firewall (RM) such as iptables for malicious packet blocking, and the guest OS task manager (PM) for restarting the Web server. Figure 14 shows the corresponding VESPA architecture with relevant agents (DA and RAs). Upon attack detection by the IDS, the DA forwards the alert to the HO that chooses a two-level compound response: (1) block attacker packets; and (2) restart the Web server. The reaction policy is then refined by the RA stack into simple reactions enforced by the relevant RM/PM: (1) ban the attacker IP address by adding a new iptable rule (for the RM); and (2) restart the httpd process (for the PM).

3.3.4.2 *Cross-Layer Self-Protection*

The real potential of the framework comes from cross-layer response (Figure 13b): layer-specific security context is sent by HOs to the VO. That orchestrator consolidates

system knowledge, and applies the cross-layer *Security Management Strategy* to generate an overall response, then refined into reaction policies for each concerned layer, subsequently enforced by the corresponding HOs, as for the intra-layer case.

In the previous example, all packets are routed to the whole physical infrastructure and hypervisor, squandering bandwidth. If a physical firewall is added upstream, another policy may be sent to the reaction agents chain of physical layer. Therefore, all packets coming from the banned IP addressed are dropped earlier in the network, saving precious bandwidth. Moreover, the Web server VM is doubly protected (defense-in-depth approach): if physical firewall is evaded, the iptables can take over.

Overall, VESPA defines an architecture where the number of layers multiply possibilities in terms of remediation actions, either locally or at infrastructure-level. Its agent-based architecture also enables to combine network and system infrastructure views both in detection and reactions phases.

3.4 FRAMEWORK

We now describe the framework refining the previous model to implement the solution. This section discusses important implementation points to consider while instantiating the VESPA model.

3.4.1 *Core Framework*

VESPA relies on a core framework built on a distributed object model. The basic elements of the core framework are *objects*, that capture functionalities needed by all framework entities. Objects are named using unique object identifiers (OIDs), and have the ability to spawn new objects. This feature is used to build the agent and orchestrator hierarchies defined in the VESPA model. Spawned objects are tagged as *slave* in the spawning object. Upon object creation, an initialization method is called specifying: the identifiers of the object and of its parent, and a list of slaves to create. Finally, the object listens for incoming communications.

Objects interact through an Object Request Broker (ORB) enabling distributed communications. Messages are formatted as:

```
function|argument1#...#argumentN
```

Message sending is based on the `send(OID, message)` interface. Identification of interacting parties and communication channel establishment is completely handled internally. Message reception is based on the `receive(message)` interface. The message arguments are then passed to the relevant available object functions.

The core object framework only provides basic primitives. To implement the VESPA model, the core framework must be extended, objects being specialized with their own specific functions. This may be realized by defining new callbacks to create new framework entities refining the concepts of the model. Our approach is based on agent-oriented programming [183] to specify interfaces and messaging capabilities.

This approach notably enables the flexibility needed to specify the various types of policies found in the self-protection model, as shown next.

3.4.2 *Architecture definition*

Our VESPA solution needs the infrastructure components to deploy the security layer, agent layer and orchestration layer. First the administrator defines execution environments available at the VM-layer, hypervisor-layer and physical layer, as illustrated into the configuration file below 3.4.2.

```
[Livebox3]
Type= Machine
Interfaces= 192.168.1.1

[pc-isaiah]
Type= Machine
Interfaces= 192.168.1.10

[perso-isaiah]
Type= VM
Interfaces= 192.168.1.21

[pro-isaiah]
Type= VM
Interfaces= 192.168.1.20
```

We have 4 execution environments contained into the physical and VM layers, with a unique name and bound to a unique interface. This text-style definition allows the architecture file to be portable: it can be parsed by the most programming languages.

3.4.3 *Implementation considerations*

3.4.3.1 *Resource and Security Planes*

Resources are viewed as black-box entities by the framework. Similarly, security plane components are off-the-shelf, and thus considered "as is". They can only be accessed through vendor-specific APIs to send alerts from PRs to SMs, and to apply commands to modify PR behavior or internal state. SMs are directly connected to framework agents to translate their own APIs to VESPA APIs.

3.4.3.2 *Agent Plane*

In the agent layer, the core object framework is extended differently according to whether agents talk to a HO, to other agents, or to a specific SM. The detection and reaction agent hierarchy is built from root agents that recursively create slave objects. Specific functions are defined to implement several agent-related functionalities defined in the model: (1) applying the AlertAggregationPolicy to received alerts; (2) refining reaction policies according to the PolicyRefinementPolicy.

The AlertAggregationPolicy of the model is enforced in the agent function `alert_handling(alert)`. This callback is called whenever a slave object sends a message containing an alert. Several behaviors are possible such as raw forwarding of the alert to the parent object, or correlating alerts before notifying the parent.

Whenever an agent receives a reaction policy from its parent, the PolicyRefinementPolicy described in the `policy_refinement_policy` agent function is applied. Two situations are then possible for enforcement. The agent can interact with a SM or with

another agent. Interactions with the security plane are completely dependent on the commodity API of its components. Therefore, there is a one-to-one mapping between SM APIs and leaf agent function callbacks. Interactions with other agents grant the ability of callbacks with extended scope. The policy is then refined into sub-policies to be enforced by slave objects.

3.4.3.3 *Orchestration Plane*

HOs create root detection and reaction agents in each layer. Similarly, the VO creates HOs for each layer defined in the VESPA model. Alert handling is similar to the agent case, aggregating and correlating received alerts and forwarding them to the VO. Similarly, a HO may refine the reaction policy chosen by a VO. For both types of orchestrators, decision-making is another function callback registered as a handler, where the HO/VO Security Management Strategy will be enforced.

3.5 IMPLEMENTATION

We highlight the VESPA benefits through 2 different implementations. The first in Python provides an easy-to-use framework with a very specific usage of dynamic code introspection of the language, to deliver obvious code development for agents and policies. The second in C with Fractal provides a faster implementation aimed towards embedded devices. Development is more error-prone, but come with a fine-grained control.

3.5.1 *VESPA in Python*

A first version of the framework, to demonstrate usability and viability, was implemented in Python. This object oriented language allows fast development at the price of a slower execution time.

3.5.1.1 *Code architecture*

The global architecture for a VESPA instance is detailed in Figure 15. We see the Model View Controller (MVC) components playing together with lightweight interfaces. The model class is a node and is a reference toward the VO for further interactions. The HO abstract class generalizes HOs specific to layers. An agent is displayed to highlight how it is connected with other classes.

3.5.1.2 *Core class*

All VESPA components extend common Node object for communications, node management and threading capabilities. Figure 16 details the core class diagram.

The Node class implements the PThread interface which extends the Thread class. Indeed, each node is independent and is not impacted by exception and other errors of colocated nodes.

The PThread interface identifies the component with the desc() method, and returns a tuple (name, ip, port) aimed to be unique among the framework instance. When a PThread is started, the thread register itself with its master by calling remotely the register() function (communications are described in the next paragraph). The master

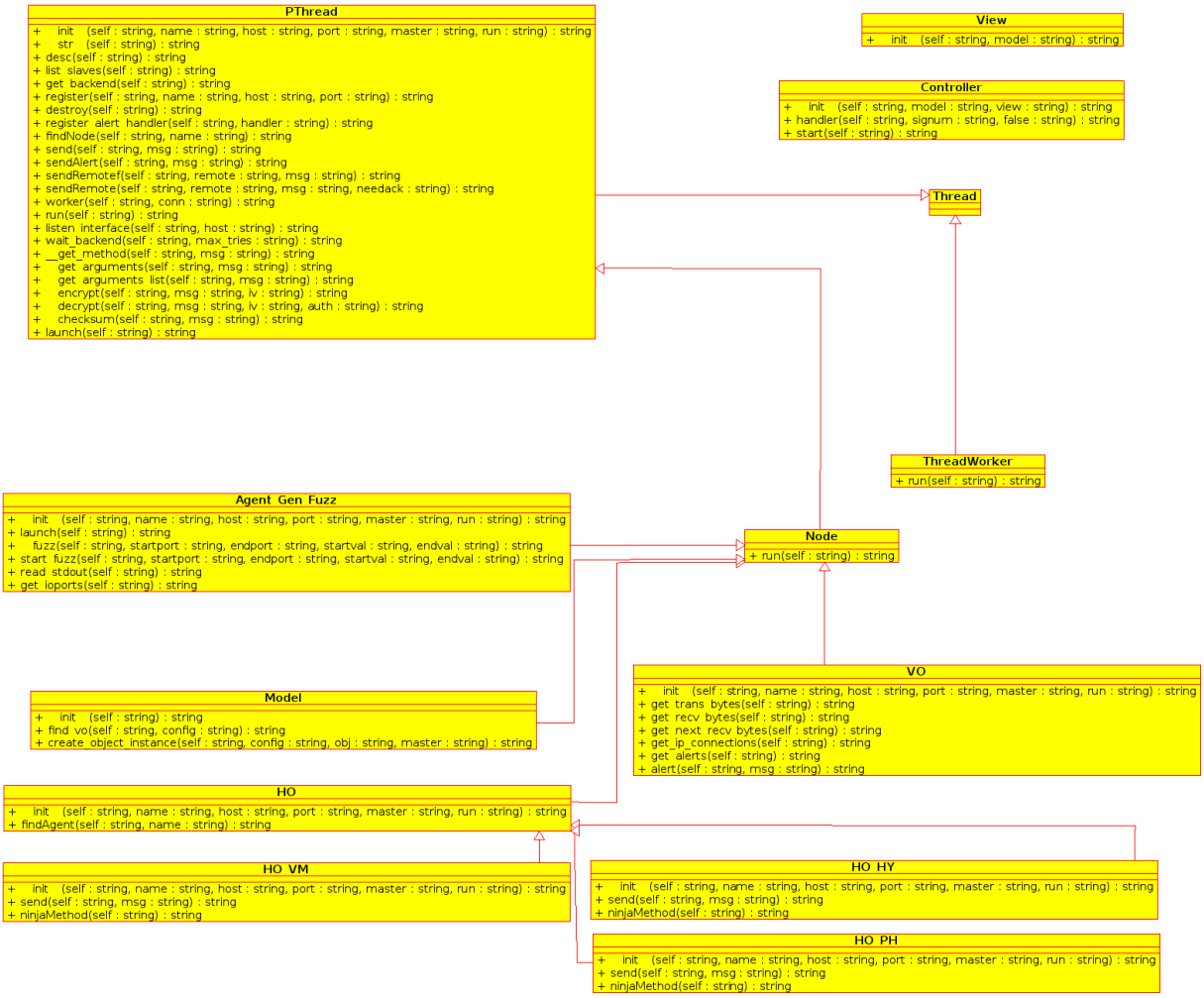


Figure 15: Python Object Hierarchy.

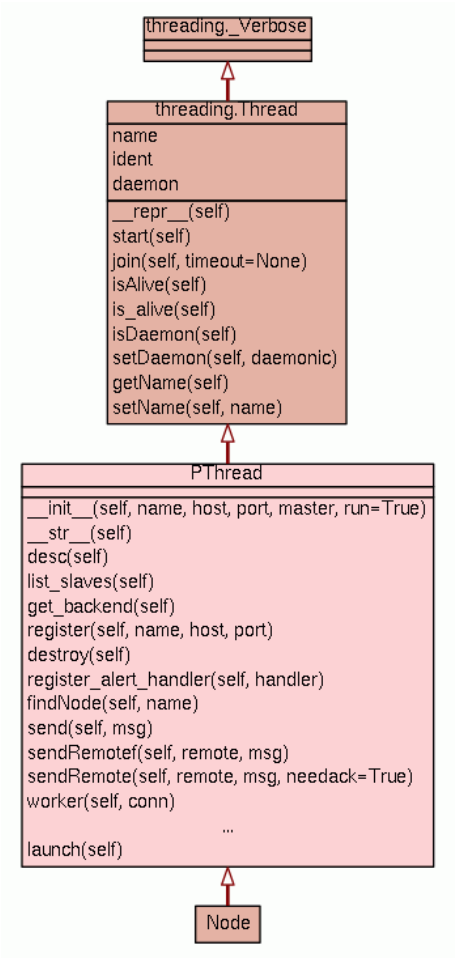


Figure 16: Node core UML diagram.

node then enlist the node as a slave, the list being available with the list_slaves() function.

COMMUNICATIONS Every node listens on the port defined into the configuration file. When another node communicate with our node, the listen_interface create a worker in another thread to handle the request. The socket is then saved and associated to this worker in order to speed up next communication. A pipe now exist between the nodes.

The node communicates through the sendRemote interface. This function ensures acknowledgements, forwarding to the correct next node and result serialization. When the message specifies a function node to call with arguments, the node perform introspection on the current agent and compare the expected function with available functions. If there is a match, the worker converts the message string to function call with arguments, and sends the result to the callee. Otherwise, the refinement policy defines how to forward the message.

Alerts are special messages with specific behavior. They are only sent to their master without acknowledgement for fast processing. In their simplest form, alerts contains the source of the alert and the message. Intermediate nodes, such as HOs and Agents, may aggregates alerts to lighten network load. A special alert handler is defined into the class specifying the node, notably the VO. The alert handlers are fired on the alert message reception, and can be registered through the register_alert_handler function.

Every node is able to query the underlying slaves with the findNode() function. It returns the identification tuple if present and None otherwise. This mechanism enables deployment checking and correct order delivery.

The communication protocol carry out coherent message delivery with a flag terminating transmission. It is needed to concatenate multiple transmissions in the same socket.send(). Unfortunately, the Python socket class is not thread-safe and we had to overload the lowest-level Python socket function and manually schedule messages. Thus, a small overhead enables correct message delivery.

Communications are encrypted and signed with various algorithm (even the NULL algorithm for debugging purposes). It protect the framework against Man-In-The-Middle (MITM) attacks and users trying to send remote orders. The key is distributed during framework deployment, but the framework can handle Public-Key Infrastructure (PKI) or Cryptographically Generated Addresses (CGA) [55].

NODE MANAGEMENT A node obeys to a master, and is the master of slaves. At node initialization, a register message is sent to the master which append the node to its slaves for further reference. The master also forwards the registration to its master for overall knowledge.

The nodes are split in two groups: (1) passive and (2) active. (1) Passive nodes only listen for incoming communication as seen in the previous paragraph. This behavior is intended for reaction agents without detection, as they are only calling functions and not autonomous. (2) Active nodes have the same features plus another thread started at the node creation for autonomous detection. Thus they are able to send alerts on events. For example, an agent watching logs wait for bytes to be written on the file descriptor. When the buffer is full it sends the whole strings as an alert to the master. The agent group is selected during node specialization into the agent class with the run variable.

The *destroy* message cleans the framework node by flooding all neighbors, the master and slaves. Hence, all nodes join their threads, close sockets and die.

MULTI-THREADING A node is divided into multiple threads for more reactivity. A global thread for VESPA initialization, a communication thread with several workers, and a thread for background detection. However the Python multi-threading is under the control of a Global Interpreter Lock (GIL) and restrict true threading to a safer alternative. It is hard to achieve multi-threading without external libraries, but enable easier variables sharing. Semaphores are somehow hidden. Thus users may benefits from this behavior to integrate new threads for further computation by adding the thread to the object thread list.

3.5.2 *VESPA in C*

To enhance the Python version, we adapted the first version to components and thus enable a more flexible architecture. The Fractal derivative in C, Cecilia, was our choice to implement the second version (see Figure 17). The embedded aspect is also our aim to integrate new devices in the VESPA architecture, such as mobile phones, with an optimized assembly and near native speed.

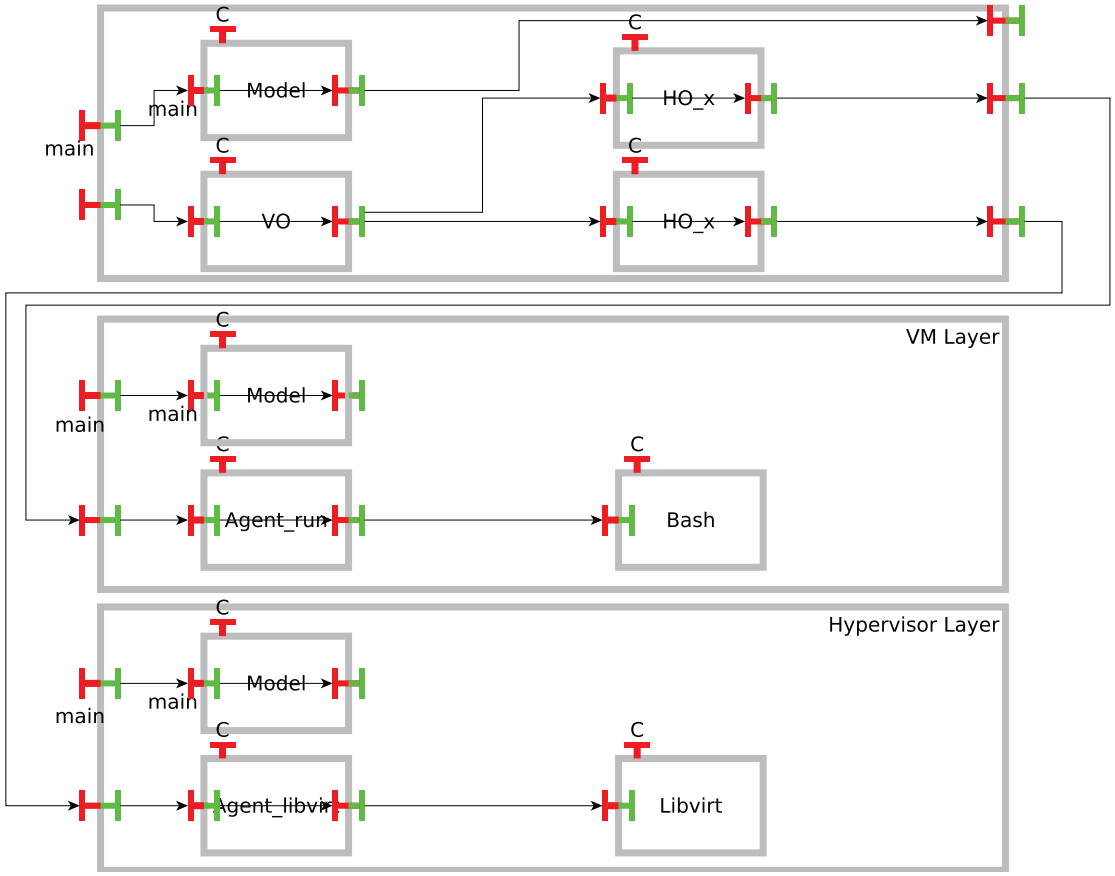


Figure 17: C Object Hierarchy.

3.5.2.1 *Code architecture*

3.6 EVALUATION

This section gives preliminary results regarding the principles detailed in the Chapter 1.3 for VESPA. We introduce several agents to underline the functions included into the framework (P4), then we illustrate the potential of multiple loops for enhanced antivirus detection (P1, P3).

3.6.1 *Available agents*

3.6.1.1 *Log agent*

The log agent wraps logging facilities for a system. The attributes define how logs are gathered: written into specific files or gathered by a more efficient software (such as SystemTap or lttng). This is a generic agent that provides a READ access to files in the current execution environment.

3.6.1.2 *Libvirt agent*

The libvirt agent orchestrates virtual machines for an hypervisor. It can start, stop, kill and migrate VMs with a one-to-one abstraction of the original APIs. This is a generic agent compatible with commonly-deployed hypervisor, and should be the first choice when dealing with VM management.

3.6.1.3 *Shared object agent*

The shared object agent is able to call native libraries in a C-like format. For instance, to reuse libraries available by default on the OS, to offload critical code to C/ASM and call it from an higher-level language (Python) or to aim at embedded deployment.

3.6.1.4 *Run agent*

The generic run agent allows code execution on the execution environment. For example, the agent can wrap bash to run shell scripts for usual administration or the default loader to read executables. The agent provides the EXEC access to the current execution environment.

3.6.1.5 *ClamAV agent*

This agent is an example to wrap the ClamAV antivirus service. Mixing kernel mode driver and user land engine, it covers numerous use cases from least privilege principle to secure I/O communications.

3.6.1.6 *Web agent*

The web agent is specific to Apache for our purpose but can be easily extended thanks to the modular architecture. Only the lower level components have to be replaced. It delivers a wrapper to control the service: start, stop, kill, and reconfigure.

3.6.1.7 *Meeting agent*

The meeting agent is even more specific with a wrapper around the OpenMeeting software. The service can be started, stopped or killed. It is a nice example to see how to wrap the run agent described above and provide simple, clear and effective functions.

3.6.1.8 *iPad agent*

The iPad agent control the iPad display through a VNC interface. This example show how to manage third-party devices with very restrictive access and almost unmanageable.

3.6.1.9 *Proc agent*

This agent supersedes the run log agent with a specific interface wrapping the /proc filesystem on Linux. It enables easy access to the filesystem tree and leafs, and comes handy when dealing with various OSes as a common base. For example the ioports, cpu and memory information give information about how VMs are using their allocated resources.

3.6.1.10 *SDN agents*

The OpenFlow, Floodlight and POX agents interconnect VESPA with the respective SDN controllers. Several alternative are available to fit with current needs: REST, HTTP or TCP levels.

AGENT	ROLE	LINES OF CODE
agent_log	Generic	50
agent_libvirt	Hypervisor / Generic	200
agent_gen_fuzz	Generic shared object wrapper	100
agent_run	Generic	50
agent_clamav	Specific / ClamAV	500
agent_web	Specific / Apache	60
agent_reunion	Specific / OpenMeeting	40
agent_ipad	Specific / IPad	50
agent_proc	Specific / Linux /proc	150
agent_openflow	Specific / OpenFlow	300
agent_floodlight	Specific / Floodlight	80
agent_pox	Specific / POX	120

Table 13: Available VESPA agents.

3.6.1.11 *Conclusion*

We developed multiple kind of agents at the different layers of VESPA to fulfill use cases and shows the flexibility of the framework. Some classes of agents are generic, such as following logs and executing commands through SSH. The available agents in VESPA are listed in Table 13.

We believe that those agents handles the majority of usual requirements. More specific agents provide generic programing, such as the libvirt agent taking advantage of the integrated Python interface to abstract interactions with common hypervisors.

Finally, leaf agents are specific to technologies with Apache web server management and log, Software-Defined Networking (SDN) controller with OpenFlow, or the ClamAV antivirus.

3.6.2 *Correlation*

This section evaluates how commodity systems may benefit from the autonomic approach perform.

3.6.2.1 *Detection*

To benchmark improvements of the detection phase, we measured detection incrementally. We start with one detection agent, then two, three and so on. Here we test with anti-viruses to improve virus file detection with a low false positive (FP) rate.

LIST OF ANTI VIRUSES We used the trial version of anti viruses shown in Table 14 to perform our evaluation. The antivirus with 0 infection mean that we were not able to connect it to VESPA. Those anti-viruses represent the majority of protection tools

ANTIVIRUS	INFECTIONS FOUND	FP
Avast Antivirus 7	22398	2%
Avira Free 2013	12731	1%
AVG Antivirus 2013	20310	2%
BitDefender Antivirus Plus 2013	24519	6%
ESET NOD32 Antivirus 6	17933	2%
F-Secure Antivirus 2013	0	0%
GDATA Antivirus 2013	0	0%
Kaspersky Antivirus 2013	23158	8%
McAfee Antivirus Plus 2013	19002	3%
Norton Antivirus 2013	16328	2%
Panda Cloud Antivirus 2.1	0	0%
Windows Defender (MSE 4)	13711	1%

Table 14: Antiviruses detection.

deployed on average computers. The methodology followed to gather detection rate and false positive is detailed in the following paragraphs.

LIST OF VIRUSES We took the contagio virus samples[1] with separate malicious and clean files to facilitate false-positive and false-negative detection. Our archive contains 16,800 clean files and 11,960 malicious files. Each file may contains multiple objects, usually the PDF with embedded fonts, images and metadata, resulting in infection rate higher than the number of files.

RESULTS Each anti-virus is tested at a time to compare they very own detection performance. The experiment is done on a fresh Windows VM. The virus database is transferred on the machine and decompressed, then we install the latest version of the anti-virus. We finally scan the entire directory containing malicious files and grab the final report. We consider that a single file may contain multiple objects, however some anti-viruses only flagged the file once. We filtered the reports to extract all information for a given file. Indeed, some AVs emit numerous alerts for a single file, while others keep the information more hidden. Thus, we tried to evaluate AVs in equal terms.

While detecting a large portion of viruses, none of them was able to detect all malicious file with no false-positive (Figure 18). BitDefender showed the best results in terms of detection rate, while ESET and AVG have the lowest FP for a given detection rate.

We then interconnect AVs through VESPA. Each AV sends the report back to our VO, which decides if a thorough analysis is needed. In this example we force the

1 Available on http://contagiodump.blogspot.fr/2013/03/16800-clean-and-11960-malicious-files.html

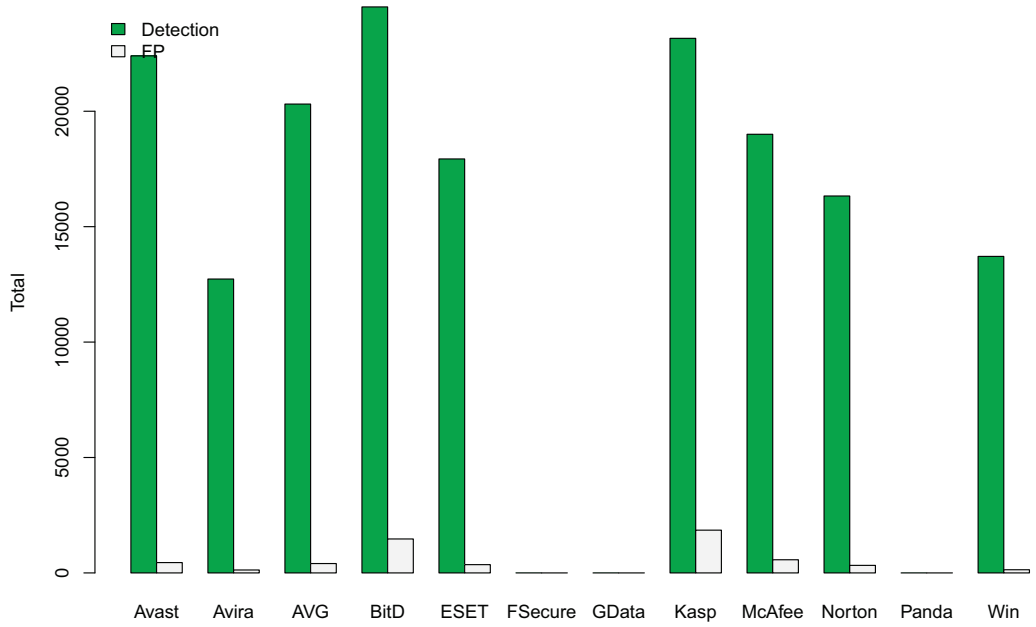


Figure 18: Comparison of antiviruses detection and FP.

VO to perform two and three passes using the AVs with the best results in terms of detection rate and FP. The results are displayed as a Venn diagram on Figure 19.

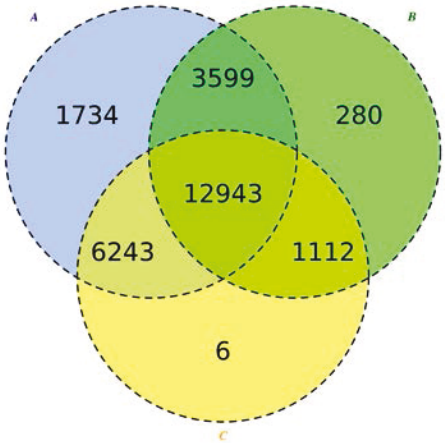


Figure 19: Multi-Antiviruses detection: (A) BitDefender, (B) ESET and (C) AVG Antivirus 2013.

We combine the most effective tools to improve their reliability, as shown on Table 15. The number of detected files is better than all AVs, while giving low FP. Those results highlight how to combine detection agent and enhance system security through a straightforward example. The question of time analysis is voluntary left unanswered, and is more detailed in the next chapter with more complex examples. For now we can say that VESPA is able to perform analysis simultaneously and infer if a file is malicious or not in a given time.

ANTIVIRUS	INFECTIONS FOUND	FP
BitDefender (A)	24519	6%
+ ESET (B)	25911	2%
+ AVG Antivirus 2013 (C)	25917	2%

Table 15: Collaborative antiviruses detection.

3.7 CONCLUSION

We presented our VESPA framework to build autonomic framework on IaaS infrastructure.

The VESPA model is independent from the systems, the programmatic language or the deployment. The node hierarchy abstracts numerous components and dissociates programmers duties. In an ideal world, the software developers provides the VESPA agent wrapping the original API. However we simplified the interface to the maximum for faster development and easier debugging. With a clear model and implementation in multiple languages, we expect that future cloud administrators and developers will adopt VESPA.

The following chapters use the VESPA architecture as the building block to derive specialized use cases on heterogeneous platforms.

4

USE CASES

This chapter presents the instantiation of our VESPA framework through 3 different use cases. First, we use VESPA to detect and react dynamically to a virus infection with available cloud resources in Section 4.1. The use case presented in the previous chapter is extended to a real scenario. Second, Section 4.2 details how to use VESPA to achieve multiple IaaS security level negotiation and reaction in a mobile cloud setting. Finally, we used the framework to benchmark messaging capabilities and performance overhead into Section 4.3. The use case twists normal usage to perform offensive testing, also named fuzzing, against the hypervisor.

4.1 DYNAMIC CONFINEMENT

This section presents the dynamic confinement of a virus spreading through the network, particularly with video streaming. This scenario reflects an everyday situation and provides a real usage.

4.1.1 *Threat Model*

In this scenario, we consider an attacker sending a virus to the user of a VM. The user is not administrator and cannot gain access to the kernel, or the anti-virus. The user is free to install and run programs. VM kernel, hypervisor and physical server are trusted without direct access from the outside. This is a practical scenario where the administrative interfaces of the infrastructure are never open for external communications.

4.1.2 *Scenario and attack*

A typical example that underlines the framework interest is shown in Figure 20:

1. A User VM (UVM) detects the presence of a virus that can compromise nearby VMs.
2. An alert message is sent to the VM-layer HO.
3. The HO sends back to the UVM an "about to disconnect" event.
4. The HO chooses to isolate the compromised VM in both networking and computing views, by cutting the network link `br0` in the underlying hypervisor.
5. The HO also decides to migrate VM ressources on a physically separated computer specially designed for this purpose.
6. Then, once isolation is completely achieved, the VM-layer HO sends an order to trigger the cleaning of the user VM.

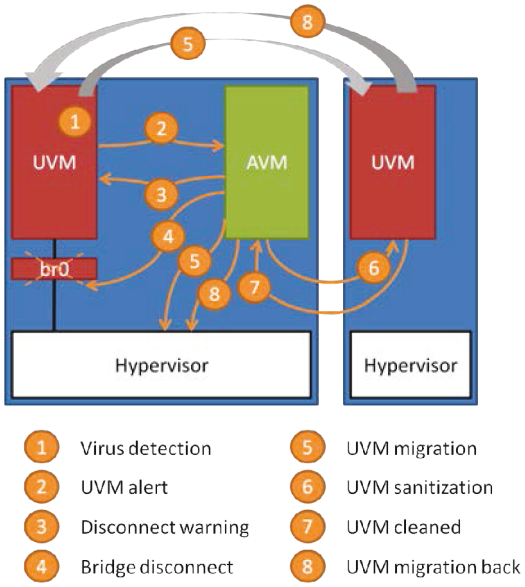


Figure 20: A Simple Use Case.

7. The sanitization status of the UVM is sent back to the HO upon cleaning completion.

8. This allows the hypervisor-layer HO to reassign original VM resources, and migrate the VM back outside the quarantine zone.

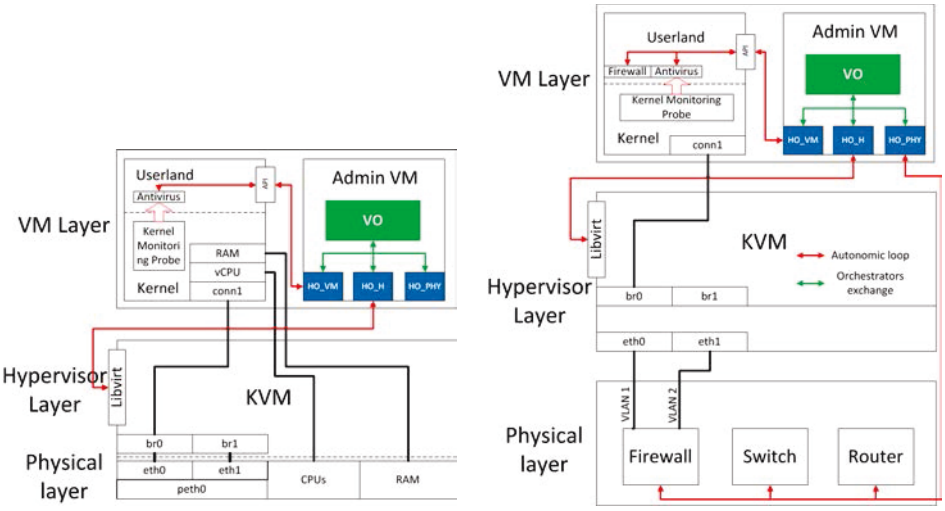
This simple example shows how to perform active defense on such an architecture.

4.1.3 *VESPA Framework Instantiation for IaaS*

Figure 21 represents a simple implementation of the isolation framework on a typical IaaS infrastructure. Dedicated network equipments provide the *physical architecture*. Network traffic is segregated by a firewall using ACLs, by a switch through VLAN tables, and by routing tables. All of these security policies are modifiable by the physical autonomic loop. For our implementation, two VLANs are plugged between the firewall and the physical machine.

The *hypervisor* (KVM) contains Linux-specific policies, such as internal routing tables, or memory associations between physical and virtual devices. In the figure, `peth0` represents the physical Ethernet interface, `eth0` and `eth1` are physically virtualized interfaces, while `br0` and `br1` are the bridge abstractions needed by the hypervisor to switch on/off an interface. Bridge `br0` memory is simply associated with `eth0`, as `br1` with `eth1`. Each bridge will be the endpoint of an emulated interface for VMs. The libvirt API handles the remote access to establish the hypervisor-layer autonomic loop.

At the *VM layer*, we consider two types of VM: the *administrative VM (AVM)* and the *user VM (UVM)*. A UVM contains at least two components: a firewall, to isolate network flows, and an antivirus that take cares of data execution prevention, program isolation and kernel signals. In the antivirus kernel component, probes monitor the loaded images in the guest OS. UVMs communicate with the hypervisor through the `conn1` connection endpoint, connected to `br0` – a second VM would be connected



(a) (b)

Figure 21: IaaS Framework Instantiation: (a) Computing View; (b) Networking View.

through `br1` and so on. The UVM virtual memory is mapped to the physical machine memory. They run on the virtual CPU (vCPU) abstraction provided by the hypervisor.

The AVM collects probes from every layer and organizes framework decisions with orchestrators (vertical and horizontal). The AVM behaves as a security management interface, collecting threat information, and deploying counter-measures. In each layer, the autonomic managers (HO_X) negotiate with both a centralized VO, and with the layer management APIs.

4.1.4 *Framework Implementation*

At the hypervisor layer, the libvirt API uses a library named netcf to enforce new network rules via XML. Although the frontend is clearly defined, the backend is OS-dependent: we thus have to fully translate netcf's XML configuration files and to implement commands for interface creation, modification and deletion. Actually, bridge modifications are fully implemented in order to have a first concrete example (use case described in Section 4.1.2).

In the VM layer, we are using ClamAV as a flexible antivirus with Python support for remote control as we can modify the source code and create VESPA-ready interfaces. Real-time protection is missing, so we implemented a kernel module to scan files when they are loaded in memory thanks to `PsSetLoadImageNotifyRoutine` and control their execution by defining a `PsSetCreateProcessNotifyRoutine` that can collect ClamAV results with I/O request packet (IRP) and act accordingly. This implementation underlines what can be achieved in the VM layer: specific functions such as filtering socket creation to ban a range of compromised VMs can also be hooked.

Communications in heterogeneous environments require clearly-specified interfaces, e.g., using an IDL. Due to its good results (see the benchmarks of [1]), we chose the Google IDL implementation named protobuf[90] to implement communications between the HOs and the different layer components, and with the VO. To implement the VM-layer components, the C language was naturally used, as low-level programming is needed for the UVM. However, the HOs and VO were chosen to be implemented in Python, as those components only need to take decisions on a high-level.

This particular implementation was deloyed as security infrastructure for the French government-funded SelfXL project, aiming at self-management of large scale systems such as cloud computing infrastructures. It allows the realization of dynamic quarantine zones to isolate and clean potentially compromised VMs.

4.1.5 *Use Case Implementation*

The implementation of the use case defined in Section4.1.2 required two main features: (1) to easily control bridges created by KVM for VMs; and (2) to migrate VMs through physical equipments with libvirt. Bridge control can be achieved in many ways, but we will focus on the following methods:

- Each newly created VM is directly connected to a `vnet` sub-interface, all of them being bridged together to a single bridge. This is the classic way to perform such a task, but it resides on the capacity of KVM to handle the network. Unfortunately, during our tests we were unable to recover connectivity after deleting a `vnet` interface from the bridge.
- For each VM created, a virtual interface is created at the hypervisor layer. A bridge is also linked to this sub-interface and will be one end of the VM connection. Sub-interfaces can be Ethernet abstractions provided by IP aliasing, or KVM `vnetX` interfaces. This approach, although more complex during the creation process, does not suffer from any major problems. Creation and deletion are totally independent, and are based on classic Linux networking operations.

To properly migrate VMs, all hypervisor interface names are synchronized. This task is handled by orchestrators that manage an association table between VM and network names.

Communication between AVM and UVMs while the network is down can be solved in many ways, around one common idea: establish a shared zone.

- Just as VMware and VirtualBox install their add-ons, communication can be achieved by emulating the insertion of a CD-ROM. If mounted as read and write, it provides an easy buffer to transfer data back to the hypervisor.
- Instead of cutting the wire directly, the action can be to isolate the VM in a specific VLAN. This VLAN contains a network storage (or equivalent) that only handles and delivers simple messages.
- Virtual Machine Introspection [163] (VMI) techniques also provide VM monitoring directly through the hypervisor.

With such techniques, the antivirus can find and send patches to the VM for a virus that was not clearly identified before the network isolation operation.

4.1.6 *Benchmarks*

The VESPA self-protection capabilities are evaluated in terms of network performance impact, overall response time, and resilience to attacks.

Our testbed is composed of three physical machines connected by a Gigabit switch. Each machine has 4 2.2 GHz Intel Core i7-2720QM CPUs with 8 GB of RAM, and

is running an Archlinux distribution with a 3.2.7 kernel. Each machine runs a KVM hypervisor, with Intel-VT instructions enabled. Hosted VMs are running Windows XP with 256 MB memory and a single virtual CPU. A RTSP video server delivers MPEG2 videos with almost constant bitrates (12 Mbps). A first physical machine is reserved to run the video server. The two others host clients. One of those machines is dedicated to quarantine infected VMs. A storage pool of VM disk images is located on the quarantine machine and accessible through an NFS server. Bandwidth sensors are based on Linux `/proc` and `/sys` facilities. All tests were run above 100 times, only the 30 best results being kept.

4.1.6.1 *Self-Protection Intrusivity*

This experience evaluates the impact of the VESPA self-protection features on the overall video streaming application performance. We use one physical machine to host virtualized clients and the video server, and another machine to quarantine infected VMs. We measured bandwidth consumption over time of 6 client VMs streaming videos while protected by VESPA, under different infection rate : a virus is launched periodically in one of the VMs. We evaluate the framework impact by comparing the measured bandwidth with that without any virus launched: a virus detection implies VM isolation, thus not consuming bandwidth, as no data is received from the video server. Results are shown in Figure 22. The α parameter represents the number of virus instances per minute, ranging from 2 to 7 minutes ($\alpha = 0.417$ and $\alpha = 0.128$ respectively).

We notice that the measured bandwidth with (8.38 MB/s for $\alpha = 0.214$) and without (8.98 MB/s for $\alpha = 0$) VESPA are very close, the difference being about 7%. The VESPA impact thus appears perfectly reasonable in practice.

4.1.6.2 *Overall Response Time*

This experience evaluates the overall latencies (in seconds) to complete a full self-protection loop for different types of security adaptations already presented in the case study, both intra-layer and cross-layer. Evaluation results for each step of the loop are shown in Table 16.

The overall response time for a cross-layer loop is about 37 s. This is a rapid reaction time, acceptable to contain infections that are not acute, which is most often the case. Note that the major part of this latency comes from phases of migration to and from the quarantine machine. Reaction (step 2 on Figure 20) seems a good trade-off between strong security and low latency, making the reaction time fall to only 6 s.

In more detail, intra-layer loop (1) results show that walking through the hierarchy of framework entities during detection is fast (0.15s) compared to the effective cleaning operation (5.67s), which involves scanning memory and files for virus eradication. Adding layer interaction with network isolation (2) shows that adding extra security by cutting and reconnecting the network only costs 13% of the total response time of (1). Finally, as expected, results (3) emphasize that migration is expensive and represents 90% of the overall latency.

4.1.6.3 *Resilience*

To assess the self-protection abilities brought by VESPA, we use the methodology coming from dependability bechmarking proposed in [40]. The idea is to inject in the

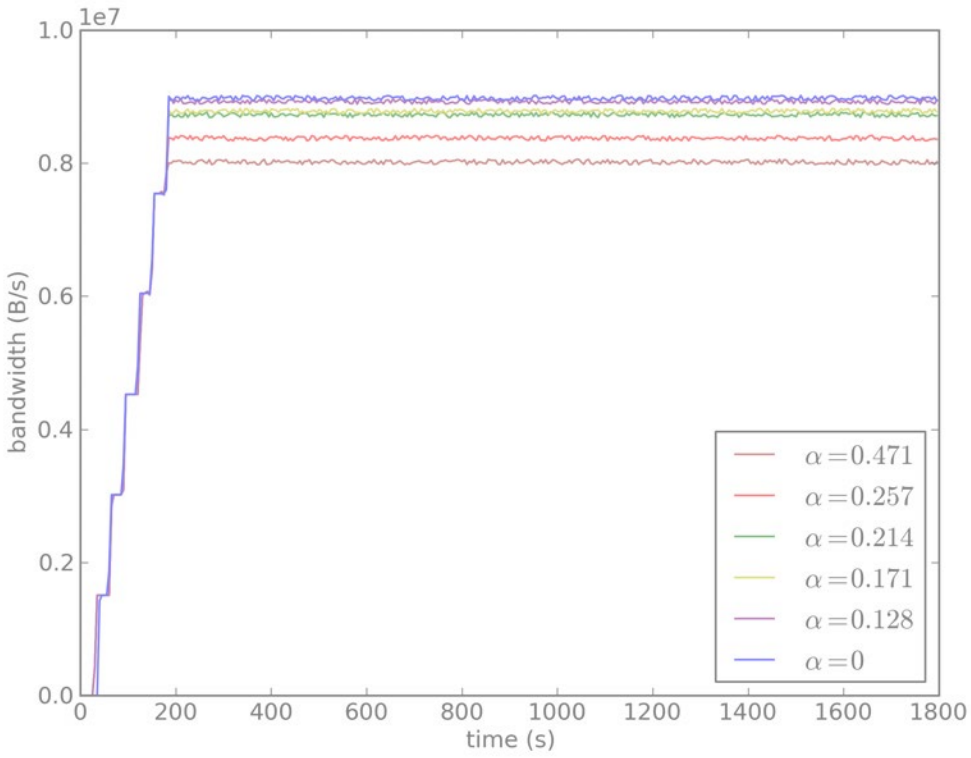


Figure 22: VESPA Intrusivity for Various VM Infection Rates.

Phase	(1) Intra-Layer Reaction	(2) Cross-Layer Reaction (w/o Migration)	(3) Cross-Layer Reaction (with Migration)
Detection	0.15	0.16	0.17
Decision	0.14	0.32	0.37
Disconnect	-	0.20	0.20
Migration	-	-	14.91
Cleaning	5.53	5.72	5.98
Migration	-	-	15.23
Reconnect	-	0.20	0.20
Total	5.82	6.60	37.06

Table 16: End-to-End Self-Protection Latencies.

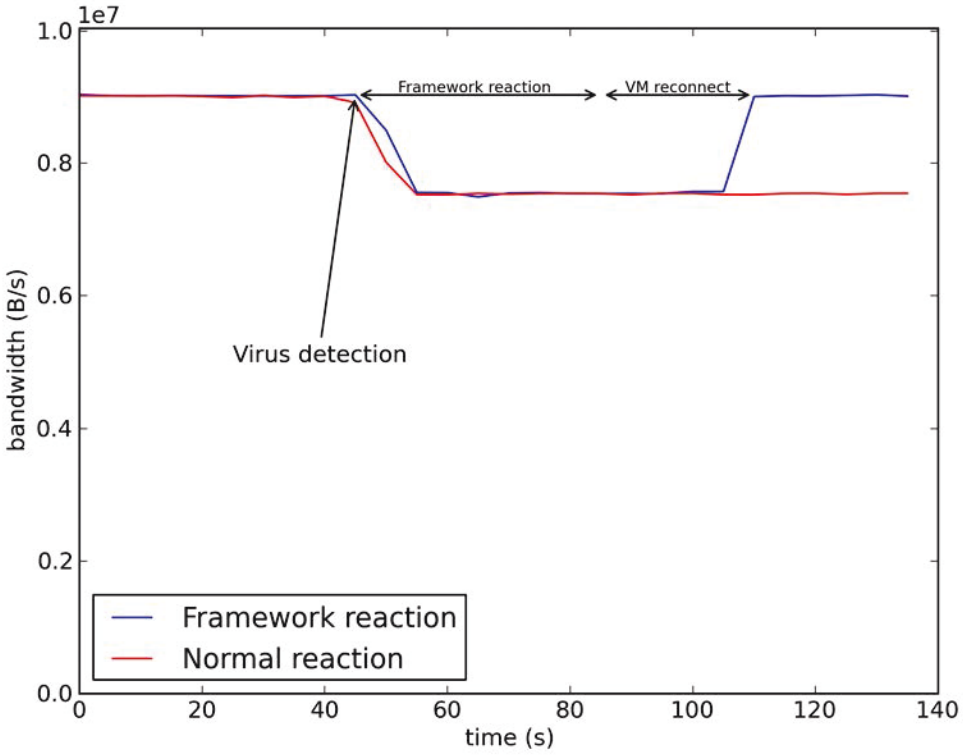


Figure 23: Self-Healing Capabilities with VESPA.

System Under Test (SUT) a disturbance in the form of an attack, and see how the system may recover to a steady state, notably focusing on the speed of recovery, and on the impact on the performance workload.

In this experience, we measure the impact on video server bandwidth of injecting a virus in the client VM protected with VESPA. We compare two scenarios: delivering the video to the client with or without the protection framework. The virus is started at $t = 40s$ for both scenarios. Results are given in Figure 23.

In the scenario without protection, virus infection results in a drop in bandwidth. The user looses completely the video, streaming is never resumed, and the virus remains present. In the other scenario, after a few seconds, the virus is detected and handled by the framework. As already seen in the previous benchmark, the overall reaction time is about 40 seconds. However, bandwidth is not immediately restored, as the video player has to re-negotiate some streaming protocols, causing an extra delay (around 20 seconds). The video resumes seamlessly on the client, the user experimenting a limited freeze of the video player. Moreover, the virus has been eradicated. Recovery time is thus about 60s, with a drop in bandwidth of about 16%. These results show that with VESPA, a cloud infrastructure is able to protect itself effectively with a reasonable performance overhead.

4.2 MOBILE CLOUD E2E SECURITY SLA MANAGEMENT

Will mobile cloud computing bring the full power of the cloud to limited devices? For instance to reap business benefits of simultaneous secure usage of multiple environments on a single smartphone, as virtualization becomes embedded [95]? In any case,

the mobile cloud has torn a security veil between two heterogeneous worlds. As a new universe of malevolence is hurled from the cloud to the embedded domain, new protection challenges are raised like security de-perimeterisation, multi-tenancy, or trust management. These new threats also join forces with an increasingly flourishing set of mobile malwares [37, 76]. End-to-end security is thus no more [52].

Current solutions are ill-prepared to face the security heterogeneity challenge. They have tackled it either from the device or from the cloud perspectives. On the device side, several container-based solutions for device management [39] or embedded hypervisors [96, 99, 91] for strong isolation have been proposed to segregate personal and professional environments on the same mobile phone. However, those solutions usually consider homogeneous sets of devices, and hardly extend into the cloud. Conversely, a number of proposals have been made to set up secure, dynamic *Virtual Organizations (VOrgs)* to manage and isolate resources of grids [215, 81], clouds [34], or multi-clouds [110, 230, 188]. However, they usually completely ignore the device aspects.

To link the two worlds, and restore end-to-end security, three main features seem to be missing:

DEVICE-TO-CLOUD VORGS In the mobile cloud, diverse *execution environments (EE)* may run on devices, connectivity gateways, and cloud infrastructures, e.g., Virtual Machines (VMs), lightweight processes, or threads. EEs may be trusted according to different levels. To guarantee end-to-end security SLAs on a subset of the infrastructure, dynamic EE federation within a VOrg spanning both different devices and clouds should be possible [168]. The SLA is then established by defining, distributing, and enforcing a security policy [82] throughout the VOrg.

END-TO-END VORG ISOLATION EEs share the same infrastructure. They may be isolated according to different system [29, 34], hardware [23], or network [48] mechanisms throughout the mobile cloud. To enforce different classes of security SLAs, VOrg isolation should be performed transparently to the underlying technology. Isolation should also take into account the multi-layer (VM, hypervisor, hardware) dimension of the infrastructure.

AUTOMATED SECURITY SUPERVISION Given infrastructure complexity, automated capabilities of detection and reaction to threats are needed, within a VOrg and between VOrgs, to lighten administration and lower security incident response times. Supervision is required both *horizontally* (between security domains) and *vertically* (between infrastructure layers).

This section presents Orange OC2, a new mobile cloud security management architecture and implementation overcoming the previous limitations. Orange OC2 views the mobile cloud as a superposition of multiple, well-isolated security planes, referred to as *Orange Community Clouds (OC2s)*.

Orange OC2 looks highly promising for end-to-end mobile cloud security. First, each OC2 sets up dynamically a device-to-cloud VOrg connecting EEs wishing to share a common security SLA throughout the infrastructure (devices, gateways, clouds). A well-defined security policy is distributed and enforced within the OC2, resulting in a homogeneous security level among member EEs. Second, strict OC2 isolation is achieved independently from underlying isolation mechanisms thanks to a policy-based security management framework to distribute and enforce security policies. Iso-

lation is both horizontal – OC2s are overlays over cloud and device physical security domains – and vertical – EEs in different infrastructure layers (e.g., hypervisor, VM) may belong to multiple OC2s. Third, security may be autonomically regulated at several granularity levels: in a physical security domain, either in an infrastructure layer, or cross-layer; or spanning security domains in an OC2 scope. SLAs are thus dynamically enforced through consistent threat supervision in and across planes.

The section also illustrates the viability of deploying the architecture in practice. It reports on a proof-of-concept implementation of Orange OC2, in terms of lessons learned and perspectives. We present a case study of a cloud-based secure multi-point conferencing service accessible from mobile devices in several home networks. Several choices of security SLAs may be enforced end-to-end in multiple concurrent OC2s to address different security expectations.

The section makes thus two contributions: (1) the Orange OC2 architecture and its implementation; (2) an experience report on its deployment in a home network-to-cloud setting.

The section is organized as follows. We describe mobile cloud security requirements (Section 4.2.1). We present our architecture, in terms of system model and security supervision (Section 4.2.2). We finally describe the case study, present experimental results and lessons learned (Section 4.2.3), and conclude (Section 4.2.4).

4.2.1 *Security Requirements*

4.2.1.1 *Mobile Device Security Challenges*

Exploding complexity in mobile computing induces many threats. Mobile devices today are a complex layering of technologies, both hardware (e.g., baseband, application processors, sensors, storage units, security chips) and software (e.g., bootloader, hypervisor, OS, mobile applications). Such diversity added to single component vulnerabilities results in a large attack surface. Devices are also connected simultaneously to several cloud infrastructures and services. Each device then becomes a potential source of compromise propagation to such services. For instance, the Bring Your Own Device business concept may create dangerous shortcuts destroying company local security policies, such as network segregation.

Three key device security challenges should be addressed:

END-TO-END SECURITY SLA Devices are used simultaneously for very different purposes (e.g., personal e-mail browsing, mobile banking, corporate private data reading). The user is thus connected to multiple service providers with variable service security levels. Unfortunately, current device virtualization solutions are unable to manage simultaneously heterogeneous third-party security policies. Different classes of security SLAs should thus be provided and enforced end-to-end, from device to service provider.

ISOLATION Usage patterns and accessed services should strongly be separated using device capabilities.

SUPERVISION Device security mechanisms remain difficult to coordinate, vertically and horizontally. Automated security management is thus needed to react efficiently

to threats, and guarantee continuously a Quality of Protection (QoP). Fast, robust security policy updates are notably required.

4.2.2 *Architecture*

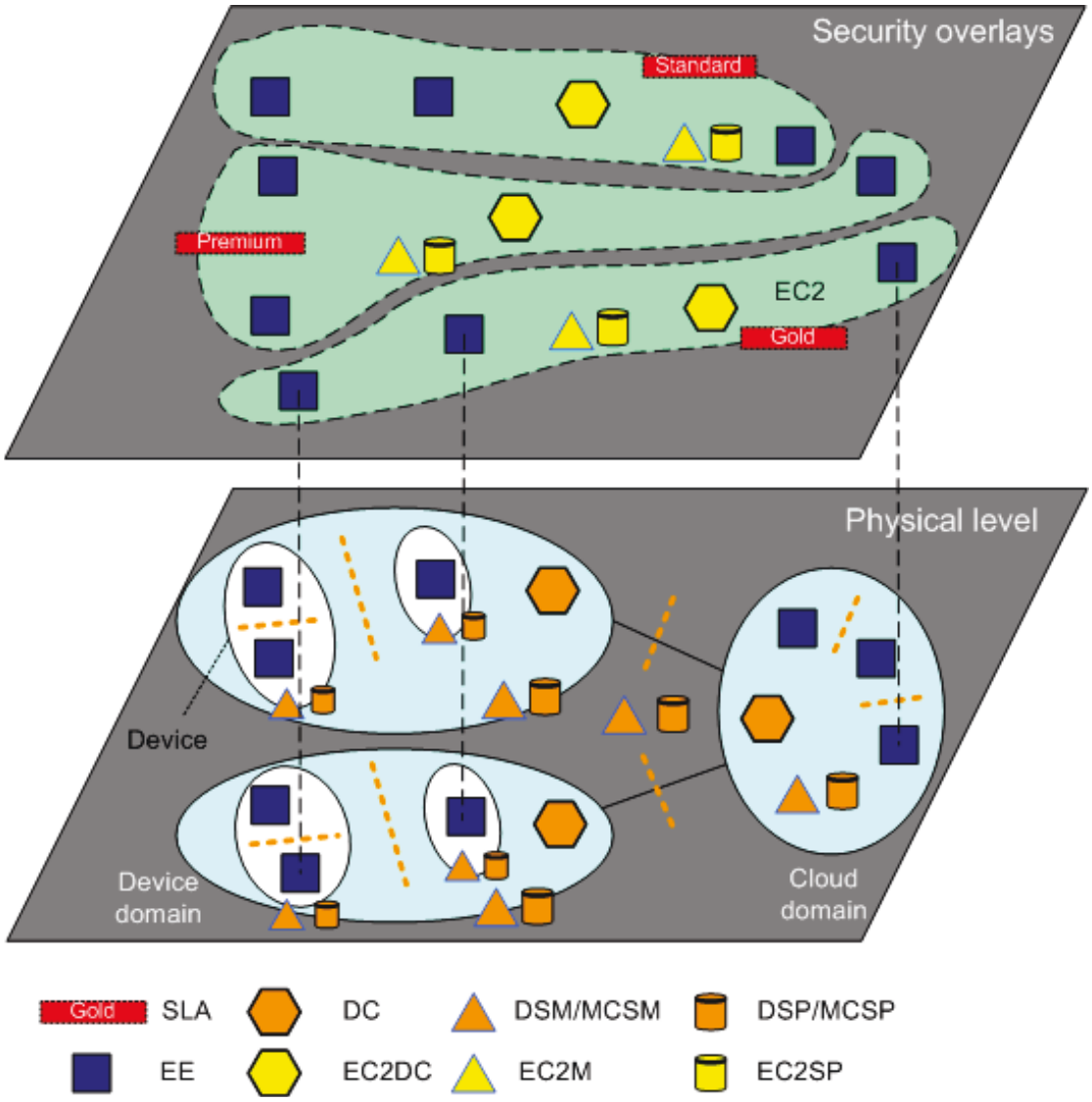


Figure 24: System Model.

4.2.2.1 *System Model*

As shown in Figure 24, our mobile cloud security architecture is composed of a *physical space*, and of a *virtual space* containing a set of security overlays.

PHYSICAL SPACE We view the mobile cloud infrastructure as a federation of physical *domains*, device- or cloud-based. For instance, a domain is the perimeter of devices connected to a same physical network, such as a Local Area Network. More generally, the concept of *Execution Environment (EE)* captures domain member entities. Horizontally, EEs may run on devices (e.g., threads, containers), connectivity gateways (e.g.,

lightweight processes), and cloud infrastructures (e.g., VMs). Vertically, EEs may be found in different infrastructure layers: VMs, hypervisor, or hardware.

A *Domain Controller (DC)* is in charge of EE management, with overall knowledge of EEs and of their capabilities. It defines the EE communication model, notably available interactions between EEs. It is also an event orchestrator for dynamic control inside the domain, such as automated EE configuration, or fault and security monitoring.

Domain security is under DC control: for each EE previously declared in the domain, authorized actions and interactions are captured in a *Domain Security Policy (DSP)*. EEs share the same infrastructure, and may be endowed with varying levels of trust. The DSP federates EE isolation requirements, enforced by *Domain Security Mechanisms (DSMs)* spread throughout the domain. Sample DSMs are an hypervisor embedded on the device, a cloud hypervisor, or hardware mechanisms [**?**]. Inter-domain isolation is managed at the network level through *Mobile Cloud Security Policies (MCSPs)* and *Mechanisms (MCSMs)*.

For device domains, the system model may include the concept of *device* as a particular type of domain, at an intermediate hierarchical level between EEs and device domains, with DCs, DSPs, and DSMs. To simplify, we consider a two-level model featuring only domains and EEs.

VIRTUAL SPACE Above the physical space, the virtual space abstracts and separates the mobile cloud infrastructure complexity according to levels of security SLA. Several overlays named *OC2s (Orange Community Clouds)* are introduced to capture definition and enforcement of a security SLA on EEs, spanning physical domains. OC2s are device-to-cloud VOrgs: OC2 EE membership aggregates that of EEs to the contributing physical domains, but sharing a homogeneous security level within the OC2 scope.

As for the physical space, OC2 security is governed by *Domain Controllers (OC2DC)*, *Security Policies (OC2SP)*, and *Security Mechanisms (OC2SM)*. Such virtual space abstractions are mapped to their physical space counterparts abstracting away domain boundaries.

4.2.2.2 *OC2 Lifecycle*

An OC2 has a 3 phase-lifecycle: (1) initialization; (2) isolation enforcement; and (3) supervision (see Figures 25, 26, and 27).

INITIALIZATION A distributed protocol defines how EEs may subscribe to an OC2. To join the gold-level SLA OC2, an EE sends a **join**$\langle \mathtt{level} = \mathtt{gold} \rangle$ to the Domain Controller DC_1 of its local domain D_1. The OC2DC may be the DC receiving the first **join** request for this OC2. Election among DCs contributing to the OC2 is also possible. After checking for SLA eligibility, the OC2DC registers the EE $\langle EE, D_1, \mathtt{gold} \rangle$, then translated into a $\langle EE, \mathtt{gold} \rangle$ mapping in DC_1.

To leave an OC2, an EE sends a **leave** request to the OC2DC, that removes the corresponding entry, and propagates the change to DC_1.

ISOLATION ENFORCEMENT The OC2SP π_{OC2} defines rules for isolating resources in the OC2. It aggregates the DSPs π_{D_i} of each physical domain D_i contributing to the OC2, restricting the scope to OC2 resources. To enforce such rules, OC2 isolation is built on the DSM_is, which are the isolation capabilities in each D_i.

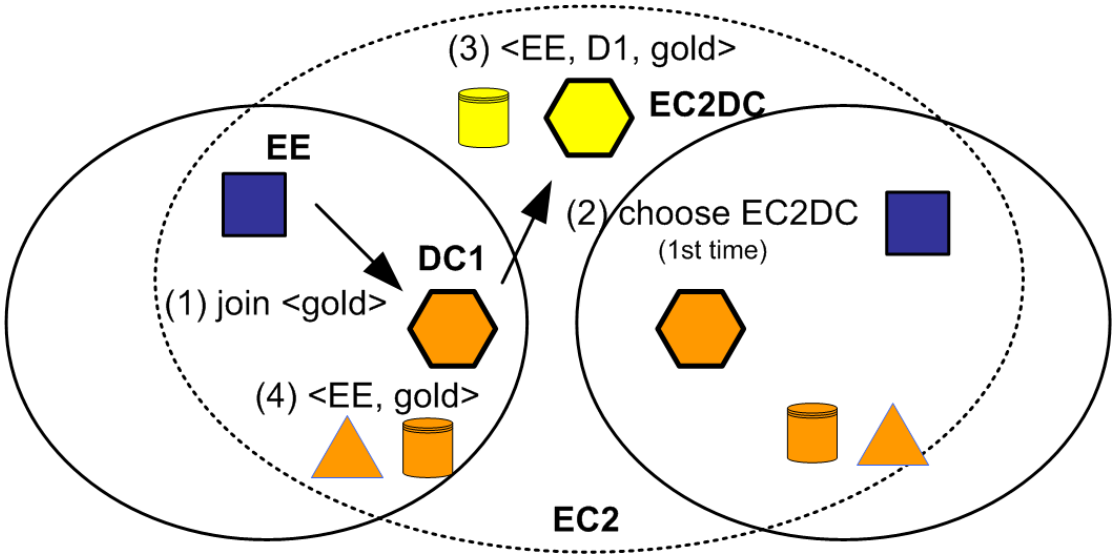


Figure 25: OC2 Creation.

Materializing the OC2 with a homogeneous security level among member EEs is performed as follows: (1) distribute π_{OC2} throughout the OC2; and (2) enforce it in EEs, relying on the DSM_is.

Step (1) contains two sub-steps: (1a) propagate π_{OC2} from the OC2DC to the DC_is; (1b) in each DC_i, translate the relevant received part into a DSP π_{D_i}, for enforcement in step (2). Implementing that protocol relies on the VESPA framework, described further [223].

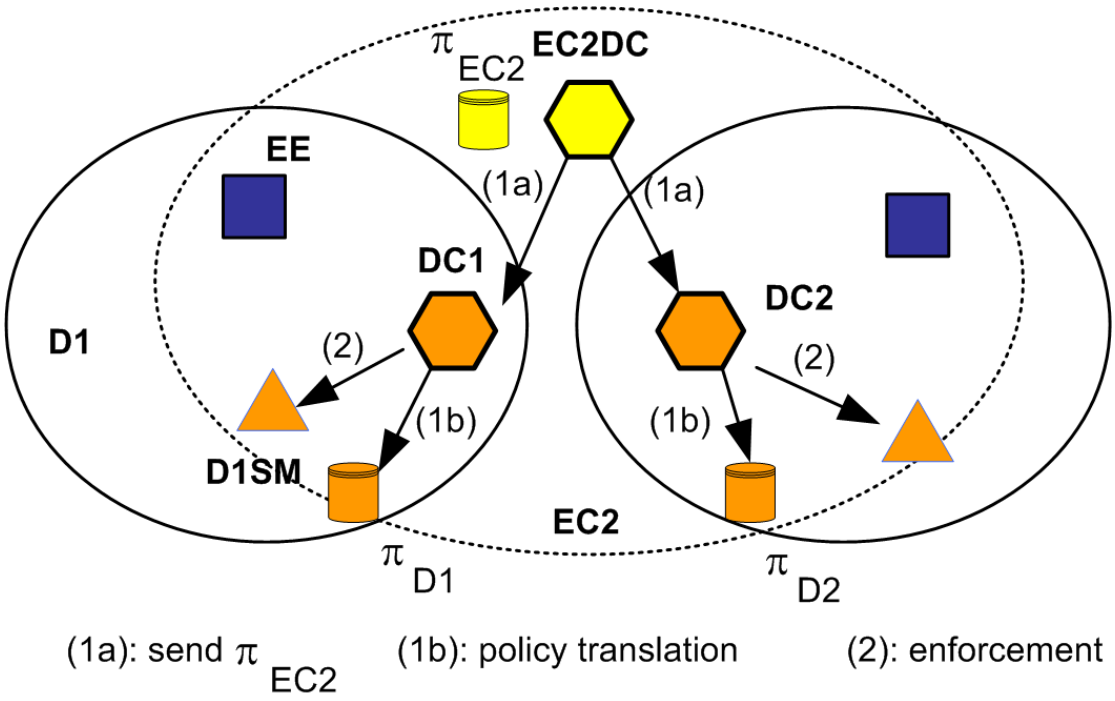


Figure 26: OC2 Isolation.

SUPERVISION Automated security supervision allows to simply respond to detected threats in an OC2. A typical cross-domain security response is: (1) detect sus-

picious activity at EE level, and notify the DC and the OC2DC; (2) select a reaction policy in the OC2DC; (3) enforce the reaction in EEs of other domains through their DCs. Response may also be intra-domain only, for instance to guarantee that inter-OC2 isolation is preserved over time. Decision is then directly taken at DC level. As EEs may be located in different infrastructure layers, cross-layer security supervision is also possible, with also finer-grained security adaptations at layer level [223].

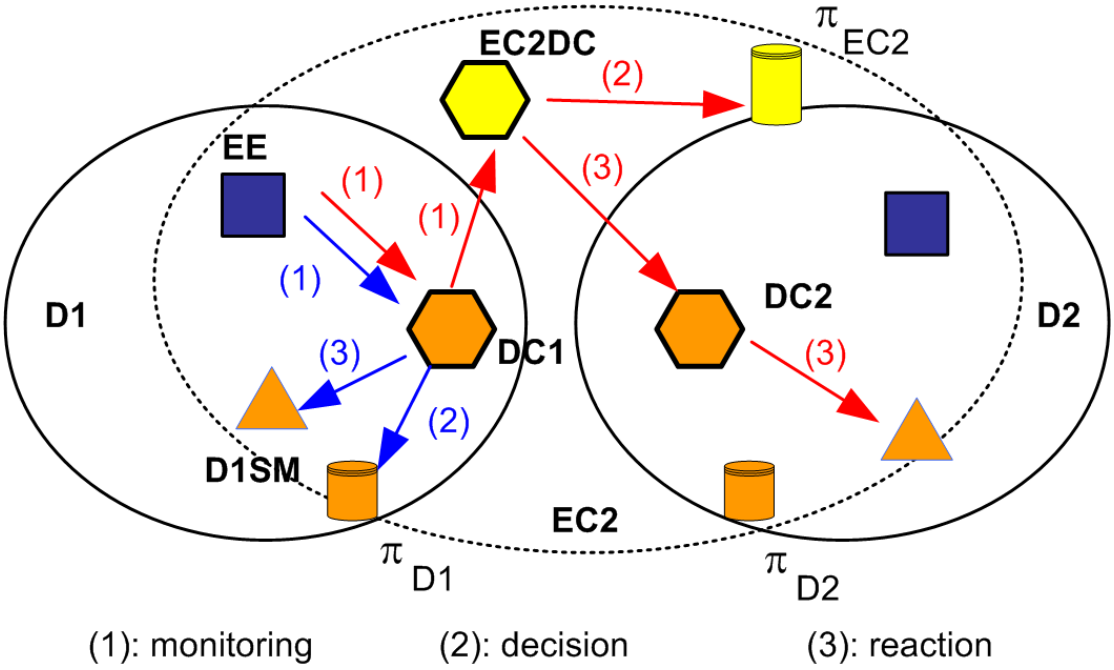


Figure 27: OC2 Supervision.

4.2.2.3 *Implementing Intra-Domain Security*

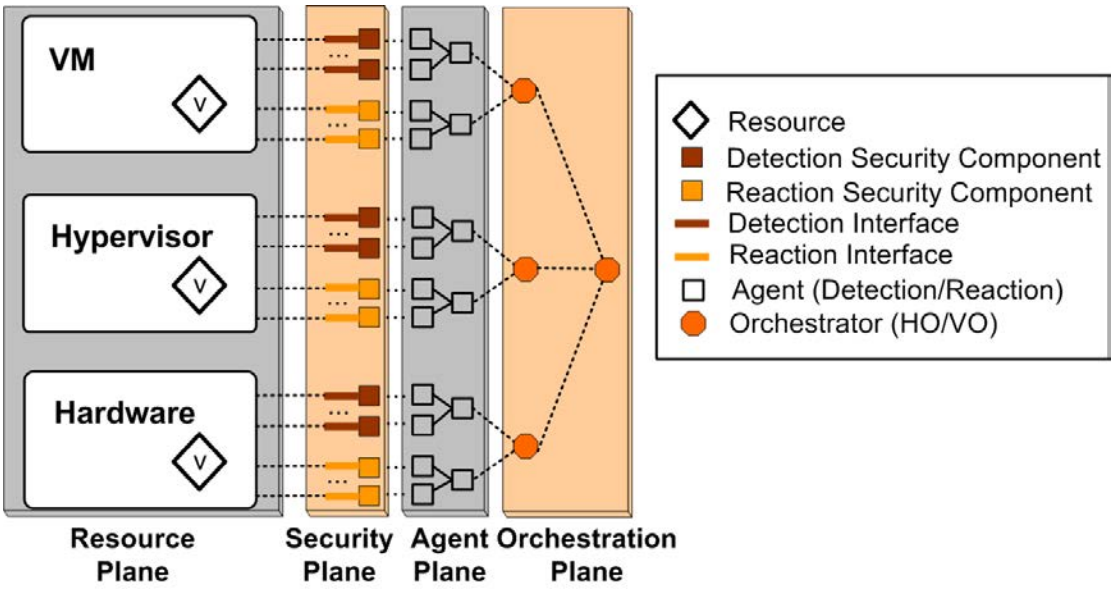


Figure 28: VESPA Architecture.

To reach an implementable architecture, we refine the system model using the VESPA framework [223]. VESPA provides policy-based security administration of cloud components through multiple coordinated autonomic loops (see Figure 28).

With this framework, security supervision of a physical domain may be delegated to the control of a single VO, to implement a DC. The resources to protect and control are simply member EEs, of which the VO has overall knowledge. Through HOs, the VO is also aware of all layer information in the domain. This design allows both intra-layer and cross-layer automated security supervision of the domain.

4.2.2.4 *Implementing Inter-Domain Security*

For inter-domain security, an OC2DC must be implemented, focusing on isolation and cross-domain security supervision, as described previously. This is achieved by deploying several VOs, working in close cooperation, with several possible VO-to-VO communication patterns, e.g., master-slave, or fully peer-to-peer. The VO interprets and enforces security policies on resources in its domain, and propagates to neighboring domains changes in isolation/reaction policies and in security context. This realizes and supervises effectively the OC2.

Isolation works by selection, distribution, and distributed enforcement of the isolation policy from source to remote VOs. *Supervision* works by: (1) propagating to remote VOs security-relevant changes in the local domain; (2) defining and propagating domain-specific responses based on local and remote security information; (3) enforcing received remote reaction policies, also based on the local security context. How such operations may be realized in practice, focusing on isolation, is illustrated next.

4.2.3 *A Scenario for Mobile Cloud*

4.2.3.1 *Scenario and Implementation Overview*

To validate our architecture, we consider a case study featuring multiple home networks connected to the cloud. In each home, viewed as a *device domain*, several users, with tablets, smartphones, laptops, or TVs, subscribe to multiple Service Providers (SP) through the home gateway. For each SP, its hosts form a *cloud domain*. We focus on a SP providing secure meetings on-demand. Several levels of security are available for the service, captured by the SLAs shown in Table 17.

SLA	Standard	Gold	Premium
Dedicated professional VM available	✓	✓	✓
Secure communications (VPN)	✓	✓	✓
Trusted device usage only	✗	✓	✓
Web filtering	✗	✓	✓
Professional VM exclusive execution	✗	✗	✓

Table 17: Classes of Security SLAs.

On each equipment, VESPA agents monitor and enforce the SP security policy π_{SP}. To manage device-to-cloud SLAs, OC2s are established as follows (see Figure 29): (1)

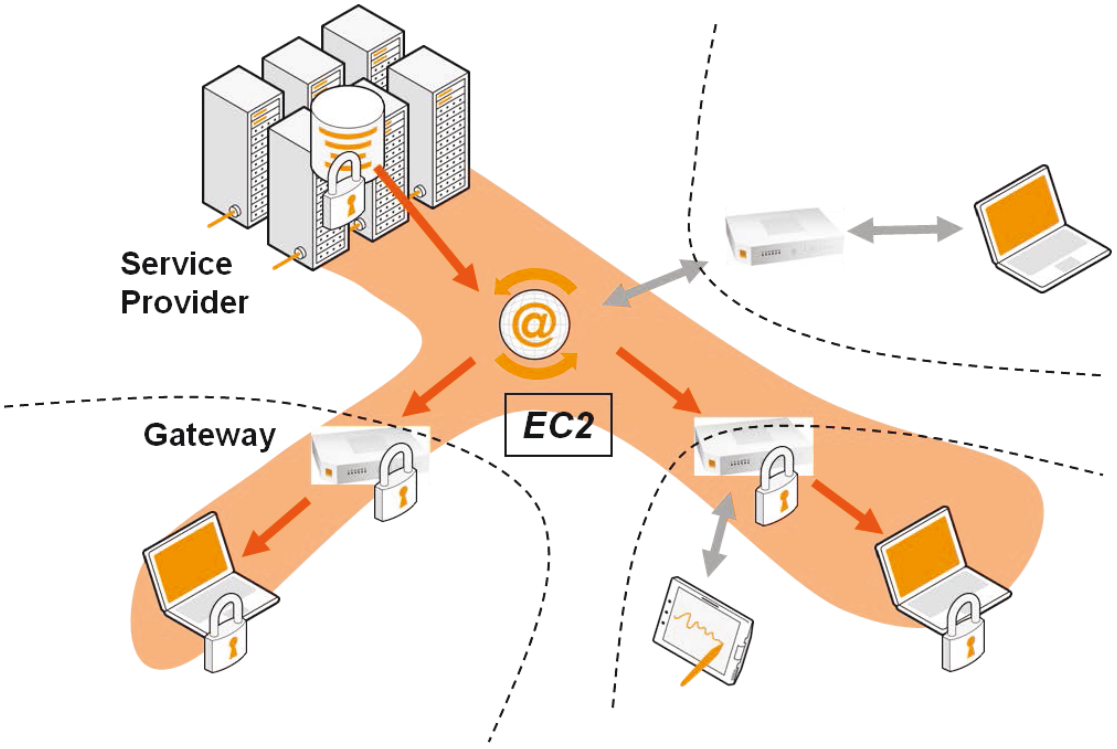


Figure 29: Establishing the OC2: Principle.

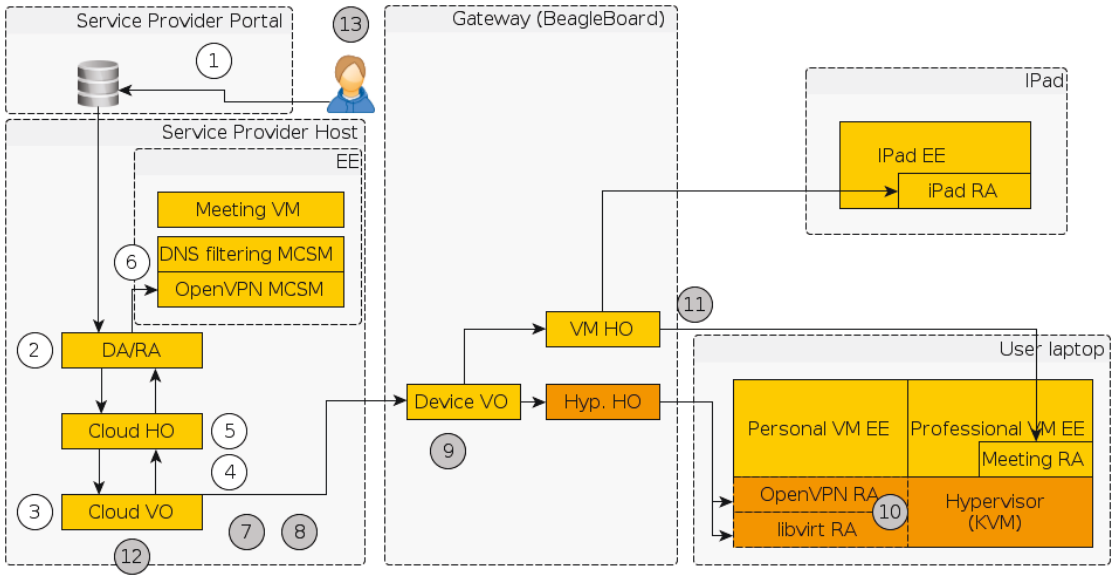


Figure 30: Establishing the OC2: Typical VESPA Deployment.

the SP defines π_{SP}; (2) π_{SP} is activated in SP cloud hosts; (3) π_{SP} is propagated into gateways, extending the OC2 from cloud to home domains; (4) π_{SP} is sent to agents in user devices, and enforced using their underlying security mechanisms.

We consider 3 users working from home subscribing to the service. The associated infrastructure contains one SP (cloud) domain connected over Internet to three device domains. Gateways are manageable home routers handling all network traffic. Each device domain is composed of a gateway and of a user laptop. One domain also contains an additional tablet (iPad) considered untrusted, from which the user may follow the meeting. Each laptop contains three EEs: two VMs and their hypervisor. One VM is dedicated to professional use. The other VM is for personal use, and is untrusted. The SP domain contains two types of EEs: on-demand meetings VMs and the cloud hypervisor. A web portal VM also offers a management interface for the service.

In our scenario, the organizer sets up a meeting with the *Standard* security SLA: a professional VM becomes available on laptops of participants with secure VPN interconnection; untrusted devices are also authorized in the meeting. To create the OC2 and enforce the SLA, the *Standard* π_{SP} is installed on cloud hosts, and propagated to gateways and user devices. Agents set up the VMs, VPN, and allow the iPad to join the meeting. The organizer then upgrades the SLA to *Gold* for more confidentiality: untrusted devices are no longer allowed in the meeting, and web filtering is enforced. The meeting session is thus closed on the iPad. Finally, the *Premium* SLA is chosen: the professional VM has exclusive access to laptop resources. Execution of the personal VM on the laptop is therefore frozen.

Figure 30 shows the instantiation of our architecture for this scenario. One VO and two HOs, handling VM- and hypervisor-layer tasks are deployed in each gateway. Deployment is similar on the cloud side. VESPA agents are present in all layers and hosts to manage the EEs previously described.

Collaborating VOs form a distributed OC2DC implementation. VOs regularly monitor and exchange DSPs to build inter-domain OC2SP knowledge. Each DSP is stored in a policy file attached to the related VO. Upon a DSP modification event, a VO applies administrator-defined reactions to the domain, and propagates updated policy files to other domains. For instance, when the meeting is initialized with the *Standard* SLA, an agent updates the SP VO policy file with the corresponding OC2SP. The policy is interpreted by the VO, and propagated to VOs of device domains for enforcement through domain isolation mechanisms.

Detection starts when the administrator selects a new security SLA on the conference web portal (1). A VESPA DA checks the validity of the request, and notifies the cloud orchestration plane (2). The Cloud VO selects a reaction policy based on administrator-defined strategies (3). The reaction policy is then sent to the HO of the cloud domain (4), where it is sanity checked (5). Cloud isolation mechanisms (e.g., OpenVPN, DNS filtering) are then applied to enforce the required security locally (6).

In parallel, the security policy is propagated to other domains in the OC2 scope. The Cloud VO selects neighboring domains (7). It then sends them the policy (8), verified by the device domain VOs upon reception (9), before enforcement. The Device hypervisor-level HO launches an OpenVPN RA to establish a secure network connection with the meeting server, and a libvirt RA to control VMs, e.g., initialize, authorize, block (10). The Device VM-level HO orders a dedicated meeting RA to start a conference call in the professional VM (11). It also manages iPad authorization to participate

Phase	Intra-Domain Security Loop		Cross-Domain Security Loop	
	Latency (ms)	%	Latency (ms)	%
Detection	**0.539**	**0.05**	**0.539**	**0.006**
1 - Select SLA	0.003		0.003	
2 - Check SLA request	0.536		0.536	
Decision (cloud-side)	**2.746**	**0.25**	**2.720**	**0.033**
3 - Select reaction policy	2.720		2.720	
4 - Notify HO	0.026			
Reaction (cloud-side)	**1076**	**99.65**		
5 - Check reaction policy	0.025			
6 - Enforce isolation	1076			
Distribution			**3.252**	**0.040**
7 - Select remote domains			0.421	
8 - Send policy to domains			2.831	
Decision (device-side)			**0.061**	**0.001**
9 - Verify received policy			0.061	
Reaction (device-side)			**8181**	**99.796**
10a - Establish VPN			118.8	
10b - Authorize/block VMs			7034	
11 - Start conference call			1028	
Post-processing	**0.494**	**0.05**	**10.17**	**0.124**
12 - Check OC2 establishment	0.483		10.16	
13 - Notify user	0.011		0.011	
Total	**1079.8**	**100**	**8197.7**	**100**

Table 18: End-to-End Latencies.

to the conference. Finally, the Cloud VO checks that OC2 enforcement is effective (12), and notifies the user of the achieved isolation level (13).

4.2.3.2 *Experimental Results*

We performed the following evaluations of our proof-of-concept. We verify that security adaptations can be achieved in near real-time, also measuring the contribution of each phase to the response time. We evaluate the ability of the framework to control multiple domains with negligible overhead. We illustrate end-to-end security guarantees, showing how OC2s are resilient to network attacks with fast recovery time.

RESPONSE TIME Table 18 shows latencies for different phases of intra-domain and cross-domain policy selection, distribution, and enforcement in the use case. Overall performance is acceptable, with a total time of about 1s to set up the OC2 on the cloud-side, and 8s end-to-end.

For intra-domain security, isolation enforcement is expensive, covering 99.6% of the response time. A possible explanation is that some isolation services need to be restarted to deploy new security configurations. The remaining 0.4% is the overhead

of the VESPA framework. Detection and reaction are lightweight, due to fast inter-agent communications and minimal policy refinement processing from the Cloud HO to MCSMs. The higher decision cost reflects mapping the security SLA/SLOs for each security metric to a high-level security policy configuration to be enforced. This includes matching security state with administrator-defined strategies.

For inter-domain security, policy distribution is fast due to an efficient flooding-based network protocol. The main overheads for reaction come from I/O slowdowns in device enforcement mechanisms, in the hypervisor and VMs. Other overheads such as Device VO decision-making, or VESPA reaction agents are negligible, similarly to the cloud case. Yet, overall response time remains relatively acceptable.

SCALABILITY We also measured how the security adaptation latencies depend on system scale, captured by the number of physical domains. Results are shown in Figure 31. Detection time is slightly increasing with the number of domains. Administrator-defined OC2 policies become larger with more domains. Thus, more time is needed to match such policies with incoming agent alerts. Policy distribution time is proportional to system scale: in our proof-of-concept, the propagation protocol has to be repeated for each device domain. Performance may be improved with a broadcast protocol, instead of the current multi-unicast protocol to distribute security policies. Reaction time is not really impacted: only light distributed verifications are performed after all reactions to ensure successful cross-domain SLA enforcement. We found latency results for combined detection, distribution, and reaction to scale well in terms of domains. However, enforcement times crushed results of all other phases. Thus, further work is required to assess overall architecture scalability. Our current proof-of-concept currently supports 4 domains. Nevertheless, we guess our approach should support far more domains before the framework overhead reaches enforcement costs.

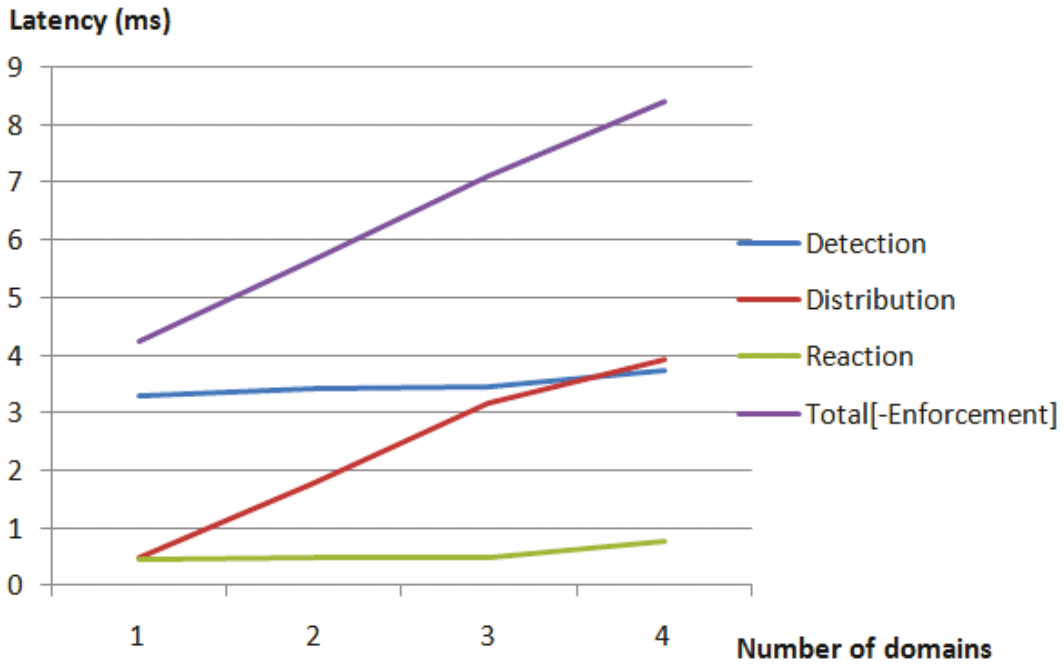


Figure 31: Latency vs. Number of Domains.

END-TO-END SECURITY We evaluate end-to-end security guarantees by showing how the mobile cloud recovers from a security SLA violation, taking as example a network attack. We consider three isolation levels: *High* (SLA fully enforced), *Middle* (SLA partially enforced), and *Low* (SLA violated). An OpenVPN connection is attacked while the meeting is in the *High* level. Figure 32 shows how our self-protection framework handles the attack. We notably measured the time required to recover the best isolation level.

The OC2 starts at *High* level: its boundaries are established, and security of its resources is supervised ($t \in [0 - 8.47s]$). The attack is launched by killing one OpenVPN client on a random OC2 device ($t = 14.33s$). The OpenVPN client is not monitored by the framework, but connection errors are watched on the cloud side: the VESPA agent parsing OpenVPN logs detects the error, sends an alert to the Cloud HO, forwarded to the Cloud VO. The Cloud VO notifies the OC2 of the SLA violation by propagating ($t = 14.49s$) a new *Low* level security policy for enforcement ($t = 24.69s$) in the OC2. As a first step to restore *High* isolation conditions, the Cloud VO then sends the *Middle* level policy to OC2 Device VOs ($t = 24.79s$). This policy is received ($t = 25.26s$) and enforced ($t = 26.34s$). The Cloud VO finally decides to apply the *High* level, enforced on the cloud side ($t = 30.2s$) and device side ($t = 38.8s$), restoring full isolation.

Results show that the attack is detected in $\delta = 0.16s$. The security SLA may be restored to *High* in $\Delta = 24.47s$, which appears quite reasonable to recover end-to-end mobile cloud security. Some high latencies in SLA level propagation are caused by slow BeagleBoard SD card I/O speeds. Such issues are under investigation to improve the response time.

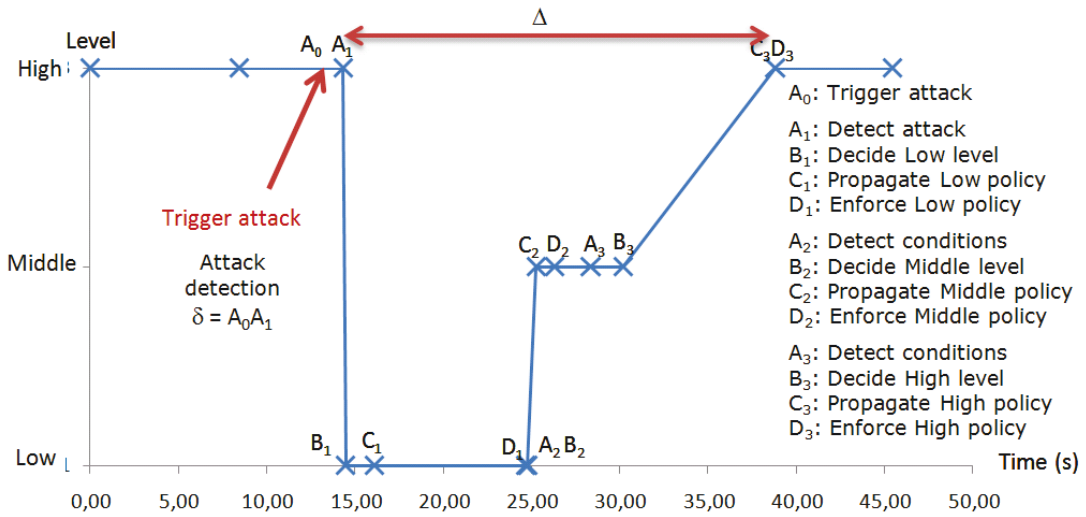


Figure 32: Security Evaluation.

4.2.3.3 *Limits and Perspectives*

Our current implementation suffers from a number of limitations. When an EE does not respond, for instance, because of a Denial of Service, the associated VO cannot decide if the EE is corrupted, buggy, or shutdown. A default choice may be defined when such situation occurs, but more in-depth study is needed. Actual communications between framework components are currently neither encrypted, nor integrity-checked. Thus, a man-in-the-middle attacker can bypass enforced isolation by faking EE alerts and forcing the VO to extend the OC2 and loosen isolation. However this

attack requires knowledge of the framework architecture and of the underlying policies. Notwithstanding, we see two solutions to overcome this limitation: using VPN between components, or defining a new communication component into agents to use SSL. The latter is currently under development.

Case study results tends to show that the proposed architecture is promising for flexible mobile cloud end-to-end security. Orange OC2 also opens a number of business opportunities for telco operators. At least three key segments of the architecture and associated stakeholders should be considered: (1) end-user mobile devices (OS developers, mobile network operators); (2) residential gateway and broadband access (fixed network operators); (3) cloud services (cloud service providers). A fixed and mobile network operator is involved in the distribution, development, and operation of all components and is thus a legitimate and efficient actor to provide a seamless operation of the whole architecture in a unified manner. Convergent operators could leverage this framework to greatly simplify the digital life of their customers by providing them working and adaptable end-to-end security.

A key pending question is: which component should be responsible for operating security policy enforcement and providing enablers to other actors? This decision and control point has to be strongly secured, as the security of the whole architecture assumes trust in this entity. As this entity will have strong control over devices, it could be convenient to have it located into end-user premises to avoid exposing the devices to Internet. The gateway could easily act as this trust anchor for the whole architecture, the control logic being either hosted in the gateway or in a cloud service under the control of the gateway. Fixed network operators have control over the software and hardware of the residential gateways they provide to their customers. Locating the trust anchor in this gateway would also allow such operators to offer cloud service providers secure access and control to mobile devices of their customers.

4.2.4 *Conclusion*

This section addresses the gap between cloud and mobile device security with the Orange OC2 architecture. Different classes of security SLAs may be defined and uniformly enforced for execution environments in device or cloud domains with the OC2 abstraction. Strict OC2 isolation is achieved independently from underlying mechanisms with the VESPA framework for security policy distribution and enforcement. Security may be autonomically regulated at several levels of granularity within and across OC2s, across domains and within infrastructure layers. Evaluation results show that the architecture may be deployed effectively in practice. End-to-end mobile cloud security comes with a reasonably small price in performance, such as fast recovery of SLA violations. Orange OC2 opens interesting perspectives such as Ã la carte security abstraction layers to interconnect isolated environments with diverse security needs between different service providers or pooled on single devices.

VOs currently push security policies without taking into account the peer security state. We are evaluating alternatives regarding policy negotiation to deal with conflicts. Another issue is secure policy transmission. We are working on a lighter protocol with built-in security, such as communication integrity and replay attack prevention. We also plan to extend our framework to support TrustZone technology, e.g., to protect VESPA agent integrity. Finally, current framework deployment requires manual intervention. We are thus striving for fully automatic security agent distribution, for instance using [74].

4.3 FUZZING INTERRUPTS

So far we have seen how to build a security solution using the VESPA framework, and how it brings autonomic security. To benchmark VESPA communications and internal mechanisms, we decided to stress the framework by fuzzing the interrupts of the KVM hypervisor. It is an attacker-oriented use case with a strong collaboration between the VM and the hypervisor.

4.3.1 *Frameworks*

Several fuzzing frameworks [196] provides facilities to instrument and test implementation limits. From network protocol [237] to cloud entrypoints [125, 234], softwares are tested against unusual inputs and under a strong load. The output consist of software behavior, especially segmentation faults, which are the preferred way to gain privileges through exploitation.

Thus, to evaluate performances of the VESPA framework, we use it to instrument the hypervisor and generate inputs at the maximum speed.

4.3.2 *Architecture*

Our architecture (see Figure 33) is made of two hosts: (1) a supervision hosts containing the orchestration part of the VESPA framework, and (2) a virtualization ready server with a KVM hypervisor and a virtual machine.

The scenario is the following: The virtual machine sends an interrupt, that is handled by the hypervisor and forwarded to the Qemu component. The latter executes a function if the I/O port is correct, and gives control back to the hypervisor.

The initialisation step contacts the hypervisor and parses the Qemu log to record valid I/O port. Then the VO policy generates an interrupt request toward the HO_VM, forwarding it to the fuzzing agent. This agent gather root privileges and send a *outl* operation corresponding to the VO request. The interrupt goes through the hypervisor, where we log every interrupts and send a summary back to the VO. Finally, the VO analyzes the summary, and continue if the interrupt was correctly handled, otherwise the VO enters into the recovery process. First the stack trace is saved for further analyzes, second the VM is restarted as the device emulator crashed. If the problem caused the hypervisor to completely crash and hang the server, we are able to reproduce it after a reboot and proceed to manual analysis.

The send and wait mechanism of this simple loop is not efficient, and we will see how to offload calculations and leverage performances.

Another problem is to deal with interrupts rebooting the machine without writing into logs. Indeed, the VO is waiting for I/O result either by the agent controlling the fuzzing tool or with the qemu log. Here the machine reboots without giving a chance to get some information and freeze the framework. Our solution is to launch the agent generating fuzzing with initscripts to connect back to the framework, and be able to resume the fuzzing.

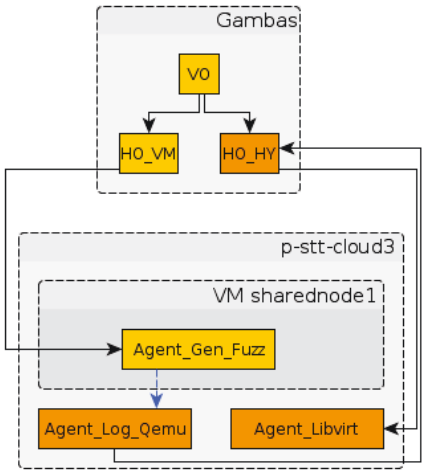


Figure 33: Hypervisor Fuzzing Architecture.

4.3.3 *Policies*

The policies used into the use case are represented on the Figure 34. It is the representation of the steps described previously. We can move from a state to another when conditions are met. Regardless of the language, it can be implemented as a switch-case scenario.

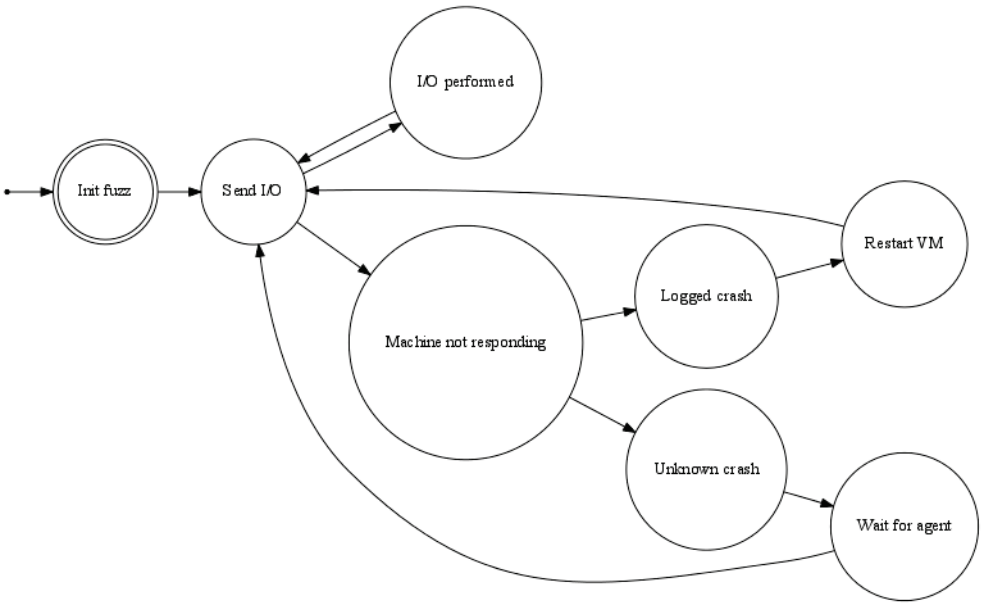


Figure 34: Fuzzing I/O Finite State Machine.

4.3.4 *Performance evaluation*

We evaluate the fuzzing scenario in terms of difficulty of adaptation and time saved compared to empiric implementation.

OPTIMIZATIONS The first results showed a slow 300KB/s as the maximum bandwidth handled by the framework on a 100mbps link. Indeed, the synchronous aspect of the framework slows down message propagation. Thus, we modified the VESPA communication component to integrate asynchronous events. We also extended the policy model to handle fuzzing without message acknowledgement.

This improvement comes without impact on previous policies definition, as message are sent synchronously by default. With the new policy, we are able to reach the maximum link bandwidth at 100MB. However, we need to offload some processing to save some bandwidth. Thus we use the aggregation policies to optimize I/O information, for example a vector is transmitted with the first and the last values. The agent buffers 0xffff log lines and transform the vector $< [0,1][0,2][0,3]...[0,0xfa][0,0xfb][0,0xfc] >$ to $< [0,1][0,0xfc] >$, meaning that all values between are present.

TRADE-OFFS We underline the trade-off to choose between network usage and CPU usage to adapt the framework. Whenever a bottleneck appears, we can split it among the other layers, e.g. if the CPU is above 80% usage. We can use another compression algorithm with less CPU usage, and thus less compression, moving the load from CPU to network bandwidth.

The time needed to assess all possible values is also high. Figure 45 (Appendix B.1) details the I/O ports available on an average machine under Qemu. We have 0xffff ports with 0xffffffff possible values, meaning 281470681677825 tests. However we measured that our framework is able to perform 58374.570763 I/O/s on a single VM, and obvious linear fuzzing requires 55807.9 days.

$$\frac{0xffff * 0xffffffff}{58374.570763} * \frac{1}{60 * 60 * 24} = 55807.9 I/O/s \quad (1)$$

Restricting the I/O ports to only registered interrupts handlers divide the possible ports to around 600, and our linear fuzzing need 465.8 days to be fulfilled.

$$\frac{547 * 0xffffffff}{58374.570763} * \frac{1}{60 * 60 * 24} = 465.8 days \quad (2)$$

Now the VESPA framework come in action and empowers the fuzzing by distributing the computation over multiple cloud IaaS. A reasonable tradeoff between the number of machines and the fuzzing time is with 1000 VMs. A single server may support up to 50 light VMs, thus we need 20 servers. Distributed computing is straightforward with the framework in our scenario and the only problem is the deployment. We did not integrate the VAMP [73] framework, and components are pushed on the VM template. If there is a programmatic error, all components are then pushed through scp. The complete fuzzing now takes around 11 hours to be fulfilled in theory, and 15h according to our experiments. The difference is explained by the timeout while loosing the agent_gen_fuzz agent and the time taken by a VM to reboot while the CPU is stressed.

$$\frac{547 * 0xffffffff}{58374.570763 * 1000} * \frac{1}{60 * 60} = 11.18 hours \quad (3)$$

4.3.5 *Conclusion*

This fuzzing use case showed how to build virtualization aware application with cross-layer and cross-domain interactions. We explored VESPA benefits to adapt an expensive problem taking many years of computation, to only eleven hours.

DISCUSSION Testing the hypervisor is necessary to discover weaknesses. However our tests focused on the benchmark of the VESPA framework, and several ways have to be explored. Our fuzzing is quite brutal and can be optimized for more efficient vulnerability assessment.

First, the code coverage. A good practice when it comes to fuzzing is to use the smallest input that will go through the maximum of code. Figure 35 details the time needed to send `0xffff` values on each valid ports. Each bar represent the mean for the number of interrupts tested on a port during 10 tests, and the standard deviation is at the top of bars in black. The speed varies from one to three, meaning that the associated code is more complex to handle. While it is not a direct implication, it is handy to detect the largest portion of code. The outlier above 300000 IO/s is performing a no operation, while the two bars below 10000 represents CPU consuming routines.

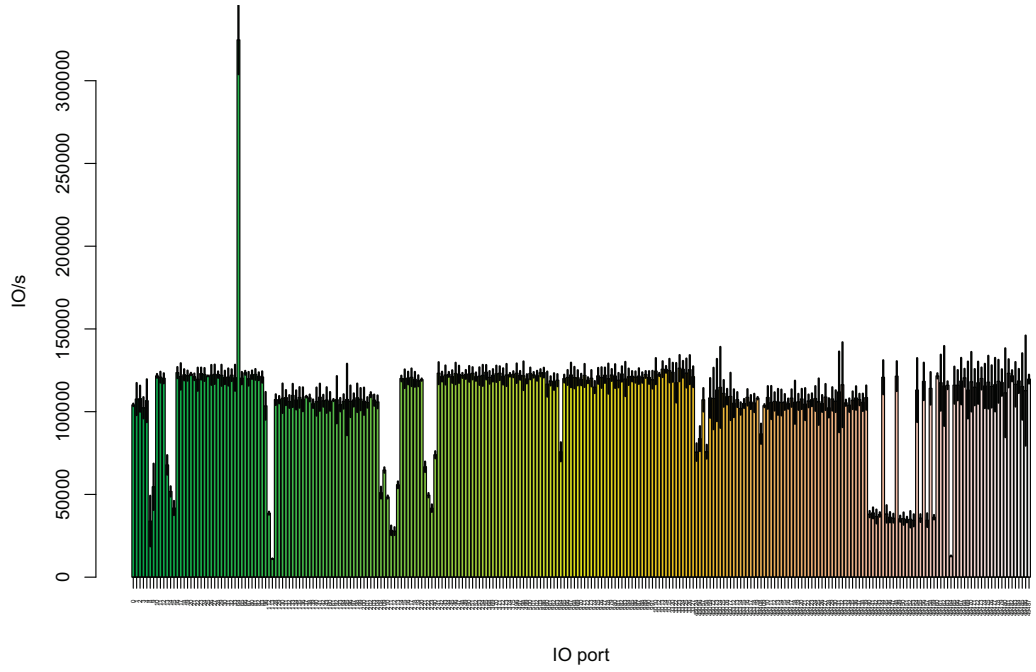


Figure 35: Input per second for valid I/O ports.

Second, the indirect impact on the hypervisor. Some bugs are triggered a long time after the effective sequence of interrupts. For example when a malicious memory write target a function only called by a timer, the bug will only appear when the timer finish. The current implementation does not address these types of bugs directly, and have to be explored if we consider the number of this class.

RESULTS The fuzzing sessions unveiled numerous emulation failures that hinder the isolation between VMs and the system stability. Those flaws are gathered in a database as a list of interrupts to avoid, used to feed the tool presented in the next Chapter, KungFuVisor.

5 KUNGFUVISOR

Showing off is the fool's idea of glory

- Bruce Lee

This chapter applies the VESPA self-protection framework (Chapter 3) to the very specific context of hypervisors. It is an opportunity to address the delicate subject of faulty third-party drivers integrated in the core of the hypervisor. The driver code size represents 80% of the hypervisor with some specific drivers that are not tested to their limits. Hence we adapted VESPA to add an interposition layer around KVM drivers and enhance their security.

The organization of the chapter is the following. First we introduce the concept of autonomic hypervisor protection (Section 5.1). Then we define an attacker model who can threaten the hypervisor security in Section 5.2. The adopted model to enhance the security is detailed in Section 5.3. We describe the implementation in Section 5.4, and evaluate the viability of our KungFuVisor solution in Section 5.5.

5.1 CONCEPT

This section describes the idea of an autonomously protected hypervisor integrating our VESPA solution.

5.1.1 *Protecting The Hypervisor*

We highlight the problem posed by current hypervisors and why existing solutions do not fulfil autonomic properties as defined in Section 2.3.1.

5.1.1.1 *The Problem*

The virtualization layer, foundation of a cloud infrastructure, is particularly vulnerable to powerful attacks based on shared resources. Subverting a hosted VM or the hypervisor may lead to breaking VM isolation, giving the attacker complete system control. So far, most of the attention has focused on protecting VMs. Unfortunately, the corresponding solutions become ineffective in case of hypervisor compromise, as they assume a trusted VMM. The true challenge lies therefore in protecting the hypervisor layer.

Several recent attacks [70, 122] show that the main threat to hypervisor isolation breakout comes from buggy or malicious device drivers inside the hypervisor: kernel exploitation is enabled by poor driver confinement. We have also seen how to gather knowledge on hypervisor interrupts handling and greedy behaviors in Section 4.3.

5.1.1.2 *Limitations of Existing Solutions*

To attempt to solve the problem, a variety of techniques were proposed. For instance, driver virtualization achieves strong isolation, but does not address protection of the virtualization layer underneath [203].

Trusted computing architectures provide strong guarantees regarding hypervisor code integrity [25]. Unfortunately, they usually only detect integrity violations, and do not include remediation operations. Integrity checking is also generally static – dynamic monitoring throughout the system life-time being much harder to achieve.

Driver sandboxing has also been heavily explored: a reference monitor mediates access between driver and device, kernel, or userland [83]. However, solutions remain limited to simple confinement, proposing no actions to sanitize the kernel. Security policies are also often hardcoded in the interception mechanisms themselves. Dynamic, reactive protection strategies are thus difficult to set up, as policies must be configured and updated manually.

To reduce the attack surface further, new designs towards componentized hypervisor security architectures also aim to strengthen driver isolation. However, they often require extensive code rewriting, making them hard to apply to most legacy hypervisors [192, 54].

Overall, current hypervisor architectures offer no – or at best rudimentary – protection for the VMM layer. Previous attempts suffer from: (1) **static, hard-to-manage security policies, not well separated from enforcement mechanisms**; and (2) **no remediation against threats**.

5.1.2 *KungFuVisor Overview*

To overcome the previous limitations, we introduce KungFuVisor, an autonomic security management framework for building self-defending hypervisors. This framework allows to set up several control loops to regulate hypervisor protection, with detection, decision, and reaction steps.

5.2 ATTACKS

5.2.1 *Threat Model*

The attacker may have arbitrary control over VMs. We assume tamper-resistant hardware and associated firmwares (CPU, BIOS), and boot-time hypervisor integrity. However, VMM device drivers may be flawed, and thus tampered with to exploit a VMM vulnerability. A typical bounce attack scenario sourced from a VM might be: (1) perform VM isolation breakout through a buggy VMM driver; (2) alter and subvert the driver; (3) from there, compromise other parts of the VMM and co-located VMs. Such exploitation may for instance result into: rootkit injection with inter-VM traffic sniffing over the hypervisor vSwitch.

5.2.2 *Flawed device drivers*

We detail 2 exploits aimed at escaping the isolation provided by the hypervisor. The first exploit, CloudBurst [122], breaks out of the VMware ESX hypervisor. The second one, Virtunoid [70], breaks out of the KVM/Qemu hypervisor.

5.2.2.1 *CloudBurst exploit*

This exploit relies on the ability to read and write memory out of the allocated Memory Mapped I/O (MMIO) ranges for video emulation. A first MMIO address space carries the frame buffer pixels. A second one is a FIFO storing commands to be interpreted by the host. These two address spaces are shared between the host and the guest. However, parsing the FIFO includes a signed operation, causing an out-of-bound reference to the frame buffer pixels.

Using a wrong source to move to a legitimate destination enables arbitrary write (mov hackersrc => dest), while using a wrong destination enables arbitrary write (mov src => hackerdest).

5.2.2.2 *Virtunoid exploit*

The bug used to escape KVM isolation is that the later does not ignore guest requests to unplug components. Indeed, some components, such as the motherboard, cannot be hot plugged. Doing so leaves an inconsistent state and dangling pointers. N. Elhage chose to unplug the Intel PIIX4 chip because it also unplugs all emulated ISA components. However, the callback registered by the Real Time Clock (RTC) device are fired timely into the Qemu internal loop and provide an exploitable use-after-free condition.

Furthermore, the attack is based on Qemu timer callbacks to make a reliable exploitation instead of more traditional Return-Oriented Programming (ROP) exploitations.

5.2.3 *BluePill attack*

The first piece of software to ask for hardware virtualization is called the **virtualization root** (VMX-root). Thus, if an attack targets the virtualization root and virtualize the running OS on-the-fly, it benefits from the usual hypervisor isolation protection to remain stealth and resident. This attack is considered as hypervisor-layer spoofing.

We have reviewed the state-of-the-art on detection of virtualized environments. From those results we allow our protection framework to protect against layer spoofing and provide an extra sanity check. Section 2.2.5 details numerous techniques to threaten the hypervisor isolation.

5.2.4 *Cold virtual drives attack*

Another common attack is to modify the virtual hard drive of the VM using the common network share for disks. Current network file systems suffer from a variety of drawbacks which go against security. The shares are usually on a dedicated (V)LAN. All hypervisors are using it as a common, fast, resilient hard drive to store VM virtual hard drives. However, this shared network drive has to be readable and writable to all hypervisors. If one of them is vulnerable the whole infrastructure is endangered.

5.3 MODEL

5.3.1 *Design*

Our protection framework operates through well-defined interception points (*hooks*) in the different hypervisor layers. KungFuVisor hooks mediate interactions between device drivers, devices, VMs, and other hypervisor data structures. Thus, dynamic monitoring (*detection*) and access control enforcement (*reaction*) over communications between the driver and its environment may be achieved. It also enables easy integration into most hypervisors, provided that defined hooks are available. Note that containment is not limited to memory-based isolation (e.g., using processor-related mechanisms such as the IOMMU [192]): enforced reaction policies may apply to other communication channels between the driver and its environment to cover a large spectrum of known exploitation techniques (I/O, bad CPU emulation, and other presented in Section 2.2).

A security management plane provides a unified view of the *decision* logic. This plane contains orchestration facilities to realize elaborate detection and reaction patterns – both in each layer, and across layers, and between computing and networking views of resources [220].

This design brings two main benefits: (1) self-managed hypervisor security automates policy administration, allowing dynamic enforcement of flexible driver isolation policies; and (2) coordination of multiple autonomic security loops enables to trigger a rich set of remediation actions over different parts of the hypervisor.

5.3.2 *Hypervisor Model*

5.3.2.1 *A 3-Layered Model*

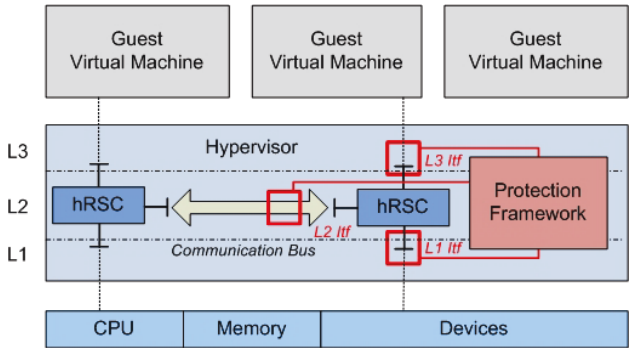


Figure 36: A 3-Layer Hypervisor Model.

We consider the generic 3-layered model shown in Figure 36 for the hypervisor architecture.

Layer 1 (L_1) contains the state of hardware computing and networking resources: CPU, physical memory, and devices (storage, network card). *Layer 2* (L_2) contains the hypervisor-level view of L_1 resources, known in KungFuVisor as *hRSCs (hypervisor ReSourCes)*: virtual CPU, host OS virtual memory, and above all device drivers. hRSCs are the weak point of hypervisor security, and should therefore be sandboxed and sanitized carefully. *Layer 3* (L_3) contains a number of services delivered by the hypervisor

to VMs in the form of hypercalls, such as exposing or modifying the state of a given hRSC (e.g, a vNIC security configuration).

5.3.2.2 *Interfaces*

Each hRSC communicates with adjacent layers through 3 interfaces. The L_1 *interface* is used for instance to handle hardware interrupts. The L_2 *interface* allows the hRSC to interact with other hRSCs through an abstract *Communication Bus* capturing internal hypervisor (e.g., IPCs such as signals, shared memory, or sockets) or vSwitch-level communications. Finally, the L_3 *interface* connects the hRSC with VM resources through specific stubs in the hypervisor.

5.3.3 *Protection Framework*

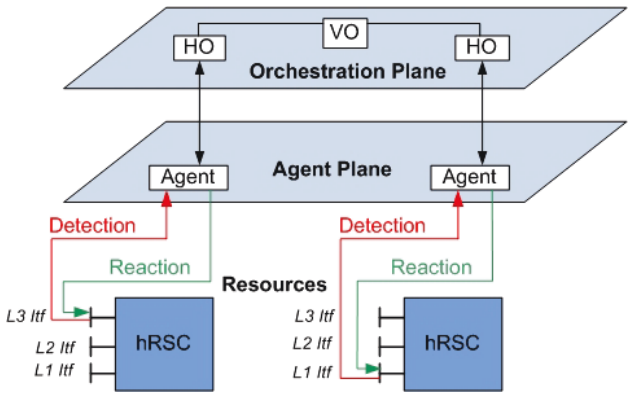


Figure 37: KungFuVisor Self-Protection Architecture.

5.3.3.1 *Multiple Loops*

Hypervisor self-defense is achieved through a set of autonomic loops operating over a number of components organized into 3 planes (see Figure 37). At the bottom, a *resource plane* contains the hRSCs to protect. Over it, an *agent plane* containing a set of *agents* is defined for performing detection and reaction over hRSCs. At the top, an *orchestration plane* coordinates decision-making between self-protection loops.

5.3.3.2 *Monitoring and Reaction*

Agents are wrappers around hRSCs which mediate communications over specific hRSC interfaces through the framework hooks, to monitor activity (e.g., detect malicious invocations), or to perform reactions (e.g., forbid access to an interface). The framework is agnostic with respect to detection and reaction components: such dedicated components can be plugged-in to mitigate specific attacks. For instance for detection, any type of lightweight IDS (signature-, anomaly-, or classifier-based) monitoring the L_x interface functions and parameters. And for reaction, firewalling outgoing L_x calls, or cleansing the driver by internal state modification.

Our approach is similar to system call interposition (SCI) but for interrupts. The benefits were studied in [84, 165, 229]. We profile a running system supposed non-infected by gathering interrupts patterns. For example during boot, patterns appear

with tens of different I/O ports and values that are added to an interrupt white list. On the other side, we maintain a blacklist of patterns derived from the fuzzing framework knowledge. When such interrupts are detected, KungFuVisor modifies the interrupt vector on-the-fly. Either by sanitizing values with legitimate one, i.e. known to have no impact, or discarding the request silently if no other option is available.

The overall knowledge is always updated during system runtime to register more patterns. KungFuVisor meticulously replays patterns on a sandbox VM to extract the problematic interrupts, and can either go through manual analysis to approve the final list selection or add them automatically.

5.3.3.3 *Decision-Making*

The self-protection decision logic is split between two types of *orchestrators*. Each hypervisor layer L_x contains a *Horizontal Orchestrator (HO)* providing a layer-view of security management. The HO is a simple autonomic security manager performing a reflex, local response to threats targetted at a specific set of hRSCs incoming and propagating through the L_x layer interface. The HO supervises agents attached to the L_x interface of monitored hRSCs, aggregating collected information, and dispatching chosen reactions.

A *Vertical Orchestrator (VO)* realizes higher-level, wider spectrum security reactions. By evaluating information provided by HOs in each layer, the VO coordinates layer-level decisions in order to provide a consistent, cross-layer response to detected threats.

Orchestrator interplay results in a very flexible self-protection model allowing to enforce a rich continuum of remediation strategies, both within and across layers, and between computing and networking views of resources. For instance, suspicious behavior (e.g., memory tampering) detected in a computing hRSC monitoring the hardware layer may trigger reactions over networking hRSCs in another layer (e.g., disable a vNIC).

5.4 IMPLEMENTATION

5.4.1 *Mapping*

The protection framework is mapped to the hypervisor model by setting all entities of management and orchestration planes directly into the hypervisor. Specific hooks then connect agents to the relevant hRSC interfaces. This design limits the attack surface, as all framework entities are in the hypervisor itself, without interfaces presented to the outside (i.e., no backdoors). Moreover, it reduces the impact on legacy hypervisor code, as agents have the same external interfaces as hRSCs. The performance overhead is also expected to be minimal, as the framework code is interfaced to hRSCs using simple function calls.

Furthermore, we enhance the protection of the KVM hypervisor security by applying well-known protections of Windows exception handlers. Static list of timers shield them and prevent the new exploitation class introduced by Nelson Elhage [70].

We also verify if the Qemu program is PIE-compiled with a strong ASLR setting. This provides another effective protection layer against state-of-the-art exploitation. ROP attacks requires some position knowledge to locate the gadgets, unavailable here as all addresses are randomized.

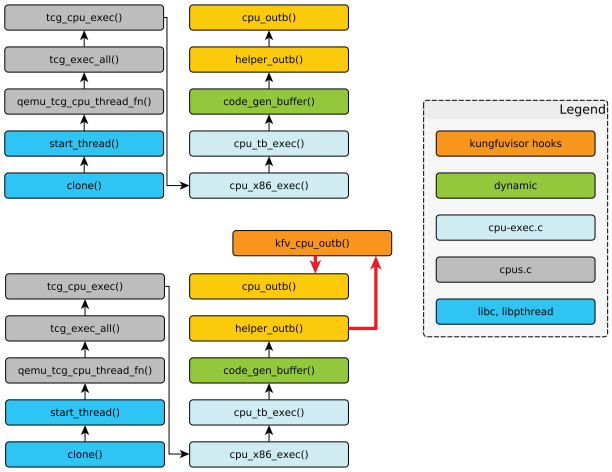


Figure 38: KungFuVisor hook for outb on Qemu.

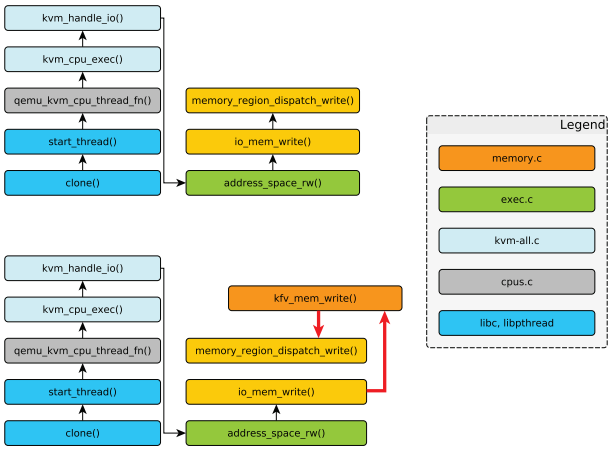


Figure 39: KungFuVisor hook for memory_write on KVM/Qemu.

5.4.2 *Qemu hooks*

Close to the VM introspection, we wrap the Qemu function closest to the IN and OUT instruction processing interrupts: cpu_in* and cpu_out*.

The processing flow for an *outb* instruction through Qemu is detailed on Figure 38. The *libc* and *libpthread* libraries create a thread (in blue) for parallel processing on SMP. Then the Tiny Code Generator (TCG) parses the instruction to emulate, say outb. The instruction architecture defines the execution handler to convert the instruction into the Qemu Intermediate Language (IR). Our example is for the Intel x86 instruction set, so the *cpu_x86_exec* is called. The instructions are transformed into a Translation Block (TB) caching further translation of the current Basic Block. The *code_gen_buffer* function feeds the TB structure with the result of the *helper_outb* function. The latter build the IR representation of the outb instruction. Thus, we hook the helper functions and redirect the control flow toward KFV agents.

Hardware assisted virtualization hides many critical interactions as the real CPU handles the processing. However, we are still able to hook critical functions (Figure 39).

It allows semantic learning of instructions executed by a VM. We can compare requested I/O with the list gathered through fuzzing and public attacks. If one of them is called, we send an alert to the VO and apply a wide variety of reaction (described in Section 2.2.6) such as ignore, pause or reboot the VM.

```
void helper_check_iob(CPUX86State *env, uint32_t t0)
void helper_check_iow(CPUX86State *env, uint32_t t0)
void helper_check_iol(CPUX86State *env, uint32_t t0)
void helper_outb(uint32_t port, uint32_t data)
target_ulong helper_inb(uint32_t port)
void helper_outw(uint32_t port, uint32_t data)
target_ulong helper_inw(uint32_t port)
void helper_outl(uint32_t port, uint32_t data)
target_ulong helper_inl(uint32_t port)
void helper_into(CPUX86State *env, int next_eip_addend)
void helper_single_step(CPUX86State *env)
void helper_cpuid(CPUX86State *env)
target_ulong helper_read_crN(CPUX86State *env, int reg)
void helper_write_crN(CPUX86State *env, int reg, target_ulong t0)
void helper_movl_drN_T0(CPUX86State *env, int reg, target_ulong t0)
target_ulong helper_read_crN(CPUX86State *env, int reg)
void helper_write_crN(CPUX86State *env, int reg, target_ulong t0)
void helper_movl_drN_T0(CPUX86State *env, int reg, target_ulong t0)
void helper_lmsw(CPUX86State *env, target_ulong t0)
void helper_invlpg(CPUX86State *env, target_ulong addr)
void helper_rdtsc(CPUX86State *env)
void helper_rdtscp(CPUX86State *env)
void helper_rdpmc(CPUX86State *env)
void helper_wrmsr(CPUX86State *env)
void helper_rdmsr(CPUX86State *env)
void helper_wrmsr(CPUX86State *env)
void helper_rdmsr(CPUX86State *env)
void helper_hlt(CPUX86State *env, int next_eip_addend)
```

Figure 40: KungFuVisor hooks in Qemu.

Figure 40 gives the KungFuVisor hooks into Qemu, functions that can be redirected to control communications with the hypervisor.

5.5 EVALUATION

We evaluate the KungFuVisor solution in terms of performance overhead to boot a VM (Section 5.5.1), and the added attack surface with a security analysis (Section 5.5.2).

5.5.1 *Performance overhead*

Several problems appeared using the Python implementation of VESPA seen in Section 5.4.

5.5.1.1 *VESPA components placement*

Putting the VESPA framework in user space requires context switching for every interrupt. Our first tests showed 10000% overhead and required 6 days to boot the VM! However, this solution is not intrusive with only 20 SLoC added to the hypervisor TCB.

Offloading aggregation of interrupts to the hypervisor multiplies by 5 the impact on the TCB size compared with the previous solution. However the performance overhead goes down to 800% and the VM boot in 8 hours. Integrating the agent of the VESPA framework directly into the TCB is far more heavy. Converting the core Node class of VESPA, with about 3000 LoC in Python with introspection would result in at least 10 000 LoC in C and is not realistic. But once the conversion is done, a lightweight version of the framework may be done with agents and security components in approximatively 5000 C LoC given our C code base. Reusing the C version of VESPA suppose the integration of the Cecilia framework directly into the hypervisor. This approach was abandonned. This would result in an even bigger code base.

We fostered the small TCB size as it is simpler to accept a small patch for a low level access if we submit it to the KVM developers.

5.5.1.2 *Lighter interposition*

The overhead with VESPA orchestration plane in user-space is unacceptable as-is. Thus, we considered detection and reaction to be asynchronous and not waiting for the VO response. This means the alert is sent as before but the interrupt is always forwarded to the associated Qemu handler. VESPA is able to perform the alert gathering and the decision in parallel. The major asset of this solution is that the overhead drops to 12%, and the TCB is still minimal. However it come with a major security drawback: if the attack is faster than the VESPA autonomic loop, the hypervisor can be exploited and not recoverable. It is therefore impossible to stick to the asynchronous approach to catch fast attacks. Slower attack can be detected using this approach.

5.5.1.3 *Dynamic interposition*

We chose to adopt a two-level granularity mixing synchronous and asynchronous approaches. The original loop waited for the VESPA response and was not able to handle the I/Os fast enough. We investigated further and analyzed the number of I/O performed by the VM life-cycle (Figure 41). The most consuming part is the boot with the load of many files (disks I/Os) and peripheral setup. Indeed, once the VM is idle and does not perform much I/Os, the system responds in a natural way and the user only experiences some short delays. The I/Os perfomed at $t = 1800s$ is a Web browser session, and at $t = 2600s$ we performed a search with the grep tool on the whole disk followed by a poweroff.

The idea is to make the synchronization adaptable during the VM life-cycle. The VM boot is unadapted to perform the attack, as the system is not fully loaded and there is no network available. It is possible to lock-in the boot with fixed kernels and modules by the provider for more security. This phase is then considered safe and VESPA let the I/O/s be forwarded to the Qemu handler during the analysis in the VO. When the number of interrupts is below the defined threshold, VESPA switches the driver agent to the synchronous mode: all interrupts are filtered. The agent will become asynchronous when the system is under heavy I/O load.

MICRO BENCHMARKS We used the standard benchmark suites to evaluate the impact of VESPA on the system. The LMBench [137] benchmarks system bandwidth and latencies at various levels. Figure 42 compares the times given by LMBench on a vanilla Qemu and on KFV in microseconds. The first eight figures measure *context switching* for 2, 8 and 16 processes and with a work size ranging from 0 to 64KB. For

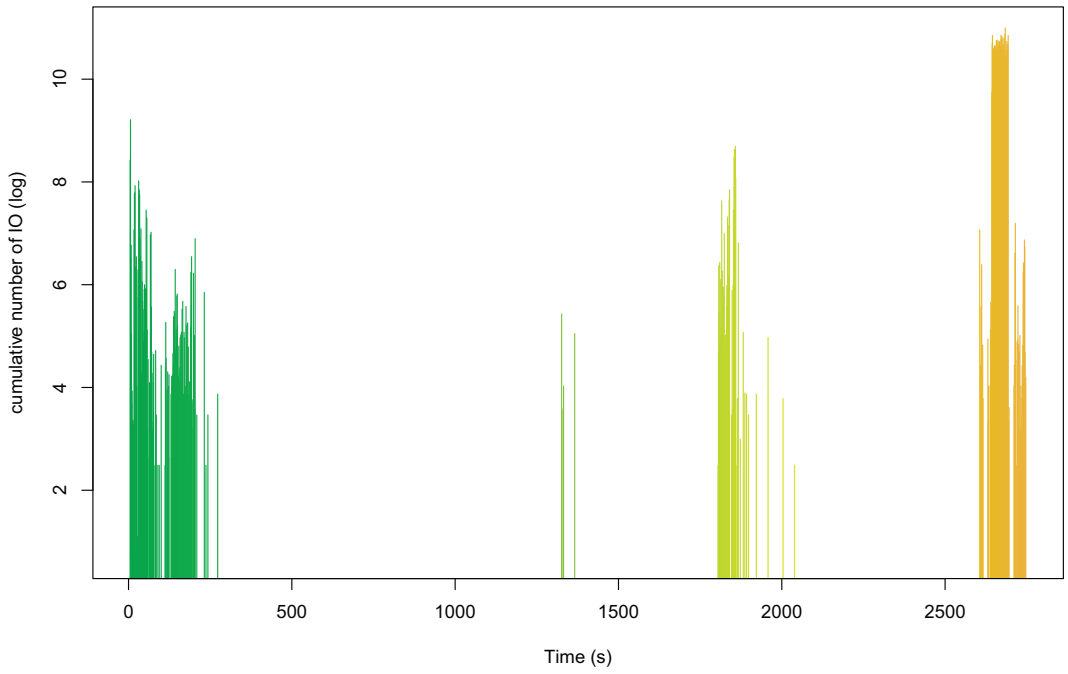


Figure 41: Number of I/Os during a VM life cycle.

example, 2p/0K indicates two processes with no workload that only pass the token to the next process. Then we measured *interprocess communication latencies* through Pipe, AFUNIX, RPC with UDP and TCP. Two processes are created and exchange data. FileCreate and FileDelete indicate the *file system latencies* for the creation and deletion of 0KB and 10KB files. Finally, the ProtFault and PageFault tests illustrate how fast a process and a page of a file can be faulted in. The file is flushed from (local) memory by using the msync() interface with the invalidate flag set. The real overhead percentage is displayed on Figure 43 with the vanilla Qemu as the base.

The results showed that KungFuVisor is able to enhance hypervisor security with a 12% overhead on an average VM life-cycle. This mean supposes that all tests are equally distributed during a VM life-cycle. However the protection fault signal latency is rare compared to others. We were not able to reduce the signal latency and thus it has to be present on the overhead graph, but the global overhead is less than 12%. Finding the exact repartition of categories require an extensive semantic reconstruction that was not performed here.

5.5.1.4 *Optimizations*

The list of faulty I/Os given by the fuzzing use case is cached into the driver agent as a radix tree with a depth of 8. It enables fast lookup directly into the kernel with an added 30 SLoC. The interrupts are compared, matched and forwarded.

The communication with VESPA was also optimized to reduce the overhead. The communication protocol is UDP with a thin layer of error correction code. Data are neither compressed nor encrypted as it increases the number of CPU cycles per I/O.

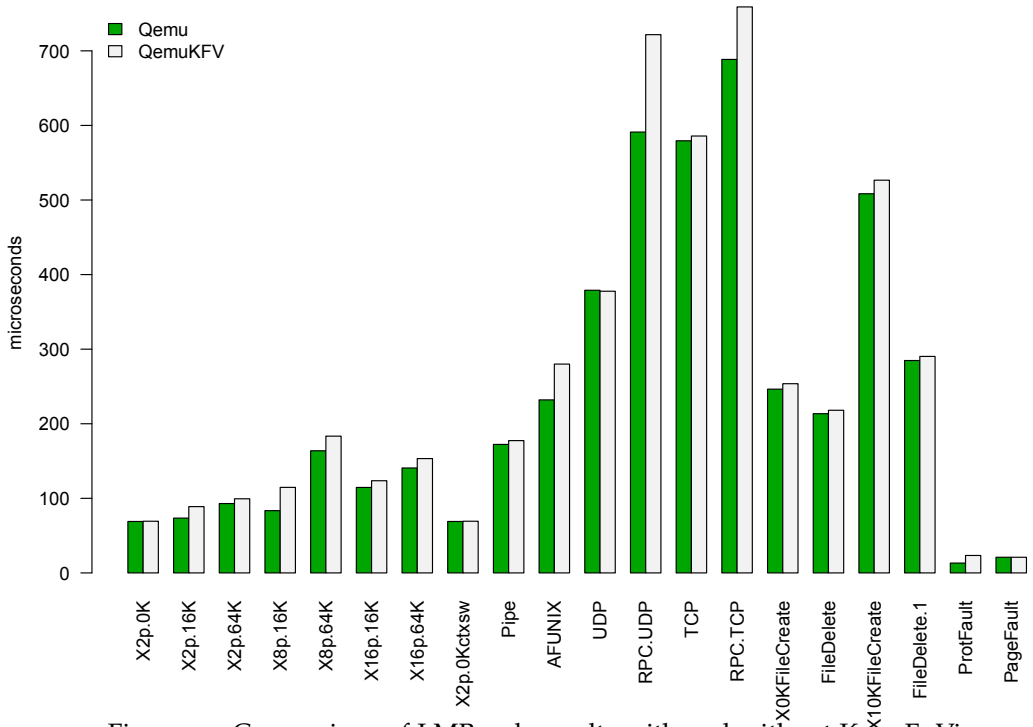


Figure 42: Comparison of LMBench results with and without KungFuVisor.

5.5.1.5 *Discussion*

Our current implementation is adapted for critical systems with an acceptable overhead and an enhanced security. The threshold to switch between synchronous and asynchronous has to be carefully chosen. If too high the system will hang and provide bad user experience, if too low security is lowered. The administrator has to define VM security levels and associate a predefined threshold. Trusted VMs will remain mostly asynchronous while untrusted VMs will be monitored synchronously.

5.5.2 *Security analysis*

5.5.2.1 *Mitigation of public threats*

The CloudBurst attack was successfully prevented by extracting the I/Os of the public exploit. However the memory corruption targets the VMWare 3D driver that is obviously not present on KVM. A fake Qemu driver emulates the original driver for our tests and give an overview of the behavior. The real impact requires to modify the I/O handling routine of VMWare hypervisors, which is not open source. Thus, the sequence of interrupts may need to be polished in the exploitation environment.

The Virtunoid attack was prevented on the latest version of KVM vulnerable to this attack. The sequence of interrupts was extracted and filtered, protecting the hypervisor without patching. Such feature is important as it is a way to protect old hypervisors from public attacks that do not have fixes. The administrator analyzes the exploit, runs it on a VM and adds the signature to the VESPA database. With further tests, it is possible to adopt KungFuVisor as a long term hypervisor that can be externally fixed.

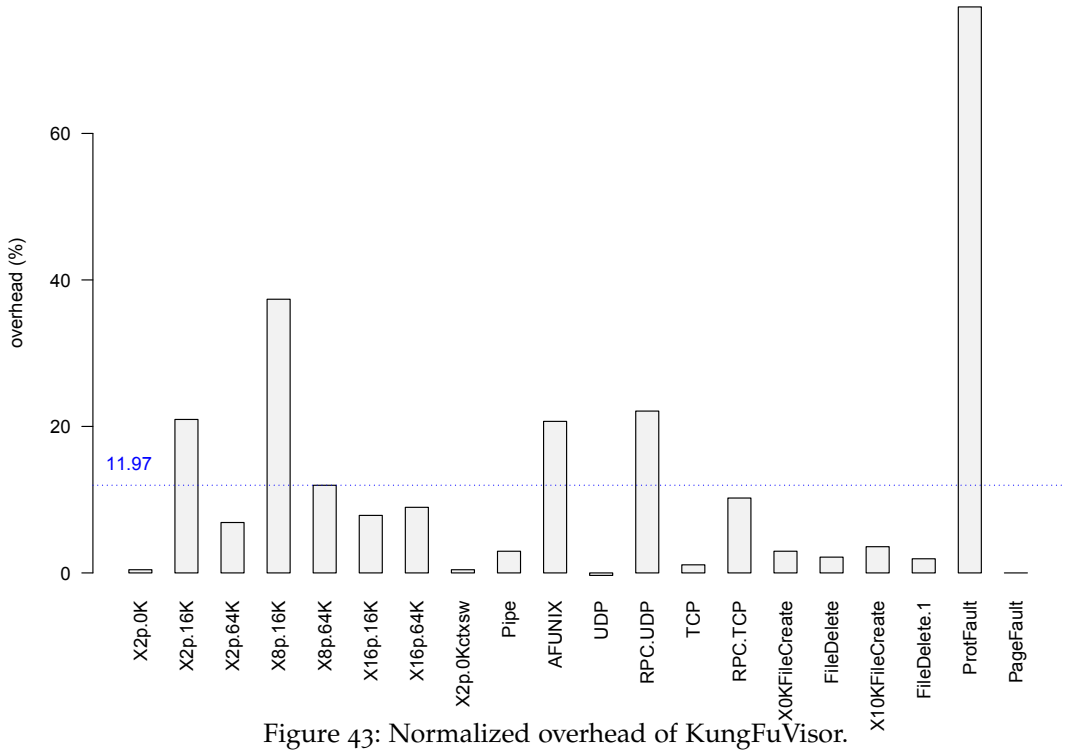


Figure 43: Normalized overhead of KungFuVisor.

A public exploit does not hinder cloud security and can be fixed without restarting the physical servers.

Three other attacks generating I/Os were prevented, but vulnerabilities regarding hypercalls remain exploitable.

5.5.2.2 *Unexpected behaviors*

In the previous chapter, we accessed the interrupts attack vector to stress drivers and extract unexpected behaviors. Our framework can be easily adapted to handle new attack vectors. The "stress" module, agent_gen_fuzz, is a wrapper around the C library for interrupts. For example, moving to syscalls to stress the kernel can be achieved by replacing the *outl* with the syscall assembly, while fuzzing the CPU instructions may require more work with assembly parsing libraries. With the fuzzing sessions, KungFuVisor is able to prevent bad emulation from Qemu. Loops, invalid writes and segmentation faults are added dynamically in the VESPA database.

The new components integrated by KungFuVisor are disaggregated to diminish the impact of compromise. The VESPA framework, which is the core of KFV, gathers knowledge and orchestrates decisions against interrupts. Thus an attacker may hijack the knowledge by detecting VESPA. Taking advantage of VESPA to setup an attack requires access to administrator policies, and to find a logic hole inside. Although, VESPA and KFV do not guarantee the formal security of policies.

5.6 CONCLUSION

This chapter described how we instantiated the VESPA framework to the hypervisor context, and showed that the adaptation require subtle trade-offs. While our evalua-

tion showed mitigated results for a straightforward integration, we believe that it is an acceptable solution with some tuning. The current public exploits were successfully prevented and the parallel fuzzing of drivers unveil unknown flaws dynamically. Mixing both defense and attack provide a new approach for third-party code that is not well-tested.

Integrating VESPA functionalities into the hypervisor is costly and increases the TCB but will also increase performance. A lighter version of the framework without the unused functionalities (policy frameworks, dynamic reconfiguration, multi-domain) is a good candidate for further research into this layer.

6 CONCLUSION

Don't cry because it's over, smile because it happened.
Dr. Seuss

This chapter summarizes contributions, and analyzes how they fulfill the objectives defined at the beginning of the thesis (Section 6.1). Then we detail limitations and foresee the future work to enhance our architectures (Section 6.3).

6.1 MAIN RESULTS

In this thesis, we showed that it is possible to model the cloud security in terms of framework components. Four design principles were defined to overcome the limits of current cloud platform: policy-based self-protection, cross-layer defense, multiple self-protection loops and open security architecture. This approach enables the setup of flexible cloud infrastructure security, from static local security to dynamic multi domain isolation.

The VESPA model was validated using two implementations in Python and C. Those frameworks expose components available at several levels of granularity: the developer can interact with the API of a complete solution, or create a link with a precise part of a system component. The composition of heterogeneous building blocks gives the opportunity to empower the IaaS infrastructure à la carte. Numerous use cases were implemented to underline the potential of VESPA, showing simple and fast adaptation in various environments. Also, the performances are not outdone with a small overhead.

The extension of VESPA to the hypervisor allowed the quantitative evaluation of the components. KungFuVisor required several adjustments to become a viable solution with enhanced security. The hypervisor is more robust against publicly disclosed attacks and malicious sequences of interrupts. The challenge was successfully answered, highlighting how VESPA is adaptable. It required architecture design and tests to deliver good performances, but the basic components, communication and policies does not need to be changed. We argue that this is a promising security framework with a modular design that answers multiple security problems of today clouds.

Table 19 summarizes principles and objectives fulfillment as defined in Section 1.2. The analysis of each design principles follows.

6.1.1 *Policy-based self-protection*

Our VESPA architecture displays a clean policy interface. However, the integration of policy models involve specific development. The example of RBAC [77] shows that we have to assign objects to roles, and the OrBAC model stick resources to organization. These concepts are not fully handled by VESPA, and policies are expressed as Finite State Machines. The OrBAC support is partial, user assigning resources to domains, which are close to the organizations defined into OrBAC.

Design Principles \ Solutions	VESPA	KungFuVisor
Policy-based self-protection	✗	✗
Cross-layer defense	✓	✗
Multiple self-protection loops	✓	✓
Open architecture	✓	✓
Multi-Cloud	✓	✗

Table 19: Extended Properties Fulfillment

It is not enough to fulfill our policy-based self-protection requirement, even if the policy interface is defined.

6.1.2 *Cross-layer defense*

The design of VESPA integrate the cross-layer defense into the architecture, validating our requirement. The VMs, hypervisors and physical devices are able to communication securely through strict interface definition.

However, KungFuVisor is specific to the hypervisor layer. It does not support collaborative defense with other hypervisors. The KFV architecture adapts the VESPA layers to the hypervisor code context, and remove cross-layer defense as defined into the requirements.

6.1.3 *Multiple self-protection loops*

VESPA provides multiple self-protection loops if the associated implementation follows the architecture design. Use cases and KFV inherits this requirement. The administrator is able to interconnect the security element of the infrastructure in multiple ways. Each loop is usually assigned a specific security function, e.g. to prevent network intrusions, isolation breakout or denial of service.

We expect loop mushrooming with new developers coming into the project. New components will be created and new loop patterns will enhance the current implementation.

6.1.4 *Open architecture*

The open architecture requirement is fulfilled in multiple ways.

The interface for communication is explained into this thesis and in further details into the code documentation. It is flexible and adaptable for new needs and tests. The internal interface for Remote Procedure Call (RPC) is also provided to integrate new languages seamlessly. The dynamic configuration with code introspection is divided into independent functions. This feature is hard to port, and developers can adopt other ways to find available attributes into a node.

All components of the VESPA architecture and use cases are available under LGPL on github and orangeforge. Developers can modify, extend and correct the available nodes to match their needs.

6.1.5 *Multi-Cloud*

The OC2 use case allows the VESPA framework to communicate with other instances. The Vertical Orchestrators (VOs) can alert each other though a peer-to-peer communication, and react accordingly. This is the extension of multiple self-protection loops to heterogeneous domains.

As previously stated, KFV is not able to perform negotiation with other hypervisors. We argue that it can be considered as a backdoor into a critical component.

6.1.6 *Conclusion*

The VESPA (Chapter 3) self-protection architecture proved cross-layer defense to be viable through the different use cases described in Chapter 4. Multiple agents have been developed at the VM layer and hypervisor layer and are able to feed the VO knowledge.

The KungFuVisor (Chapter 5) adaptation of VESPA to the hypervisor underlined the adaptability of our self-protection architecture to a very specific context.

6.2 LIMITS

Obviously, the work realised in this thesis is not complete. Several research have to be conducted to build the completely secure cloud infrastructure.

First, the policies used in the framework are twofold: (1) machine state in Python for the simplicity and adoption; (2) OrBAC to fit to domains. These are pragmatic, generic and straightforward policies for distributed contexts. However finer policies can handle subtle cloud-related oddities, such as isolation policies on different hardware. When a user wants to migrate its VM, how to abstract a single policy and provide the same isolation level on the destination ? The VESPA framework provides elements of answer with the agent hierarchy and the associated refinement policies, the two of them not being fully exploited.

Second, the physical layer was not studied in-depth. The VESPA framework may thus be missing some properties specific to this layer. Yet, physical security equipments are always present on the provider infrastructure. It is therefore a layer to integrate to reuse available security equipments.

6.3 FUTURE WORK

6.3.1 *Physical layer*

Although the physical layer was not exploited in this thesis, we still need to handle physical agents to reach the maximum potential of our framework. While not really a limitation, there is some work needed to gather physical management interfaces, such as the libvirt for hypervisors. There is no common base to manage physical equipment and it usually results in a specific agent for a single equipment.

Also, physical attestation components can be used to measure the VESPA framework integrity and protect it against memory corruptions. As a quick answer, it can be done using the TPM or the Trusted Execution Environment (TEE) modules that ship with current computers and phones. However those components already exposed sev-

eral weaknesses and the question of the trust given to the chip makers remain. If a solution emerges, VESPA will have to adopt it to ensure that components are deployed on the right execution environment.

6.3.2 *Framework protection*

VESPA components are not designed to be tamper-resistant. The implementation of the architecture does not provide protection against code modifications.

The self-protection framework aims at protecting existing heterogeneous resources, but the framework components protection was not addressed directly. Communications between security layer, agent layer and orchestration layer are encrypted to guarantee confidentiality, and signed to guarantee integrity, but availability remains the weakness. The framework does not support the lack of the VO, while HOs should try to protect at least the underlying agents. This issue was not addressed either.

However, each instantiation can define self-protecting nodes. The KFV hypervisor performs checksums on both the program during the startup and live when running. Some approaches offloaded the Qemu component as non-privileged to limit the impact while being exploited (DeHype, Nooks). DeHype splits the KVM TCB down to 2.3K LoC, which is a reduction of about 94%. However, the remaining code is copied into each virtual address space and consumes memory. In terms of security, the VM running on the unprivileged hypervisor will crash if it is exploited. The physical machine is still able to reboot the compromised part as a remediation. With KungFuVisor, we protect the hypervisor with a new mechanism generating bad interrupts and filtering them before they compromise the system. The memory is not duplicated and the original KVM code does not need to be redesigned. The IPC problem encountered during KungFuVisor design was addressed using hardware facilities. The VMCS traps I/Os by filling the correct structure, but given the results showed in Section 2.2.5, hitting the VMCS is costly and is not adapted to intense I/O loads.

The self-stabilization [65] can provide better protection to our VESPA architecture. Models used by this approach consider several correct states for an execution. If the actual state derives from a stable point, remediations are applied to move toward the closest correct state. VESPA execution can be secured by giving correct states to the self-stabilization layer and run under its control.

6.3.3 *Multi-Cloud*

Cloud administrators are looking for an abstraction to interact with the variety of cloud providers. The interconnection of heterogeneous clouds is not ready yet, but is under active research. This issue is likely to become a tough question in a near future.

It raises multiple question regarding the VESPA framework. New policy attributes, more complex negotiations among VOs, the associated decisions and trust management are just but a few questions to answer.

6.3.3.1 *Nested virtualization*

With the trending nested virtualization, new challenges raised and were not heavily studied in this dissertation. The generic VESPA framework is not limited by the number of layers, but more work is needed to tackle down new implications. Some work has been done [21] to provide a first security architecture compatible with VESPA.

6.3.3.2 *KungFuVisor*

We addressed the vulnerabilities related to the hypervisor interrupts. However there are other attack vectors to weaken the hypervisor isolation. For example the component loading VM configuration files may be vulnerable to malicious strings attacks. The user is able to define the machine name and information that will be parsed by the hypervisor. If the latter does not properly escape the strings, it can provide a malicious root access. A composed approach between VESPA and the architecture design of Xen disaggregation is thus needed to limit the attack vectors.

Part I

APPENDIX

A NINE CLOUD COMPUTING SECURITY ROADBLOCKS

We identify nine major roadblocks of cloud computing security to capture the security barriers to adoption of cloud infrastructures [221]. Those roadblocks are classified into the 4 main areas shown in Figure 44: *local security* (protection of computing resources), *network security* (protection of communications), *data protection* (protection of storage), and other *trust enablers*. Their criticality may be qualified using the following scale:

- *Critical:* lifting the roadblock is essential for adoption, resulting in a breakthrough if successful.
- *Important:* lifting the roadblock will be a major step forward.
- *Incremental:* lifting the roadblock is possible by natural enhancement of already existing technologies.

In what follows, we discuss for each area the corresponding security roadblocks.

A.1 LOCAL SECURITY

This area deals with the protection of servers which compose a data center. The main issue is how to guarantee security when computing resources are virtualized, i.e., as VMs running above a hypervisor on each host. The key roadblock is thus naturally the *security of the hypervisor*.

A.1.1 *Hypervisor Security*

Virtualization introduces many security vulnerabilities. Clouds are by essence multi-tenant environments: the crux is thus strict isolation between VMs, which may fail if the hypervisor is compromised. In theory, the most widespread hypervisors have a relatively low attack surface. In practice, new variety of attacks [169, 174] such as installing rootkits inside the hypervisor (*hyperjacking*) or using covert channels call for higher degrees of assurance [172]. The main weaknesses are misconfigurations, malicious device drivers, and backdoors between the VM and the hardware, issues for which there are today no real answers. Hypervisor security is only part of the problem since VMs may also bring their own set of vulnerabilities: these may be mitigated using hardened VM images, or strict VM security life-cycle management. In all cases, "security by default" configurations should be applied, with clear delineation of responsibilities between customer and cloud vendor.

A.2 NETWORK SECURITY

This area deals with the protection of communication channels to access or inside a data center. The key issue is how to guarantee security when networking resources are virtualized, i.e., firewalls, IDS, routers run as virtual appliances. The main roadblocks here are *network isolation* and *elastic security*.

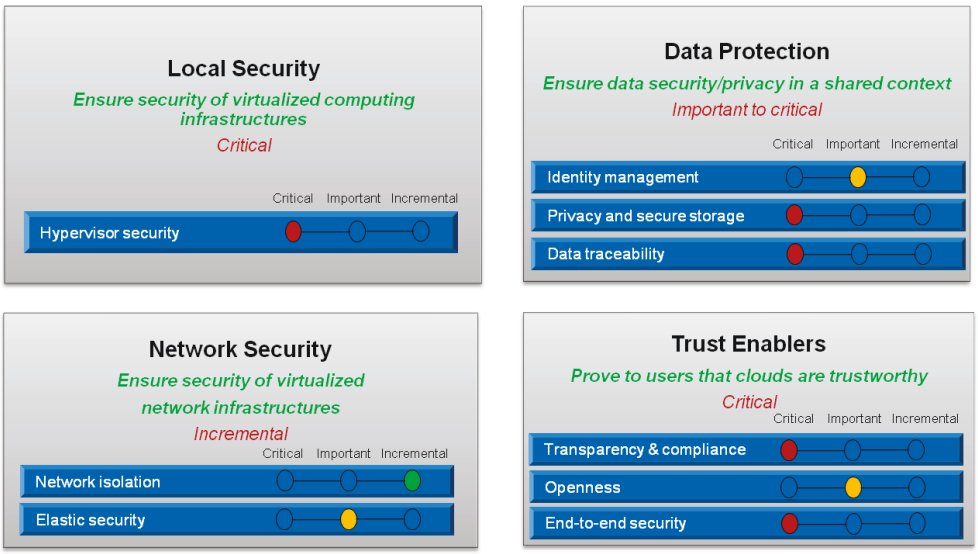


Figure 44: Major Security Roadblocks of Cloud Computing.

A.2.1 *Network Isolation*

In a data center where some of the communication links may be fully virtualized, are traditional network security architectures still effective? Where security controls should be placed? The risks are broadly comparable to those of traditional networks (confidentiality/integrity of network connections to/in clouds, Authentication, Authorization and Accounting (AAA), availability). The corresponding counter-measures (encryption, digital signatures, Network Access Control (NAC), VPN, Network IDS (NIDS), Network IPS (NIPS), ...) are thus still applicable. The main change is that network isolation is no longer physical but logical: network zones where traffic could be segregated physically (e.g., to separate production from supervision hosts) are replaced with logical security domains, where traffic between VMs is filtered by "virtual" firewalls. As a result, isolation is less precise, and the security guarantees weaker. Overall, the technical security components are available today to lift this roadblock. However, the main difficulty is to map them to cloud architectures.

A.2.2 *Elastic Security*

Flexible allocation and rapid provisioning of security resources able to respond to dynamic evolutions in the cloud is still a challenge due to the high rate of change in virtual servers. First solutions are emerging for flexible and dynamic management of VPNs, with the notion of virtual private clouds. However, fully automated security supervision of cloud infrastructures is still lacking due to the complexity and short response times needed to manage vulnerabilities, detect intrusions, and activate defenses. Research initiated by IBM on autonomic, self-protecting security architectures should enable to build infrastructures where security is self-managed, security parameters autonomously being negotiated with the environment to match the ambient estimated risks and provide an optimal level of protection [94].

A.3 DATA PROTECTION

This area deals with the security and privacy of data (at rest and in transit). The main issue is how to guarantee such security in a shared, multi-tenant environment. Key roadblocks are both technical such as *identity management*, but also non-technical, such as *privacy/secure storage* or *data traceability*.

A.3.1 *Identity Management*

The number and diversity of principals using cloud services internally and externally, and the volume of resources accessed call for end-to-end solutions for managing identities. Unfortunately, cloud infrastructures are still lacking consistent identity information architectures: multiple administrators, credential repositories, and processes which are neither automated nor orchestrated may induce new vulnerabilities. Barriers are thus scalability, heterogeneity and interoperability. An unsolved problem is how to achieve federation of identities across data centers, organizations, or cloud providers to avoid duplications of identities, or privileges between information systems.

While the basic mechanisms already exist to exchange identity claims, how to integrate them in the cloud context is still unknown. If authentication challenges might be overcome in the near future, uniform management of authorizations is still in its infancy despite standards like XACML. Nonetheless, identity management is one of the main opportunities for Security-as-a-Service in cloud infrastructures, to outsource authentication and authorization components from IT systems.

A.3.2 *Privacy and Secure Storage*

Today's privacy concerns will be magnified in cloud environments due to technology sharing in a multi-tenant context. The main challenge is thus strong data isolation throughout the life-cycle of personal information. Difficult problems include data access enforcement on a need-to-know basis, secure data storage and information flow control, and data retention and destruction. While standard cryptography is still applicable to protect data at rest and in motion, today's privacy best practices are not enough to fully address this roadblock. Possible technical answers include self-destructing data [88], "sticky" (i.e., directly attached to the data) privacy policies, and more generally promoting "privacy by design". In any case, responsible data stewardship in the cloud requires both in depths understanding of possibilities of Privacy-Enhancing Technologies (PETs), and of legal implications in contractual agreements.

A.3.3 *Data Traceability*

Adding further to the loss of control of user over their data, locating the data itself is a major concern in a shared and virtualized infrastructure: at a given time, a cloud provider might not know exactly where (i.e., in which country) data is stored, processed, or accessed from. Without special care, data could move freely around between organizations, or even between international borders. This raises legal and political issues, since several jurisdictions (EU Data Protection Directive, US Safe Harbor Pro-

gram) specifically require the provider to have such knowledge. Data hosted abroad might also be exposed to foreign governments (USA Patriot Act). Data traceability is also needed to prove to users that data comes from a trusted source. Overall, this domain is still a widely unchartered area.

A.4 TRUST ENABLERS

This last area is perhaps the most important one, since the main issue is how to prove to customers that their cloud infrastructure is trustworthy. The main roadblocks are *transparency and compliance*, *openness*, and how to guarantee *end-to-end security*.

A.4.1 *Transparency and Compliance*

Unlike traditional environments, in the cloud, the nebula of stakeholders, logical instead of physical isolation, and transfer of resources outside the control of organizations, all make customers uneasy as where trust boundaries have moved to. A maximum level of transparency is thus required from the vendor to dispel this confusion. Customers need tangible evidence of the security hygiene of a provider infrastructure, to verify security claims of contractual agreements, or compare with other providers practices. Elements of assurance are required by authorities to check compliance with established standards and regulations. Unfortunately, providers remain so far opaque on these aspects.

Auditability of infrastructures have thus to be enhanced, to convince third parties that the necessary detective and preventive security controls are in place. Setting up a certification process helps the provider move forward. Unfortunately, there is no real agreement today on the right assurance framework (SAS 70, ISO 27001,...). Documents published by ENISA [71] or the Cloud Security Alliance [51] facilitate a risk analysis. Yet this analysis remains difficult due to increased complexity and openness compared with traditional computing. Trusted computing technologies [35] also foster trust by providing users cryptographic evidence of infrastructure integrity, but a lot of work remains to be done in this area. In any case, responsibilities between providers and customers have to be established using clear-cut SLAs.

A.4.2 *Openness*

This issue is necessary to overcome vendor lock-in, viewed by the Cloud Security Alliance (CSA) as a top threat. Proprietary, closed, and non-standard-compliant cloud technologies will make it very complex for the customer to change cloud provider, verify the vendor security promises, or react in case of incident. Interoperability with other cloud infrastructures will also be difficult, each vendor having its own APIs. Deployment of applications distributed across several infrastructures will thus be hampered, limiting scalability. Recommendations are to stick to best practices [51, 71] and standards, and push standardization efforts on open APIs. Open source cloud architectures will also bring additional benefits in terms of flexibility (modular architectures) and security (careful code scrutiny by the security community).

A.4.3 *End-to-End Security*

To foster trust, data isolation has to be guaranteed both at rest and in transit in all layers of the cloud infrastructure (processing, network, storage). Unfortunately, the few security building blocks available are highly heterogeneous and fragmented. How to orchestrate them seamlessly into an end-to-end security infrastructure for cloud environments is still undefined. This issue which crosscuts all the previous technological barriers could be overcome by standardizing reference security architectures for cloud environments which describe the organization of the different security components, to provide an overall view of cloud security.

B

QEMU I/O

B.1 EXAMPLE OF I/O PORTS AVAILABLE ON A QEMU VM

```
[root@p-stt-cloud-sharednode1 vespa]# cat /proc/ioports
0000-0cf7 : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
  0050-0053 : timer1
  0060-0060 : keyboard
  0064-0064 : keyboard
  0070-0071 : rtc0
  0080-008f : dma page reg
  00a0-00a1 : pic2
  00c0-00df : dma2
  00f0-00ff : fpu
  0170-0177 : 0000:00:01.1
    0170-0177 : ata_piix
  01f0-01f7 : 0000:00:01.1
    01f0-01f7 : ata_piix
  0376-0376 : 0000:00:01.1
    0376-0376 : ata_piix
  03c0-03df : vga+
  03f2-03f2 : floppy
  03f4-03f5 : floppy
  03f6-03f6 : 0000:00:01.1
    03f6-03f6 : ata_piix
  03f7-03f7 : floppy
0cf8-0cff : PCI conf1
0d00-ffff : PCI Bus 0000:00
  afe0-afe3 : ACPI GPE0_BLK
  b000-b03f : 0000:00:01.3
    b000-b003 : ACPI PM1a_EVT_BLK
    b004-b005 : ACPI PM1a_CNT_BLK
    b008-b00b : ACPI PM_TMR
    b010-b015 : ACPI CPU throttle
  b100-b10f : 0000:00:01.3
    b100-b107 : piix4_smbus
  c000-c0ff : 0000:00:03.0
    c000-c0ff : 8139cp
  c100-c11f : 0000:00:01.2
    c100-c11f : uhci_hcd
  c120-c13f : 0000:00:04.0
    c120-c13f : virtio-pci
  c140-c14f : 0000:00:01.1
    c140-c14f : ata_piix
```

Figure 45: I/O ports on Qemu VM.

C

RÉSUMÉ DE LA THÈSE EN FRANÇAIS

C.1 INTRODUCTION

Les ordinateurs ont évolué depuis les unités centrales complexes et massives vers des stations de travail légères et pratiques. Nous découvrons de nouvelles utilisations des machines, pouvant s'adapter dynamiquement aux demandes les plus contraignantes en terme d'espace disque, de processeur ou de mémoire. Nous sommes entrés dans l'aire du Cloud computing.

Ce modèle introduit de nouvelles fonctionnalités dans les infrastructures comme un approvisionnement flexible et dynamique des ressources. Cependant cette approche du tout-partagé rend perplexe les futurs utilisateurs. De nouvelles menaces venant de l'intérieur comme de l'extérieur apparaissent. Les mécanismes traditionnels comme le chiffrement ne suffisent plus, et les outils disponibles sont hétérogènes et fragmentés. Ils leur manquent une vision d'ensemble qui leur permettrait d'orchestrer la sécurité de façon intégrée.

Cette thèse fournie des réponses pour protéger les architectures virtualisées avec une solution de gestion de la sécurité flexible, dynamique et automatique.

C.1.1 *Notre approche*

La plupart des obstacles à l'adoption du Cloud computing sont définis dans l'Appendix A. Bien que tous ces derniers soient critiques, notre approche se focalise sur l'isolation des ressources dans les infrastructures IaaS. Nous adressons les obstacles liés à la sécurité de l'hyperviseur, à l'isolation réseau et à l'élasticité. L'ouverture et la sécurité de bout-en-bout sont eux aussi pris en compte pour définir les briques de référence d'un cloud sécurisé.

C.1.2 *Propriétés de sécurité*

Les infrastructures doivent intégrer trois fonctionnalités pour adresser les problèmes définis dans l'approche.

Multi-couches. Une infrastructure Cloud est composée d'un ensemble de couches indépendantes ayant chacune leur propre mécanisme de sécurité, alors qu'une attaque peut en cibler plusieurs. Aussi, l'efficacité d'une défense est accrue. Par exemple, une couche spécifique peut avoir une vue globale avec des informations venant des autres couches.

Multi-latéralité. Un Cloud peut être composé de plusieurs organisations. Chaque organisation peut avoir ses propres objectifs de sécurité. Des politiques flexibles de haut niveau sont donc nécessaires pour abstraire les relations avec des équipements spécifiques.

Extensibilité. Les Clouds évoluent constamment pour devenir interopérables. La vision habituelle des sources fermées n'est donc pas adéquate. L'intégration de nouveaux équipements de sécurité doit être simple, rapide et transparente.

C.1.3 *Objectifs de recherche*

Les objectifs sont divisés en deux parties: (1) proposer et implémenter une architecture de sécurité de bout-en-bout pour les environnements virtualisés fournissant une vue intégrée des mécanismes de protection; et (2) définir dans cette dernière des mécanismes d'autoprotection pour l'infrastructure sous-jacente.

C.1.3.1 *Une architecture de sécurité pour les environnements virtualisés*

Cette thèse explore les projets à base de composants pour orchestrer et composer dynamiquement les différentes briques de sécurité comme les hyperviseurs, les éléments de sécurité physiques (TPM), les protections réseau (IDS/IPS, VPN), les stockages sécurisés ou encore les mécanismes de gestion de la confiance. Chaque composant déclare des propriétés garanties en utilisant des contrats, qui sont composés pour dériver les objectifs de sécurité globaux du cloud. Cette architecture de sécurité de bout-en-bout est validée au travers d'un prototype et de plusieurs cas d'usage.

Pour atteindre ce premier objectif, des recherches ont démontré la viabilité des projets à base de composants pour construire des systèmes complexes depuis des briques hétérogènes et atteindre une sécurité flexible. Nous explorons cette approche pour orchestrer et adapter les services de sécurité dans un cloud et pour composer des services de sécurité flexibles dans une architecture de sécurité unifiée. Les propriétés de sécurité fournies par les services de sécurité sont exprimées comme des contrats flexibles, comme des SLAs, pour dériver des objectifs de sécurité garantis par l'infrastructure cloud.

C.1.3.2 *Clouds auto-protégés*

Le deuxième objectif consiste à spécifier et implémenter des mécanismes d'autoprotection au sein du cloud. Nous identifions les composants nécessaires pour réaliser une ou plusieurs boucles d'autoprotection qui rendront le cloud autogéré. Cette thèse définit aussi une architecture d'autoprotection. Les composants de sécurité sont ensuite implémentés et intégrés dans le prototype du cloud sécurisé.

Pour atteindre le second objectif, l'approche autonomique d'IBM pour une sécurité autogérée a prouvé son efficacité pour construire des infrastructures sécurisées avec des surcoûts minimaux. Un canevas logiciel générique à base de composants a été défini. La première partie de cette dissertation étudie si ce canevas logiciel est suffisant pour la gestion automatique de la sécurité du cloud, et les extensions nécessaires.

C.1.4 *Principes de conception*

Connaissant les propriétés de sécurité et les objectifs, nous exposons un ensemble de principes de conception pour une architecture cloud auto protégée:

Auto-protection à base de politiques. Le paradigme à base de politiques a réussi à démontrer sa capacité à améliorer l'adaptabilité des systèmes autogérés. En effet,

un large choix de stratégies est supporté dans chaque partie de la boucle de contrôle. Ce principe adresse l'hétérogénéité, le multi-couche et la multi-latéralité.

Défense multi-couches. Les évènements détectés dans une couche déclenchent des réactions dans d'autres couches. Cette approche coordonnée réconcilie les différentes sémantiques des couches. L'approche multi-couche améliore la sécurité en évaluant l'étendue d'une attaque, souvent plus large qu'une seule couche, et peut donc mieux y répondre. Ce principe aborde principalement le multi-couche.

Boucles d'autoprotection multiples. Une seule décision peut générer plusieurs réactions et ainsi étendre la granularité de la surveillance. Plusieurs niveaux de décision sont alors possibles. Plusieurs degrés d'optimalité peuvent être choisis pour la réponse: rapide et locale, mais avec une précision moyenne, ou plus large et avec un haut niveau de pertinence. Ce principe aborde la multi-latéralité.

Architecture ouverte. L'architecture doit intégrer simplement des composants de sécurité hétérogènes. Ce principe permet une meilleure interopérabilité, et aborde donc l'extensibilité.

C.1.5 *Nos contributions*

Cette thèse présente une architecture et un canevas logiciel basés sur les principes cités précédemment pour construire une infrastructure cloud auto protégée. Notre solution appelée VESPA régule la protection des ressources au travers de plusieurs boucles de sécurité cordonnées qui surveillent les différentes couches de l'infrastructure.

VESPA est une architecture auto protégée pour les environnements virtualisés qui outrepassent les limitations que nous venons de voir. A base de politique, il régule la sécurité dans les couches et au travers des couches. La coordination flexible entre les boucles d'autoprotection permet de mettre en place un riche panel de stratégies de sécurité. L'évaluation d'une implémentation de VESPA démontre que son architecture est applicable pour sécuriser une infrastructure cloud. Cependant certains composants n'offrent pas assez de fonctionnalités de sécurité, comme les hyperviseurs.

KungFuVisor est un canevas logiciel pour construire des hyperviseurs auto protégeables en utilisant VESPA. Les hyperviseurs les plus courants sont capables de détecter les attaques contre leur intégrité, mais ne peuvent pas se réparer. Pour répondre à ce problème, le canevas régule la protection de l'hyperviseur au travers de plusieurs boucles de sécurité autonomiques coordonnées. Les interactions entre un pilote de périphérique et l'environnement du VMM peuvent être strictement surveillés et contrôlés. Le résultat est une architecture d'autoprotection très flexible qui applique des actions de réparations à plusieurs niveaux du VMM.

C.2 VESPA: ARCHITECTURE AUTOPROTÉGÉE D'ENVIRONEMENTS VIRTUALISÉS

C.2.1 *Modèle de menace*

Les attaques d'une infrastructure IaaS peuvent être regroupées en 3 catégories détaillées en suite: les ressources de calcul (comme le CPU, la RAM ou les périphériques); le

réseau (comme les switch virtuels ou les périphériques réseau); et le stockage (comme les disques dur virtuels).

Les ressources du cloud ne sont pas épargnées par les infections virales. Cependant il faut garantir l'isolation des clients quel que soit les composants partagés, l'allocation physique aux machines virtuelles doit donc être garantie. Une VM se voit assignée un nombre spécifique de CPU, RAM et périphériques. Plusieurs attaques ont démontré qu'il était possible d'outrepasser ces limites et de compromettre la confidentialité des VMs.

Les ressources allouées au réseau cloud sont phénoménales, afin de supporter un très grand nombre de VMs. Les attaquants utilisent cette bande passante pour menacer et neutraliser la disponibilité des services.

Un stockage central sert généralement à regrouper les disques durs virtuels. Ceci est une opportunité pour un attaquant qui aurait accès à ce stockage et qui pourrait compromettre les données des disques durs à froid.

C.2.2 *Architecture*

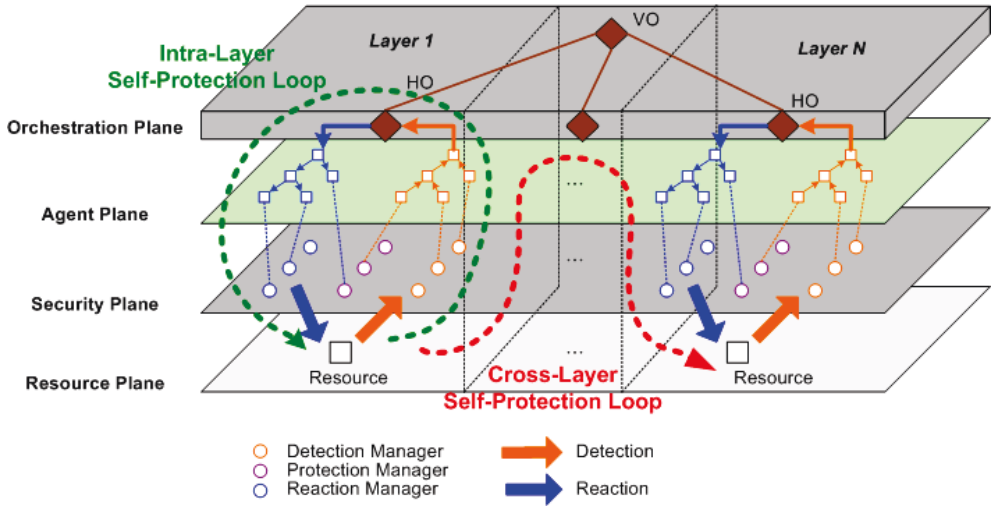


Figure 46: Architecture VESPA autoprotégée.

Une infrastructure IaaS regroupe les ressources en couches propres à la virtualisation. VESPA considère la gestion de la sécurité comme orthogonale aux couches, et met en place l'autoprotection au travers d'un ensemble de boucles autonomiques. Ces boucles sont opérées par des composants organisés en quatre plans distincts comme le montre la figure 46.

Tout en bas, le *Plan de Ressources* contient des ressources IaaS à surveiller et protéger. Au-dessus, un *Plan de Sécurité* contient les équipements de détection et de réaction qui assurent la sécurité des services comme la surveillance comportementale (via un IDS), ou une réaction avec un pare-feu. Ces composants sont les sondes et les déclencheurs des architectures de sécurité traditionnelles. Leurs APIs sont souvent spécifiques au vendeur.

Le plan suivant, le *Plan d'Agents*, abstrait l'hétérogénéité des composants de sécurité en définissant une couche de médiation entre les services de sécurité et les éléments décisionnaires. Ce plan est construit sur deux hiérarchies d'agents, une pour la détection, et une autre pour la réaction. Les agents ont deux rôles principaux. Premièrement, les

agents feuilles sont des adaptateurs entre le canevas logiciel VESPA et les composants de sécurité, utilisé pour traduire les APIs spécifique des vendeurs en un format normalisé. Cela permet d'introduire des composants tiers au sein de VESPA. Des agents de plus hauts niveau sont en charge soit de la corrélation d'alerte ou de l'affinage des politiques de réaction. Les agents permettent donc de définir des niveaux de sécurité granulaires des ressources pour la supervision de la sécurité.

Le plan de plus haut niveau, le *Plan d'Orchestration*, contient la logique de décision. Il est composé de deux types de gestionnaires autonomiques (appelés orchestrateurs dans VESPA): les orchestrateurs horizontaux qui définissent les adaptations de sécurité d'un niveau spécifique; et l'orchestrateur vertical qui se charge de superviser la sécurité entre les couches.

C.2.3 *Le modèle VESPA*

C.2.3.1 *Modèle de ressource*

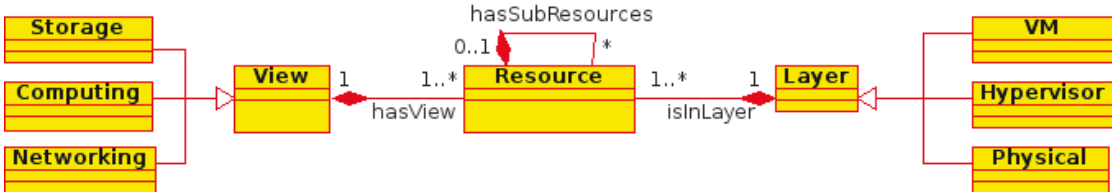


Figure 47: Modèle de ressource.

Les ressources IaaS sont classés suivant deux critères orthogonaux (voir la Figure 47). Une *couche* définie la localisation d'une ressource dans un pile IaaS. Les piles actuelles sont construites sur une *machine physique* qui fait tourner un *hyperviseur*, qui à son tour exécute des *machines virtuelles* (VM). Trois couches séparées sont alors identifiées. L'abstraction *vue* capture la classe de la ressource: *calcul*, *réseau* ou *stockage*.

Nous définissons les interactions entre les couches et les vues dans une infrastructure IaaS typique comme suit : La couche physique fournie les composants bruts de calcul, réseau et stockage aux autres composants de l'infrastructure. Des exemples pour chaque vues sont respectivement: le CPU, la mémoire et les cartes graphiques; les équipements d'interconnexion; et les périphériques de stockage connectés au réseau ou au PCI. Au-dessus, l'hyperviseur multiplexe et isole les ressources physiques pour fournir des versions virtualisées aux VMs. Les exemples pour chaque vues sont ici: le processeur virtuel et la mémoire virtuelle; les équipements réseau virtualisés comme les routeurs et pare-feu; et les espaces de stockage accessibles comme des périphériques dédiés. Au-dessus, les VMs ont leurs propres ressources comme tout système d'exploitation (OS), en se basant sur l'hyperviseur pour l'accès aux périphériques émulés.

C.2.3.2 *Modèle de sécurité*

VESPA protège les biens les plus critiques de l'infrastructure contre les attaques appelés Ressources protégées (PR) (voir Figure 48). Les attaques peuvent corrompre une PR, ou déconnecter le sujet qui l'utilise.

Dans VESPA, nous considérons plusieurs menaces qui peuvent cibler la couche VM: une VM malicieuse berne la stratégie de placement du IaaS pour se localiser sur le même serveur physique que la VM ciblée. Une attaque par canal caché peut ensuite

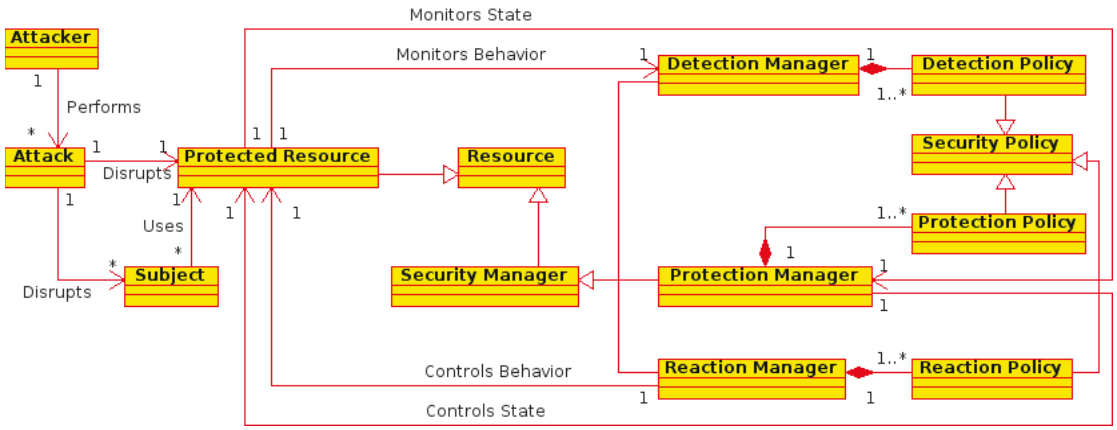


Figure 48: Modèle de sécurité.

exfiltrer des données. Une variante de cette attaque peut contaminer les OS invités des VMs, avec, par exemple, un virus se propageant sur le réseau ou le système de fichier. Les conséquences vont du trafic réseau inconnu à l'exécution de code arbitraire, en passant par l'escalade de privilèges. Les menaces de sécurité réseau traditionnelles sont aussi considérées entre les VMs, comme la récupération de trafic, l'usurpation d'adresse MAC/IP, ou le saut de VLAN.

Cependant il existe aussi des attaques plus puissantes contre l'hyperviseur. Une VM perce l'isolation de l'hyperviseur et prend le contrôle de la couche de virtualisation. Les possibles vecteurs d'attaques sont des mauvaises configurations ou des pilotes de périphériques mal confinés dans l'hyperviseur. L'attaquant peut ensuite compromettre l'intégrité de l'hyperviseur, installer des rootkits, et rebondir pour attaquer une autre machine virtuelle.

Les attaques contre la couche physique comme les attaques DMA, la compromission du mode SMM du CPU, ou les attaques traditionnelles sur le réseau physique sont aussi protégeables. Dans VESPA, les PR sont sous la supervision d'un gestionnaire de sécurité (SM). Ce dernier: (1) surveille le comportement d'une ressource au travers d'un gestionnaire de détection (DM), comme un IDS; (2) modifie le comportement de la ressource grâce à un gestionnaire de réaction (RM), comme un firewall; ou (3) surveille et modifie l'état interne de la ressource avec un gestionnaire de protection (PM), comme un gestionnaire d'intégrité des systèmes de fichier. Ces composants de sécurité sont souvent sur étagère et accessible uniquement au travers d'APIs spécifiques. Les comportements de tous les SM sont gouvernés par des politiques de sécurité.

C.2.3.3 *Modèle d'agent*

Le plan d'agent joue un rôle de médiation des couches entre les composants de sécurité sur étagère (SM) dans le plan de sécurité et la prise de décision dans le plan d'orchestration, pour la phase de détection et de réaction (voir Figure 49). Les agents agrègent les contextes de sécurité depuis des évènements du plan de sécurité pour une évaluation du risque haut niveau capable de diriger le processus de décision. Dans l'autre sens, ils réalisent aussi l'affinage des politiques de réaction choisies après la décision vers des politiques plus bas niveaux qui seront mises en place par les mécanismes de réaction du plan de sécurité. Les agents sont donc naturellement organisés en structure hiérarchique, avec des agents racines qui capturent les analyses et

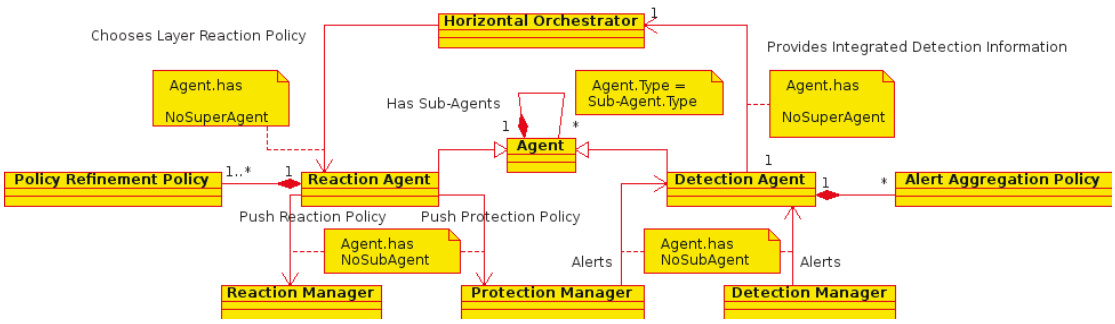


Figure 49: Agent Model.

réponses hauts niveaux. Deux hiérarchies d'agents sont définies, une pour la détection, une pour la réaction. Le comportement des agents est gouverné par des politiques de transformations, pour la corrélation d'alerte et l'affinage des politiques.

En accord avec le principe d'architecture ouverte, le plan d'agent doit être capable de prendre en compte des composants tiers. Les agents feuilles peuvent être vus comme des adaptateurs d'API.

La détection est mise en place de la façon suivante : Un DM ou PM notifie sont *Agent de Détection (DA)* d'un évènement de sécurité. Chaque DA applique ensuite une *Politique d'agrégation d'alerte* pour corréler les informations collectée depuis les agents sous-jacents avant de les envoyer aux agents parent. Quand l'information arrive à l'agent de détection racine, le contexte de sécurité est transmis au plan d'orchestration au travers de l'*Orchestrateur Horizontal* (HO).

Le processus de réaction est symétrique: après avoir choisi d'appliquer une politique de réaction spécifique à une couche, le HO envoie cette politique à l'agent de réaction racine (RA). Chaque RA va appliquer une *Politique d'affinage de politique* pour adapter au mieux la réponse, avant de l'envoyer aux agents sous-jacents les plus pertinents. Lorsqu'un agent feuille est atteint, les politiques de réaction ou de protection sont poussées vers les gestionnaires correspondants dans le plan de sécurité.

C.2.3.4 *Modèle d'orchestration*

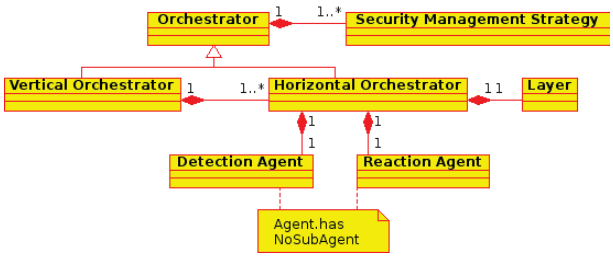


Figure 50: Orchestration Model.

La logique de décision est contenue dans le plan d'orchestration et divisée entre deux types d'orchestrateurs, comme le montre la Figure 50. Chaque couche de l'infrastructure IaaS contient un HO qui fournit une vue par couche de la gestion de la sécurité. Le HO est un gestionnaire de sécurité autonomique simple qui délivre une réponse rapide et locale. Il récupère le contexte de sécurité haut niveau du DA racine. La *Stratégie de gestion de la sécurité* permet de choisir la politique la plus adaptée pour une couche, qui est ensuite envoyée au RA racine pour application.

Le HO applique aussi les décisions venant de l'*Orchestrateur vertical* (VO), un gestionnaire autonomique global qui réalise des réactions de sécurité à spectre large et de haut niveau. Le VO coordonne les décisions propres aux couches pour fournir une réponse multi niveau consistante face aux menaces détectées. En se basant sur les informations collectées de chaque HO, le VO construit un savoir haut niveau global de toutes les ressources de l'infrastructure. La stratégie de gestion de la sécurité du VO contient les politiques définies par l'administrateur en cas d'alerte. Cela permet au VO de choisir la politique de réaction globale, qui sera poussée et mise en application par le HO.

C.2.3.5 *Modèle de politique*

Plusieurs types de politiques sont utilisés dans VESPA. Les politiques de surveillance définissent comment collecter, filtrer et corréler les alertes depuis les détecteurs. Les politiques d'affinage expriment comment dériver une réponse large et générique vers un sous ensemble d'actions compréhensibles par un mécanisme de réaction. Les stratégies de gestion de la sécurité gouvernent la prise de décision, définissant quelle réaction il faut appliquer ou générer dans un contexte de sécurité donné. Finalement, les politiques de réaction spécifient comment modifier le comportement ou l'état d'une ressource.

Notre canevas logiciel considère les politiques comme un élément précieux. Cependant, un vaste choix de modèle de politique et de langage a été proposé pour la détection ou la réaction. Choisir un modèle de politique trop spécifique limiterait fortement l'application de VESPA au Cloud, avec peu d'extensibilité. Par conséquent, plutôt que de développer un nouveau modèle de politique, nous avons basé VESPA sur un modèle à base d'évènement-condition-action générique qui peut prendre en compte un grand nombre de politiques.

C.2.4 *Une approche à deux niveaux*

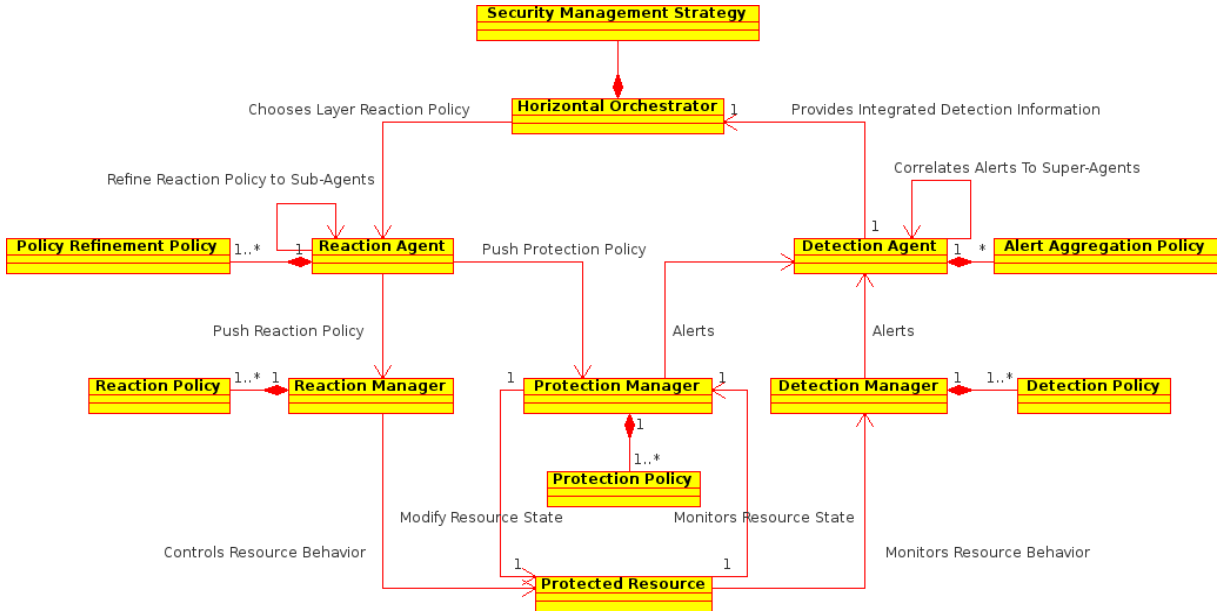
C.2.4.1 *Gestion autonomique intra-couche*

La boucle intra-couche fonctionne comme illustré sur la Figure 51a. Le comportement et l'état de chaque PR peuvent être surveillés et modifiés. Lorsqu'une menace est détectée par un DM et/ou un PM, l'agent associé est directement informé. Les informations de surveillance collectées sont ensuite agrégées et corrélées en suivant les politiques vue précédemment et transmises aux agents de niveau supérieur. Le processus est répété jusqu'à atteindre l'agent racine. L'information de contexte de sécurité globale est ensuite transmise au plan d'orchestration, où le HO prend des décisions d'adaptation de la sécurité, puis sélectionne une politique de réaction à appliquer. Les politiques de réactions affinées descendent vers les agents feuilles, qui poussent des politiques vers les RMs et/ou les PMs pour modifier leur comportement et leur état interne.

C.2.4.2 *Gestion autonomique multi couches*

Le vrai potentiel du canevas logiciel réside dans les réponses multi niveaux (voir Figure 51b): les contextes de sécurité propres aux couches sont envoyés par les HOs vers le VO. Cet orchestrateur consolide la connaissance du système, et applique la stratégie de gestion de la sécurité multi niveaux pour générer une réponse globale.

(a)



(b)

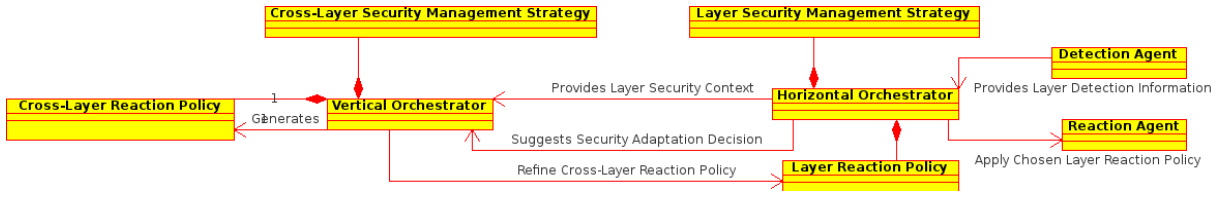


Figure 51: (a) Boucle intra couche; (b) Boucle multi niveaux .

AGENT	ROLE	LINES OF CODE
agent_log	Generic	50
agent_libvirt	Hypervisor / Generic	200
agent_gen_fuzz	Generic shared object wrapper	100
agent_run	Generic	50
agent_clamav	Specific / ClamAV	500
agent_web	Specific / Apache	60
agent_reunion	Specific / OpenMeeting	40
agent_ipad	Specific / IPad	50
agent_proc	Specific / Linux /proc	150
agent_openflow	Specific / OpenFlow	300
agent_floodlight	Specific / Floodlight	80
agent_pox	Specific / POX	120

Table 20: Agents VESPA disponibles.

Dans l'ensemble, VESPA définit une architecture où le nombre de couches multiplie les possibilités en terme d'actions de remédiation, locales ou au niveau de l'infrastructure. Son architecture à base d'agents permet aussi de combiner les vues en phase de détection et de réaction.

C.2.4.3 *Implémentation*

Deux versions de VESPA ont été implémentées en Python et en C. La première délivre un canevas logiciel simple d'utilisation grâce à l'introspection dynamique du code. Le second, combiné avec Fractal, délivre une implémentation plus rapide orienté embarqué. Cependant le développement se prête plus aux erreurs.

Chaque composant de VESPA hérite d'une classe centrale qui fournit notamment le multi tâches et les communications sécurisées.

C.2.5 *Evaluation*

C.2.5.1 *Extensibilité*

Plusieurs agents VESPA ont été créés au cours du développement (voir Tableau 20). Ils démontrent la flexibilité du canevas logiciel. Certaines classes d'agents sont génériques, comme le suivi des journaux systèmes ou l'exécution de commandes SSH.

Nous pensons que ces agents couvrent la majorité des cas d'usage habituels. Des agents plus spécifiques permettent de fournir des niveaux d'abstraction, comme l'agent libvirt qui profite de l'interface python intégrée pour contrôler les hyperviseurs.

Finalement, les agents feuilles sont liés aux technologies sous-jacentes comme le serveur web apache et la gestion de ses journaux, les contrôleurs réseaux définis par logiciel (SDN) ou l'antivirus ClamAV.

C.2.5.2 *Corrélation*

Pour illustrer les améliorations liées à la phase de détection, nous mesurons le taux de détection d'un ensemble de virus de façon incrémentale. D'abord avec un seul agent, puis deux, puis trois, pour augmenter le taux de détection tout en diminuant le nombre de faux positifs.

Chaque anti-virus est testé séparément pour comparer les performances propres à chacun. L'expérimentation est faite sur une VM Windows venant d'être installée. La base de données des virus est transférée sur la machine et décompressée, pour ensuite installer la dernière version de l'antivirus à tester. Finalement le répertoire des fichiers malicieux est scanné pour générer le rapport final. Nous prenons en compte le fait qu'un fichier peut contenir plusieurs menaces, et qu'un antivirus peut le signaler qu'une seule fois. Les rapports ont donc été filtrés pour extraire le maximum d'informations, et donc d'évaluer les antivirus de façon égale. Bien qu'ils détectent tous une grande partie des virus, aucun ne détecte tous les virus sans faux positifs (voir la Figure 52).

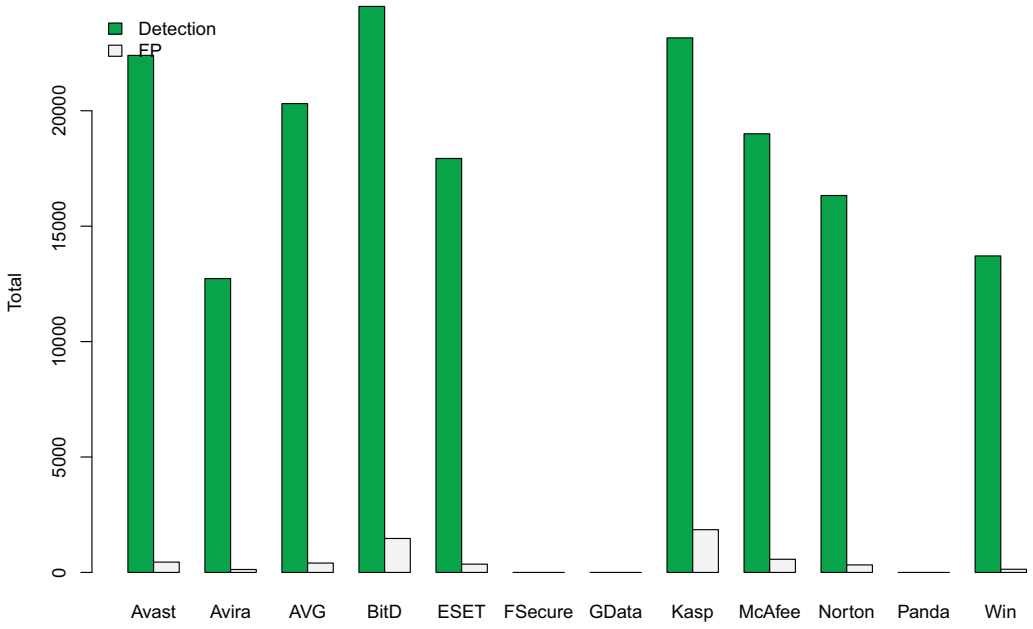


Figure 52: Comparaison des taux de détection et des faux positifs.

Nous interconnectons ensuite les antivirus avec VESPA. Chacun envoie le rapport vers le VO, qui décide si une analyse supplémentaire est nécessaire. Ici nous forçons le VO à faire deux puis trois passes en utilisant les antivirus ayant les meilleurs taux de détection. Les résultats sont représentés sur le diagramme de Venn Figure 53.

La combinaison des antivirus permet donc d'améliorer significativement le taux de détection en un temps donné. L'évaluation du temps est donnée sur les exemples plus complexes qui suivent.

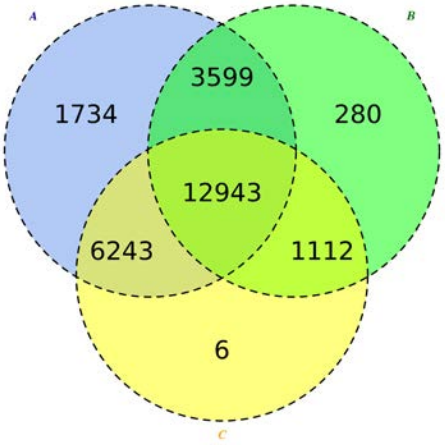


Figure 53: Détection Multi-Antivirus: (A) BitDefender, (B) ESET et (C) AVG Antivirus 2013.

C.2.5.3 *Conclusion*

Nous avons présenté notre canevas logiciel VESPA pour construire des infrastructures IaaS auto protégées.

Le modèle de VESPA est indépendant des systèmes, du langage de programmation ou du déploiement. La hiérarchie à base de noeud, abstrait de nombreux composants et facilite la tâche du développeur. Dans un monde idéal, les fournisseurs de composants délivrent un agent VESPA qui s'attache aux APIs d'origine. Cependant nous avons simplifié l'interface au maximum pour un développement très rapide et un débogage assisté. Avec un modèle clair et des implémentations dans plusieurs langages, nous anticipons une adoption de VESPA par les développeurs et les administrateurs de cloud.

C.3 KUNGFUVISOR

Cette section pousse VESPA dans le contexte des hyperviseurs. C'est une opportunité qui aborde le sujet des pilotes de périphériques tiers défectueux directement intégrés dans le coeur des hyperviseurs. Le code des pilotes représente en moyenne 80% de la taille d'un hyperviseur avec plusieurs pilotes qui n'ont pas été complètement testés. Nous avons donc adapté VESPA pour ajouter une couche d'interposition autour des pilotes de l'hyperviseur KVM et améliorer son niveau de sécurité.

C.3.1 *Le problème*

La couche de virtualisation, qui est la base des infrastructures cloud, est particulièrement vulnérable aux attaques basées sur les ressources partagées. Le sabotage d'une VM hébergée ou de l'hyperviseur peut conduire à rompre l'isolation des VM, offrant ainsi un contrôle complet du système. Jusqu'à présent, nous nous sommes attardés sur la protection des VMs, malheureusement les solutions correspondantes deviennent inefficaces lors de la compromission d'un hyperviseur vu qu'elles font confiance au VMM. Le vrai défi réside donc dans la protection de l'hyperviseur.

Des attaques récentes ont montré que la principale menace face à l'isolation des hyperviseurs provient des pilotes de périphériques défectueux ou malicieux.

C.3.2 *Limites des solutions existantes*

Pour tenter de résoudre ce problème, plusieurs techniques ont été proposées. Par exemple, la virtualisation des drivers permet une isolation forte, mais ne prend pas en compte la protection de la couche de virtualisation sous-jacente.

Les architectures de type "trusted computing" délivrent de fortes garanties vis à vis de l'intégrité du code. Malheureusement, elles ne détectent que les violations d'intégrité, et ne peuvent pas remettre le système dans un état sain. La vérification d'intégrité est généralement statique vu qu'une surveillance dynamique du système tout entier durant son fonctionnement deviendrait difficile à mettre en oeuvre.

Le sandboxing de pilotes a lui aussi été exploré: un moniteur de référence arbitre les accès entre pilote et périphérique, noyau ou espace utilisateur. Cependant les solutions demeurent limitées à du simple confinement, sans proposition d'action pour réparer le noyau. Les politiques de sécurité sont souvent définies en dur dans le mécanisme d'interception. Il est donc difficile de mettre en place des stratégies de protection dynamique, vu qu'elles doivent être configurées et mise à jour manuellement.

Pour réduire encore plus la surface d'attaque, de nouvelles architectures d'hyperviseurs à base de composants sont apparues. Mais elles nécessitent une réécriture de code poussée, les rendant difficile à appliquer sur les hyperviseurs les plus répandus.

Dans l'ensemble, les architectures d'hyperviseurs actuelles n'offre pas - ou peu - de protection pour la couche de virtualisation. Les tentatives précédentes souffrent: (1) de politiques statiques, difficiles à gérer et implémentée dans les mécanismes de protection, et (2) de peu de mécanismes de réparations face aux menaces.

C.4 ATTAQUES

C.4.1 *Modèle de menace*

L'attaquant a un contrôle arbitraire des machines virtuelles. Nous supposons que les périphériques physiques sont inaltérables, et que l'intégrité de la séquence de démarrage de l'hyperviseur est vérifiée. Cependant, un pilote de périphérique peut être défectueux, et peut donc être trafiqué pour exploiter une vulnérabilité dans le VMM. Une attaque par rebond depuis une VM serait: (1) rompre l'isolation de la VM grâce à un bug de pilote; (2) altérer et saboter le driver; (3) compromettre d'autres parties du VMMs ainsi que les VMs co-localisées. Une telle exploitation peut conduire à l'injection d'un rootkit et la récupération des communications inter VMs.

C.5 MODÈLE

Le canevas logiciel opère au travers de points d'interception répartis à plusieurs niveaux de l'hyperviseur. KungFuVisor arbitre les interactions entre les pilotes, les périphériques, les VMs et les autres structures critiques de l'hyperviseur. Il est alors possible de surveiller dynamiquement et de contrôler l'accès aux communications entre un pilote et son environnement. Cela permet de s'intégrer simplement dans les hyperviseurs dès lors que les points d'interception existent. Il faut noter que le confinement n'est pas limité à l'isolation mémoire (par exemple en utilisant des mécanismes processeur comme l'IOMMU): les politiques de réaction misent en place peuvent s'appliquer à d'autres canaux de communication.

Un plan de gestion de la sécurité fourni une vue unifiée de la logique de décision. Ce plan contient les briques d'orchestration réalisant de la détection et de la réaction à base de motifs complexes - au sein des couches, entre les couches et entre les différentes vues.

Ce modèle apporte deux bénéfices: (1) une sécurité autonomique de l'hyperviseur qui automatise l'administration des politiques, permettant une mise en place dynamique des politiques d'isolation des pilotes; et (2) une coordination des boucles autonomiques de sécurité qui peuvent déclencher un riche ensemble d'actions de réparation sur plusieurs parties de l'hyperviseur.

C.5.1 *Modèle d'hyperviseur*

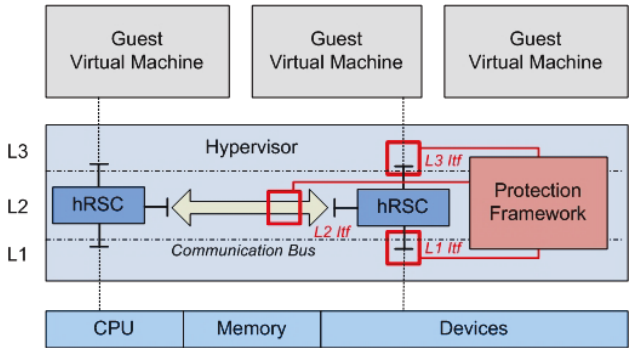


Figure 54: Modèle d'hyperviseur à 3 couches.

Nous considérons un modèle d'hyperviseur générique à trois couches (voir Figure 54).

La couche 1 (L1) contient l'état des ressources de calcul et de réseau physiques: le CPU, la mémoire physique, et les périphériques. La couche 2 (L2) contient la vue de l'hyperviseur des ressource de L1, que nous nommerons hRSC (ReSsourCes Hyperviseur): CPU virtuel, mémoire virtuelle du système d'exploitation hôte, et surtout les pilotes de périphériques. Les hRSCs sont les points critiques de la sécurité de l'hyperviseur, et doivent donc être sandboxés et analysés en détail. La couche 3 (L3) contient des services délivrés par l'hyperviseur au VMs sous forme d'appels à l'hyperviseur (hypercalls) qui peuvent exposer et modifier l'état d'un hRSC donné.

Chaque hRSC communique avec les couches adjacentes au travers de 3 interfaces. L'interface de L1 est utilisée par les interruptions physiques. L'interface L2 autorise les hRSC à communiquer avec les autres hRSCs au travers d'un bus de communication comme les IPCs. Finalement, l'interface L3 connecte un hRSC avec les ressources d'une VM grâce à des routines spécifiques de l'hyperviseur.

C.5.2 *Canevas logiciel de protection*

C.5.2.1 *Boucles multiples*

La défense automatique de l'hyperviseur est assurée au travers d'un ensemble de boucles autonomiques qui gère un ensemble de composants organisés en 3 plans (voir Figure 55). Tout en bas, un plan de ressources contient les hRSC à protéger. Au dessus, un plan de gestion contenant un ensemble d'agents est défini pour piloter les hRSCs.

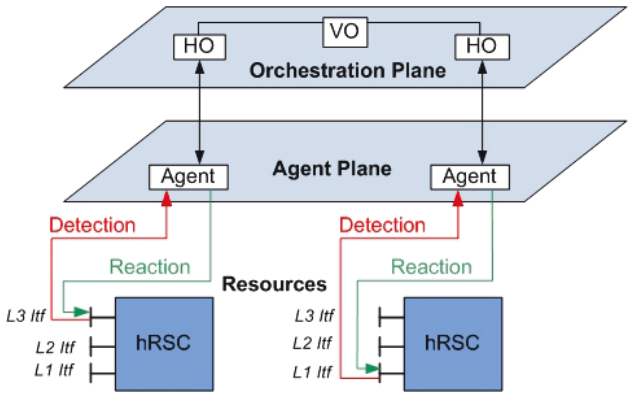


Figure 55: Architecture autoprotégée KungFuVisor.

Tout en haut, un plan d'orchestration coordonne la prise de décision entre les boucles d'autoprotection.

C.5.3 *Evaluation*

C.5.3.1 *Composants de VESPA*

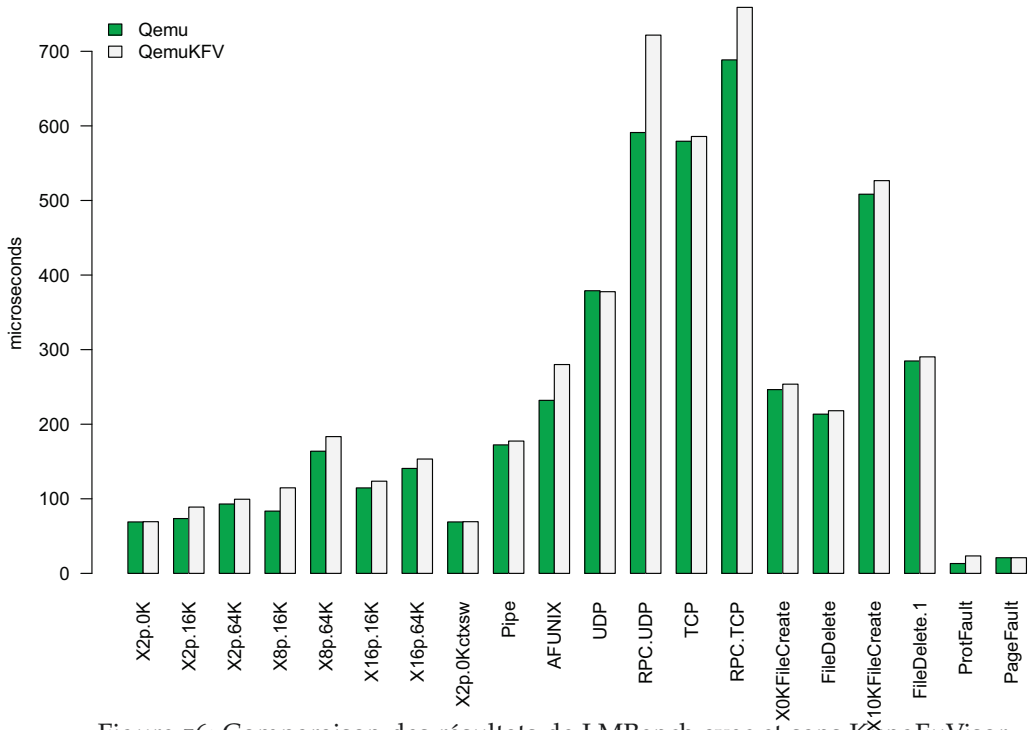


Figure 56: Comparaison des résultats de LMBench avec et sans KungFuVisor.

Nous avons utilisé les suites standard d'évaluation des performances pour évaluer l'impact de VESPA sur le système. La suite LMBench mesure la bande passante et les latences à plusieurs niveaux. La Figure 56 compare les temps donnés par LMBench sur une version non modifiée de Qemu et sur KungFuVisor en microsecondes.

Design Principles \ Solutions	VESPA	KungFuVisor
Protection avec politiques	✗	✗
Défense multi niveaux	✓	✗
Boucles multi niveaux	✓	✓
Architecture ouverte	✓	✓
Multi-Cloud	✓	✗

Table 21: Satisfaction des propriétés étendues

Notre implémentation est donc adaptée pour les systèmes critiques avec un surcoût acceptable et une sécurité accrue.

C.6 CONCLUSION

C.6.1 *Résultats principaux*

Dans cette thèse, nous avons montré qu'il est possible de modéliser la sécurité du cloud comme des composants d'un canevas logiciel. Quatre principes de conception ont été définis pour dépasser les limites des plateformes cloud actuelles: une protection automatique à base de politiques, une défense multi niveaux, plusieurs boucles d'auto protection et une architecture de sécurité ouverte. Cette approche permet de mettre en place des infrastructures cloud sécurisées et d'évoluer d'une sécurité locale statique vers une isolation dynamique sur plusieurs domaines.

Le modèle VESPA a été validé en utilisant deux implémentations en Python et en C. Ces canevas logiciels exposent des composants disponibles à plusieurs niveaux de granularité: un développeur peut interagir avec l'API d'une solution complète, ou créer un lien avec une partie précise d'un composant système. La composition de briques de bases hétérogènes valorise la sécurité des infrastructures IaaS à la carte. Plusieurs cas d'usages ont été implémentés pour démontrer le potentiel de VESPA, démontrant ainsi une adaptation simple et rapide dans plusieurs environnements. Aussi, les performances ne sont pas en reste avec un surcoût minimal.

L'extension de VESPA aux hyperviseurs a permis une évaluation quantitative des composants. KungFuVisor a nécessité plusieurs astuces pour devenir une solution viable de sécurité améliorée. L'hyperviseur est plus robuste contre les attaques publiques connues et les séquences d'interruptions malicieuses. L'adaptabilité de VESPA a été démontrée. Cela a nécessité la mise en place d'une architecture spécifique et des tests pour atteindre de bonnes performances, mais les composants de bases, de communication et de politiques n'ont pas été modifiés. Nous soutenons donc que VESPA est une architecture de sécurité prometteuse avec une conception modulaire qui peut répondre à plusieurs problématiques cloud actuelles.

C.6.1.1 *Protection automatique à base de politiques*

Notre architecture VESPA affiche une interface de politique épurée. Cependant, l'intégration de modèles de politiques nécessite des développements spécifiques. L'exemple de RBAC montre que nous devons assignés des objets à des rôles, et le modèle OrBAC

assigne les ressources à des organisations. Ces concepts ne sont pas complètement pris en charge par VESPA, et les politiques sont exprimés comme des machines à états finis. Le support d'OrBAC est partiel, les utilisateurs assignent des ressources à des domaines, qui sont proches des organisations d'OrBAC.

Cela ne suffit pas à remplir notre besoin de protection automatisée à base de politique, même si l'interface est définie.

C.6.1.2 *Défense multi niveaux*

La conception de VESPA intègre la défense multi niveaux dans l'architecture, ce qui valide notre besoin. Les VMs, les hyperviseurs et les périphériques physiques sont capables de communiquer de façon sécurisée via une interface stricte.

Cependant, KungFuVisor est spécifique à la couche de virtualisation. Il ne supporte pas la défense collaborative multi hyperviseurs. L'architecture de KungFuVisor adapte celle de VESPA au contexte des hyperviseurs et supprime la défense multi niveaux telle que définie dans nos principes de conception.

C.6.1.3 *Boucles d'auto protection multiples*

VESPA fournit plusieurs boucles d'autoprotection si les implémentations suivent l'architecture que nous avons définie. Les cas d'usages et KFV héritent de ce besoin. L'administrateur est capable d'interconnecter des éléments de sécurité de l'infrastructure de plusieurs façons. Chaque boucle revête en général une fonction de sécurité particulière, par exemple de prévenir les intrusions réseau ou les dénis de service.

C.6.1.4 *Architecture ouverte*

Le principe d'architecture ouverte a été rempli de plusieurs façons.

L'interface de communication est expliquée dans cette thèse et est abordée d'une manière plus détaillée dans la documentation du code. Elle est flexible et adaptable aux nouveaux besoins. L'interface interne de RPC fournie peut être adaptée à de nouveaux langages de façon transparente. La configuration dynamique avec l'introspection de code est divisée en fonctions indépendantes. Cette fonctionnalité est difficile à porter, et les développeurs peuvent adopter une autre approche pour récupérer les attributs d'un noeud.

Tous les composants de l'architecture VESPA et les cas d'usage sont disponibles sous licence LGPL sur github et l'orangeforge. Les développeurs peuvent donc modifier, étendre et corriger les noeuds disponibles pour leurs besoins spécifiques.

C.6.1.5 *Multi-Cloud*

Le cas d'usage OC2 a permis à VESPA de communiquer avec d'autres instances de VESPA. Les VOs s'envoient des alertes de pair à pair et réagir avec une meilleure vision. C'est une extension des boucles autonomiques de sécurité aux domaines hétérogènes.

C.6.2 *Limites*

Evidemment, le travail réalisé dans cette thèse n'est pas complet. Plusieurs axes de recherche doivent être creusés pour construire une infrastructure cloud sécurisée.

Premièrement, deux types de politiques sont utilisées dans le canevas logiciel: (1) des machines à états pour leurs simplicité et leurs capacités à favoriser l'adoption; (2) OrBAC pour s'associer aux domaines. Elles sont pragmatiques, génériques et évidentes dans des contextes distribués. Cependant des politiques plus fines peuvent mieux capturer les problématiques liées au cloud, comme des politiques d'isolation communes à différents matériaux informatiques. Lorsqu'un utilisateur veut migrer sa VM, comment abstraire une seule politique et assurer le même niveau d'isolation sur la destination? Le canevas logiciel VESPA fournit des éléments de réponse avec la hiérarchie d'agents et les politiques d'affinage associée, mais elles n'ont pas été complètement exploitées.

Secondement, la couche physique n'a pas été étudiée en profondeur. Le canevas logiciel VESPA peut donc louper des propriétés propres à cette couche. Pourtant, les équipements de sécurité physiques se trouvent partout dans les infrastructures des fournisseurs. Il faut donc intégrer cette couche et réutiliser les équipements de sécurité existants.

C.6.3 *Travaux futurs*

C.6.3.1 *Physique*

Bien que le niveau physique n'ai pas été exploité dans cette thèse, il faut prendre en compte les agents physiques pour atteindre le potentiel maximum de VESPA. Même si ce n'est pas une réelle limitation, il y a quelques adaptations à faire pour parler aux interfaces de gestion, comme la libvirt pour les hyperviseurs. Il n'y a aucune base permettant de gérer les équipements physique et il faut souvent un agent spécifique à chaque équipement.

Aussi, l'attestation physique des composants peut être utilisée pour garantir l'intégrité du canevas logiciel VESPA et le protéger des corruptions mémoire. Un exemple rapide serait basé sur les modules TPM ou la TEE qui vient avec les ordinateurs et téléphones actuels. Cependant ces composants ont déjà montrés des faiblesses, et la question de la confiance accordée aux fondeurs reste entière. Si une solution émerge, VESPA doit l'adopter pour s'assurer que les composants sont déployés sur les bons environnements d'exécution.

C.6.3.2 *Protection du canevas logiciel*

Les composants de VESPA n'ont pas été conçus pour être résistants aux attaques physiques. L'implémentation de l'architecture ne fournit pas de protection contre la modification du code.

Le canevas logiciel de protection automatisé vise à protéger les ressources hétérogènes existantes, mais les composants du canevas ne le sont pas directement. Les communications entre les couches sont chiffrées pour garantir la confidentialité et signées pour garantir l'intégrité, mais la disponibilité reste la grande faiblesse. Le canevas logiciel ne supporte pas une instance sans VO, alors que les HOs devraient protéger les agents sous-jacents.

L'autostabilisation peut fournir une meilleure protection de l'architecture VESPA.

C.6.3.3 *Virtualisation imbriquée*

Avec la mode de la virtualisation imbriquée, plusieurs défis sont apparus et n'ont pas étaient traités dans cette thèse. Le canevas logiciel VESPA générique n'est pas limité en nombre de couches mais il faut plus de travail pour appréhender les nouveaux enjeux. Une ébauche d'architecture de sécurité compatible avec VESPA a été élaborée.

C.6.3.4 *Hyperviseur autoprotégé*

Nous avons adressé les vulnérabilités liées aux interruptions de l'hyperviseur. Cependant il y a plusieurs autres vecteurs d'attaque qui peuvent affaiblir l'isolation de l'hyperviseur. Par exemple le composant qui charge la configuration d'une VM peut être vulnérable aux attaques par chaînes de caractères mal formées. L'utilisateur est capable de définir le nom de la machine et des informations qui doivent être analysée par l'hyperviseur. Si ce dernier n'échappe pas correctement la chaîne de caractère, c'est potentiellement un accès root arbitraire sur la machine. Une approche combinée entre VESPA et la désaggrégation de Xen est donc nécessaire pour limiter ces vecteurs d'attaque.

BIBLIOGRAPHY

[1] thrift-protobuf-compare: Comparing Various Aspects of Serialization Libraries on the JVM Platform. URL http://code.google.com/p/thrift-protobuf-compare/.

[2] The path to multi-level security in red hat enterprise linux. URL http://www.redhat.com/resourcelibrary/whitepapers/path-to-mlsec.

[3] Top threats to cloud computing. URL https://downloads.cloudsecurityalliance.org/initiatives/top_threats/Top_Threats_Cloud_Computing_Survey_2012.pdf.

[4] Pygrub vulnerability (cve-2007-4993), . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4993.

[5] Buffer overflow in xen para virtualized frame buffer (cve-2008-1943), . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1943.

[6] Cross-site scripting (xss) vulnerability in the xenapi http interfaces in citrix xenserver express, standard, and enterprise edition 4.1.0 (cve-2008-3253), . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3253.

[7] Heap-based buffer overflow in the flask_security_label function in xen 3.3 (cve-2008-3687), . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3687.

[8] Read/write kernel memory with the kvm_emulate_hypercall function (cve-2009-3290), . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3290.

[9] Invalid use fo cpl and iopl (cve-2010-0298), . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0298.

[10] Invalid use of pit_state data structure (cve-2010-0309), . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0309.

[11] The x86 emulator in kvm 83, when a guest is configured for symmetric multiprocessing (smp), does not properly restrict writing of segment selectors to segment registers (cve-2010-0419), . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0419.

[12] Invalid use of vmcs on fully virtualized xen guest (cve-2010-2938), . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2938.

[13] Uninitialized kvm_vcpu_events->interrupt.pad structure member (cve-2010-4525), . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4525.

[14] The pciej_write function in hw/acpi_piix4.c in the piix4 power management emulation in qemu-kvm does not check if a device is hotpluggable before unplugging the pci-isa bridge (cve-2011-1751), . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1751.

[15] Presentation of seccomp and setuid sandbox. URL http://www.imperialviolet.org/2009/08/26/seccomp.html.

[16] Orbac api implementation, 2011. URL http://orbac.org/?page_id=16.

[17] Linux kernel perf_event perf_swevent_init() function privilege escalation vulnerability, May 2013. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2094.

[18] C.L. Abad and R.I. Bonilla. An analysis on the schemes for detecting and preventing arp cache poisoning attacks. In *Distributed Computing Systems Workshops, 2007. ICDCSW '07. 27th International Conference on*, pages 60–60, 2007. doi: 10.1109/ICDCSW.2007.19.

[19] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 2–13, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. doi: 10.1145/1168857.1168860. URL http://doi.acm.org/10.1145/1168857.1168860.

[20] Y. Al-Nashif, A.A. Kumar, S. Hariri, Guangzhi Qu, Yi Luo, and F. Szidarovsky. Multi-Level Intrusion Detection System (ML-IDS). In *International Conference on Autonomic Computing (ICAC)*, 2008.

[21] Palesandro Alex, Lacoste Marc, and Wailly Aurelien. Analyse et prototypage de la virtualisation imbriquee dans le cloud. 2013.

[22] AMD. Amd virtualization (amd-v) technology, 2010. URL http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx.

[23] ARM. ARM Security Technology: Building a Secure System using TrustZone Technology, 2009.

[24] Fabien Autrel, Céline Coma, and et al. Motorbac 2: a security policy tool, 2008. URL http://www.rennes.enst-bretagne.fr/~fcuppens/articles/motorbac-sarssi08.pdf.

[25] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. HyperSentry: Enabling Stealthy In-Context Measurement of Hypervisor Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[26] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 375–388, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046752. URL http://doi.acm.org/10.1145/2046707.2046752.

[27] Arati Baliga, Liviu Iftode, and Xiaoxin Chen. Automated Containment of Rootkits Attacks. *Computers & Security*, 27:323–334, 2008.

[28] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945462. URL http://doi.acm.org/10.1145/945445.945462.

[29] Ken Barr et al. The VMware Mobile Virtualization Platform: Is that a Hypervisor in your Pocket? *SIGOPS Oper. Syst. Rev.*, 44(4):124–135, 2010.

[30] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *Proceedings of the 17th international conference on Formal methods*, FM'11, pages 231–245, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21436-3. URL http://dl.acm.org/citation.cfm?id=2021296.2021322.

[31] Mick Bauer. Paranoid penguin: an introduction to novell apparmor. *Linux J.*, 2006(148):13–, August 2006. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=1149826.1149839.

[32] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1247360.1247401.

[33] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[34] Stefan Berger, Ramón Cáceres, Dimitrios Pendarakis, Reiner Sailer, Enriquillo Valdez, Ronald Perez, Wayne Schildhauer, and Deepa Srinivasan. TVDc: Managing Security in the Trusted Virtual Datacenter. *SIGOPS Oper. Syst. Rev.*, 42: 40–47, January 2008.

[35] Stefan Berger, Ramon Careces, Dimitrios Pendarakis, Reiner Sailer, and Enriquillo Valdez. TVDc: Managing Security in the Trusted Virtual Datacenter. *ACM SIGOPS Operating Systems Review*, 42(1), 2008.

[36] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[37] Jeffrey Bickford et al. Rootkits on Smart Phones: Attacks, Implications and Opportunities. In *Eleventh Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2010.

[38] Tegawende Bissyande. *Contributions for improving debugging of kernel-level services in a monolithic operating system*. PhD thesis, Université Sciences et Technologies - Bordeaux I, 2013. URL http://tel.archives-ouvertes.fr/tel-00821893/.

[39] Blackberry. BES10. http://www.blackberry.com/BES10.

[40] A. Brown and C. Redlin. Measuring the Effectiveness of Self-Healing Autonomic Systems. In *International Conference on Autonomic Computing (ICAC)*, 2005.

[41] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36 (11-12):1257–1284, September 2006. ISSN 0038-0644. doi: 10.1002/spe.v36:11/12. URL http://dx.doi.org/10.1002/spe.v36:11/12.

[42] Shakeel Butt, H. Andrés Lagar-Cavilla, Abhinav Srivastava, and Vinod Ganapathy. Self-service cloud computing. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 253–264, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382226. URL http://doi.acm.org/10.1145/2382196.2382226.

[43] David Champagne. *Scalable security architecture for trusted software*. PhD thesis, Princeton University, 2010.

[44] Huoping Chen, Youssif B. Al-Nashif, Guangzhi Qu, and Salim Hariri. Self-Configuration of Network Security. In *IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, 2007.

[45] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[46] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177–186. IEEE, 2008.

[47] D. Chess, C. Palmer, and S. White. Security in an Autonomic Computing Environment. *IBM Systems Journal*, 42(1):107–118, 2003.

[48] N.M. Mosharaf Kabir Chowdhury and University of Waterloo Raouf Boutaba. Network virtualization: State of the art and research challenges. *IEEE Communications Magazine*, 47(7):20–26, 2009.

[49] Mihai Christodorescu, Reiner Sailer, Douglas Lee Schales, Daniele Sgandurra, and Diego Zamboni. Cloud Security is not (just) Virtualization Security. In *ACM Workshop on Cloud Computing Security (CCSW)*, 2009.

[50] Cisco. Nexus 1000v. URL http://www.cisco.com/web/go/nexus1000v.

[51] Cloud Security Alliance. Security Guidance for Critical Areas of Focus in Cloud Computing. http://www.cloudsecurityalliance.org/csaguide.pdf.

[52] Cloud Security Alliance. Top Threats to Mobile Computing, 2012.

[53] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Micha? Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03358-2. doi: 10.1007/978-3-642-03359-9_2. URL http://dx.doi.org/10.1007/978-3-642-03359-9_2.

[54] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[55] J.M. Combes, A. Wailly, and M. Laurent. Cga as alternative security credentials with ikev2: implementation and analysis.

[56] corbet. Shared subtrees namespace details, 2005. URL http://lwn.net/Articles/159077/.

[57] corbet. Net namespace details, 2007. URL http://lwn.net/Articles/219794/.

[58] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *International Workshop on Policies for Distributed Systems and Networks (POLICY)*, 2001.

[59] S. De Capitani di Vimercati, S. Foresti, P. Samarati, and S. Jajodia. Access Control Policies and Languages. *International Journal of Computational Science and Engineering*, 3(2):94–102, 2007.

[60] N. De Palma, D. Hagimont, F. Boyer, and L. Broto. Self-Protection in a Clustered Distributed System. *Parallel and Distributed Systems, IEEE Transactions on*, 23(2): 330 –336, 2012.

[61] David Defour and Eric Petit. Températures, erreurs matérielles et GPU. In *COMPAS'2013*, pages 1–11, Grenoble, France, 2013. URL http://hal.archives-ouvertes.fr/hal-00785386.

[62] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[63] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 297–312, 2011. doi: 10.1109/SP.2011.11.

[64] Brendan Dolan-Gavitt, Bryan Payne, and Wenke Lee. Leveraging forensic tools for virtual machine introspection. 2011.

[65] Shlomi Dolev, Juan A. Garay, Niv Gilboa, Vladimir Kolesnikov, and Yelena Yuditsky. Towards efficient private distributed computation on unbounded input streams - (extended abstract). In *ACNS*, pages 69–83, 2013.

[66] Jason Donenfeld. Linux local privilege escalation via suid /proc/pid/mem write, 2012. URL http://blog.zx2c4.com/749.

[67] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: architecture, design and implementation. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855865.1855875.

[68] Georges Dunlap. Securing your cloud with xen's advanced security features. 2013. URL http://www.xenproject.org/component/allvideoshare/video/latest/securing-your-cloud-with-xen.html.

[69] A Abou El Kalam and Yves Deswarte. Multi-orbac: A new access control model for distributed, heterogeneous and collaborative systems. In *8th IEEE International Symposium on Systems and Information Security*, 2006.

[70] Nelson Elhage. Virtunoid: Breaking out of KVM. In *Black Hat USA*, 2011.

[71] ENISA. Cloud Computing: Benefits, Risks and Recommendations for Information Security, 2010.

[72] J. Strassner et al. The Design of a New Context-Aware Policy Model for Autonomic Networking. In *International Conference on Autonomic Computing (ICAC)*, 2008.

[73] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, Noel de Palma, and Gwen Salaun. Automated configuration of legacy applications in the cloud. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 170–177, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4592-9. doi: 10.1109/UCC.2011.32. URL http://dx.doi.org/10.1109/UCC.2011.32.

[74] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, NoÃ«l de Palma, and Gwen SalaÃŒn. Automated Configuration of Legacy Applications in the Cloud. In *IEEE International Conference on Utility and Cloud Computing (UCC)*, 2011.

[75] Jean-Philippe Fassino. *THINK: vers une architecture de systemes flexibles*. PhD thesis, Télécom ParisTech, 2001.

[76] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[77] David Ferraiolo, Janet Cugini, and D Richard Kuhn. Role-based access control (rbac): Features and motivations. In *Proceedings of 11th Annual Computer Security Application Conference*, pages 241–48. sn, 1995.

[78] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Controls* -. Artech House, Norwood, 2003. ISBN 978-1-580-53370-6.

[79] Peter Ferrie. Attacks on more virtual machine emulators. .

[80] Peter Ferrie. Attacks on virtual machine emulators. .

[81] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Internaional Journal of High Performance Computing Applications*, 15:200–222, 2001.

[82] Eric Freudenthal et al. dRBAC: Distributed Role-based Access Control for Dynamic Coalition Environments. In *International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[83] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The Design and Implementation of Microdrivers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[84] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *In Proc. Network and Distributed Systems Security Symposium*, pages 163–176, 2003.

[85] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[86] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *ACM Symposium on Operating Systems Principles (SOSP*, 2003.

[87] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, HOTOS'07, pages 6:1–6:6, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1361397.1361403.

[88] Roxana Geambasu, Tadayoshi Kohno, Amit Levy, and Henry M. Levy. Vanish: Increasing Data Privacy with Self-Destructing Data. In *USENIX Security Symposium*, 2009.

[89] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser Intrusion Recovery System. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[90] Google. protobuf Serializer. URL http://code.google.com/p/protobuf/.

[91] Green Hills Software. Integrity Multivisor. http://www.ghs.com/products/rtos/integrity_virtualization.html.

[92] T Harada, T Handa, and Y Itakura. Design and implementation of tomoyo linux. In *IPSJ Symposium Series*, number 13, pages 101–110, 2009.

[93] R. He, M. Lacoste, and J. Leneutre. A Policy Management Framework for Self-Protection of Pervasive Systems. In *International Conference on Autonomic and Autonomous Systems (ICAS)*, 2010.

[94] Ruan He, Marc Lacoste, and Jean Leneutre. ASPF: A Policy Administration Framework for Self-Protection of Large-Scale Systems. *IARIA International Journal On Advances in Security*, 3(3–4):104–122, 2010.

[95] Gernot Heiser. The Role of Virtualization in Embedded Systems. In *Workshop on Isolation and Integration in Embedded Systems (IIES)*, 2010.

[96] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *ACM SIGCOMM Asia-Pacific Workshop on Systems (APSys)*, 2010.

[97] Kenneth Hess and Amy Newman. *Practical Virtualization Solutions: Virtualization from the Trenches*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2009. ISBN 0137142978, 9780137142972.

[98] Jinho Hwang, Sai Zeng, Frederick Wu, and Timothy Wood. A Component Based Performance Comparison of Four Hypervisors. In *to appear in IFIP/IEEE Integrated Network Management Symposium (IM 2013)*, May 2013.

[99] J.-Y. Hwang et al. Xen on ARM: System Virtualization using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *5th IEEE Consumer Communications & Networking Conference (CCNC)*, 2008.

[100] Amani Ibrahim, James Hamlyn-Harris, John Grundy, and Mohamed Almorsy. CloudSec: A Security Monitoring Appliance for Virtual Machines in the IaaS Cloud Model. In *International Conference on Network and Systems (NSS)*, 2011.

[101] A.S. Ibrahim, J. Hamlyn-Harris, John Grundy, and M. Almorsy. Cloudsec: A security monitoring appliance for virtual machines in the iaas cloud model. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 113–120, 2011. doi: 10.1109/ICNSS.2011.6059967.

[102] Int. Introduction to software guard extensions, Website 2013. URL http://privatecore.com/wp-content/uploads/2013/06/HASP-instruction-presentation-release.pdf.

[103] Intel. Introduction to memory protection extensions. Website, 2013. URL http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[104] Tarush Jain and Tanmay Agrawal. The haswell microarchitecture-4th generation processor.

[105] Marek Jawurek and RWTH Aachen. Rsbac-a framework for enhanced linux system security. *Dependable Distributed Systems, Laboratory of dependable distributed systems, RWTH Aachen University*, 2006.

[106] X. Jiang and D. Xu. VIOLIN: Virtual Internetworking on OverLay INfrastructure. In *International Symposium on Parallel and Distributed Processing and Applications*, 2004.

[107] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy Malware Detection through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. *ACM Trans. Inf. Syst. Secur.*, 13:1–28, 2010.

[108] A. Abou El Kalam, Y. Deswarte, A. Baï¿œna, and M. Kaï¿œniche. Polyorbac: A security framework for critical infrastructures. *International Journal of Critical Infrastructure Protection*, 2(4):154 – 169, 2009. ISSN 1874-5482. doi: 10.1016/j.ijcip.2009.08.005. URL http://www.sciencedirect.com/science/article/pii/S1874548209000262.

[109] A.A.E. Kalam, R.E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin. Organization based access control. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 120–131, 2003. doi: 10.1109/POLICY.2003.1206966.

[110] Katarzyna Keahey, MaurÃcio Tsugawa, AndrÃ©a Matsunaga, and JosÃ© Fortes. Sky Computing. *IEEE Internet Computing*, 13:43–51, 2009.

[111] Jeffrey Kephart and William Walsh. An Artificial Intelligence Perspective on Autonomic Computing Policies. In *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, 2004.

[112] Jeffrey O. Kephart. Autonomic computing: the first decade. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 1–2, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0607-2. doi: 10.1145/1998582.1998584. URL http://doi.acm.org/10.1145/1998582.1998584.

[113] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36 (1):41–50, 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1160055.

[114] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 403–412. ACM, 2011.

[115] Kirill Kolyshkin. Lxc website, . URL http://lxc.sourceforge.net/.

[116] Kirill Kolyshkin. Virtualization in linux, . URL http://download.openvz.org/doc/openvz-intro.pdf.

[117] Kirill Kolyshkin. Pid namespace details, 2007. URL http://lwn.net/Articles/259217/.

[118] J. Kiszka and B. Wagner. Domain and type enforcement for real-time operating systems. In *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference*, volume 2, pages 439–446 vol.2, 2003. doi: 10.1109/ETFA.2003.1248732.

[119] Avi Kivity. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[120] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596. URL http://doi.acm.org/10.1145/1629575.1629596.

[121] Ricardo Koller, Raju Rangaswami, Joseph Marrero, Igor Hernandez, Geoffrey Smith, Mandy Barsilai, Silviu Necula, S. Masoud Sadjadi, Tao Li, and Krista Merrill. Anatomy of a Real-Time Intrusion Prevention System. In *International Conference on Autonomic Computing (ICAC)*, 2008.

[122] Kostya Kortchinsky. CloudBurst: A VMware Guest to Host Escape Story. In *BLACKHAT*, 2009.

[123] D. Koufaty and D.T. Marr. Hyperthreading technology in the netburst microarchitecture. *Micro, IEEE*, 23(2):56–65, March 2003. ISSN 0272-1732. doi: 10.1109/MM.2003.1196115.

[124] Sacha Krakowiak. *Principes des systemes d'exploitation des ordinateurs (Dunod informatique) (French Edition)*. Dunod, 1987. ISBN 2040186328. URL http://www.

amazon.com/Principes-systemes-dexploitation-ordinateurs-informatique/dp/2040186328%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D2040186328.

[125] Ralph LaBarge and Thomas McGuire. Cloud penetration testing. *CoRR*, abs/1301.1912, 2013.

[126] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1): 18–24, 1974.

[127] Carl E Landwehr. Formal models for computer security. *ACM Computing Surveys (CSUR)*, 13(3):247–278, 1981.

[128] Boris Lau and Vanja Svajcer. Measuring virtual machine detection in malware using dsd tracer. *Journal in Computer Virology*, 6(3):181–195, 2010.

[129] Kevin P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux J.*, 1996(29es), September 1996. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=326350.326357.

[130] Dirk Leinenbach and Thomas Santen. Verifying the microsoft hyper-v hypervisor with vcc. In Ana Cavalcanti and DennisR. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-05088-6. doi: 10.1007/978-3-642-05089-3_51. URL http://dx.doi.org/10.1007/978-3-642-05089-3_51.

[131] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection*, pages 338–357. Springer, 2011.

[132] Linux-VServer. Secure capabilities. URL http://linux-vserver.org/Paper#Secure_Capabilities.

[133] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[134] Marco d'Itri. Evading from linux containers. URL http://blog.bofh.it/debian/id_413.

[135] J.M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158, 2010. doi: 10.1109/SP.2010.17.

[136] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38:69–74, March 2008.

[137] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical*

Conference, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1268299.1268322.

[138] P. Mell and T. Grance. The nist definition of cloud computing (draft). *NIST special publication*, 800:145, 2011.

[139] Microsoft. Hyper-v architecture, 2008. URL http://msdn.microsoft.com/en-us/library/cc768520%28v=bts.10%29.aspx.

[140] Omid Mola and Michael Bauer. Towards Cloud Management by Autonomic Manager Collaboration. *International Journal of Communications, Network and System Sciences*, 4(12):790–802, 2011.

[141] James Morris. sVirt: Hardening Linux Virtualization with Mandatory Access Control. In *Linux.conf.au Conference*, 2009.

[142] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving Xen Security through Disaggregation. In *International Conference on Virtual Execution Environments (VEE)*, 2008.

[143] Kara Nance, Matt Bishop, and Brian Hay. Virtual Machine Introspection: Observation or Interference? *IEEE Security and Privacy*, 6:32–37, September 2008.

[144] Anh Nguyen, Himanshu Raj, Shravan Rayanchu, Stefan Saroiu, and Alec Wolman. Delusional boot: securing hypervisors without massive re-engineering. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 141–154, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168851. URL http://doi.acm.org/10.1145/2168836.2168851.

[145] Anh M Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T King, and Hai D Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 441–450. IEEE, 2009.

[146] Vincent Nicomette. *La protection dans les systèmes à objets répartis*. These, Institut National Polytechnique de Toulouse - INPT, December 1996. URL http://tel.archives-ouvertes.fr/tel-00175252. Rapport LAAS n96496.

[147] ObjectSecurity. OpenPMF White Paper, 2011. URL http://www.openpmf.org/.

[148] K.A. Oostendorp, L. Badger, C.D. Vance, W.G. Morrison, M.J. Petkac, D.L. Sherman, and D.F. Sterne. Domain and type enforcement firewalls. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 1, pages 351–361 vol.1, 2000. doi: 10.1109/DISCEX.2000.825039.

[149] Open vSwitch. URL http://openvswitch.org.

[150] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. *Test*, pages 1–10, 2007. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.6943&rep=rep1&type=pdf.

[151] Said Oulmakhzoune, Nora Cuppens-Boulahia, Frédéric Cuppens, Stéphane Morucci, Mahmoud Barhamgi, and Djamal Benslimane. Privacy query rewriting algorithm instrumented by a privacy-aware access control model. *annals of telecommunications-annales des télécommunications*, pages 1–17, 2013.

[152] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: how to automatically generate procedures to detect cpu emulators. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, WOOT'09, pages 2–2, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855876.1855878.

[153] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, volume 41, page 86, 2009.

[154] Ahmed Patel, Qais Qassim, and Christopher Wills. A Survey of Intrusion Detection and Prevention Systems. *Information Management & Computer Security*, 18(4): 277–290, 2010.

[155] Bryan Payne. Simplifying virtual machine introspection using libvmi. 2012.

[156] Bryan Payne, Martim Carbone, and Wenke Lee. Secure and Flexible Monitoring of Virtual Machines. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[157] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, security threats, and solutions. *ACM Comput. Surv.*, 45(2):17:1–17:39, March 2013. ISSN 0360-0300. doi: 10.1145/2431211.2431216. URL http://doi.acm.org/10.1145/2431211.2431216.

[158] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. nether: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the Fourth European Workshop on System Security*, page 3. ACM, 2011.

[159] Ronald Perez, Reiner Sailer, and Leendert van Doorn. vtpm: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium*, pages 305–320, 2006.

[160] Diego Perez-Botero and Ruby B Lee. Characterizing the vm-hypervisor attack surface.

[161] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing*, Cloud Computing '13, pages 3–10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2067-2. doi: 10.1145/2484402.2484406. URL http://doi.acm.org/10.1145/2484402.2484406.

[162] Jérôme Petazzoni. Lxc website, 2013. URL http://www.socallinuxexpo.org/sites/default/files/presentations/Jerome-Scale11x%20LXC%20Talk.pdf.

[163] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A Formal Model for Virtual Machine Introspection. In *Workshop on Virtual Machine Security (VMSec)*, 2009.

[164] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974. ISSN 0001-0782. doi: 10.1145/361011.361073. URL http://doi.acm.org/10.1145/361011.361073.

[165] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 18–18, Berkeley, CA, USA, 2003. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251353.1251371.

[166] Danny Quist, Val Smith, and Offensive Computing. Further down the vm spiral, 2006.

[167] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. *Information Security*, pages 1–18, 2007.

[168] Massimiliano Rak, Loredana Liccardo, and Rocco Aversa. A SLA-Based Interface for Security Management in Cloud and Grid Integration. *International Conference on Information Assurance and Security (IAS)*, 2011.

[169] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud! Exploring Information Leakage in Third-Party Compute Clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[170] Sebastian Roschke, Feng Cheng, and Christoph Meinel. Intrusion Detection in the Cloud. In *International Conference on Dependable, Autonomic, and Secure Computing (DASC)*, 2009.

[171] Sandra Rueda, Yogesh Sreenivasan, and Trent Jaeger. Flexible Security Configuration for Virtual Machines. In *ACM Workshop on Computer Security Architectures (CSAW)*, 2008.

[172] J. Rutkowska and R. Wojtczuk. The Qubes OS Architecture. Technical report, Invisible Things Lab, 2010.

[173] Joanna Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction. *Invisible Things*, 2004.

[174] Joanna Rutkowska and Alexander Tereshkin. Bluepilling the Xen Hypervisor. In *BlackHat Technical Security Conference (BLACKHAT)*, 2008.

[175] Joanna Rutkowska and Rafał Wojtczuk. Preventing and detecting xen hypervisor subversions. *Blackhat Briefings USA*, 2008.

[176] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[177] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[178] Casey Schaufler. The simplified mandatory access control kernel, 2008. URL http://schaufler-ca.com/yahoo_site_admin/assets/docs/SmackWhitePaper.257153003.pdf.

[179] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[180] Monirul Sharif, Wenke Lee, Weidong Cui, and Wenke Lee. Secure In-VM Monitoring Using Hardware Virtualization. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[181] Weidong Shi, JongHyuk Lee, Taeweon Suh, Dong Hyuk Woo, and Xinwen Zhang. Architectural support of multiple hypervisors over single platform for enhancing cloud computing security. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 75–84, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1215-8. doi: 10.1145/2212908.2212920. URL http://doi.acm.org/10.1145/2212908.2212920.

[182] 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, and Kazuhiko Kato. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *International Conference on Virtual Execution Environments (VEE)*, 2009.

[183] Yoav Shoham. Elsevier artint 931 agent-oriented programming, 1991.

[184] F.M. Sibai and D.A. Menascé and. Defeating the Insider Threat via Autonomic Network Capabilities. In *Third International Conference onCommunication Systems and Networks (COMSNETS)*, 2011.

[185] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. Technical report, NAI Labs Report, 2001.

[186] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. ISBN 1558609105.

[187] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89861-0. doi: 10.1007/978-3-540-89862-7_1. URL http://dx.doi.org/10.1007/978-3-540-89862-7_1.

[188] Borja Sotomayor, Rubén S. Montero, Ignacio M. Llorente, and Ian Foster. Virtual Infrastructure Management in Private and Hybrid Clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.

[189] Brad Spengler. Detection, prevention, and containment: A study of grsecurity. In *Libres Software Meeting*, 2002.

[190] M.H. Sqalli, F. Al-Haidari, and K. Salah. Edos-shield - a two-steps mitigation technique against edos attacks in cloud computing. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 49–56, 2011. doi: 10.1109/UCC.2011.17.

[191] Abhinav Srivastava and Jonathon Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 39–58,

Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87402-7. doi: 10.1007/978-3-540-87403-4_3. URL http://dx.doi.org/10.1007/978-3-540-87403-4_3.

[192] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *European conference on Computer systems (EuroSys)*, 2010.

[193] J. Strassner. *Policy-Based Network Management: Solutions for the Next Generation*. Morgan Kaufman, 2003.

[194] J. Strassner, N. Agoulmine, and E. Lehtihet. FOCALE: A Novel Autonomic Networking Architecture. In *Latin American Autonomic Computing Symposium (LAACS)*, 2006.

[195] A. Sundararaj and P. Dinda. Towards Virtual Networks for Virtual Machine Grid Computing. In *USENIX Virtual Machine Research and Technology Symposium (VM)*, 2004.

[196] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007. ISBN 0321446119.

[197] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[198] SWSoft. Virtuozzo linux virtualization. URL http://www.virtuozzo.com.

[199] Jakub Szefer and Ruby B. Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 437–450, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151022. URL http://doi.acm.org/10.1145/2150976.2151022.

[200] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[201] Jakub Szefer, Eric Keller, Ruby B Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 401–412. ACM, 2011.

[202] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[203] Lin Tan, E.M. Chan, R. Farivar, N. Mallick, J.C. Carlyle, F.M. David, and R.H. Campbell. iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support. In *International Symposium on Dependable, Autonomic and Secure Computing (DASC)*, 2007.

[204] Andrew S. Tanenbaum. *Modern Operating Systems (3rd Edition)*. Prentice Hall, 2007. ISBN 0136006639. URL http://www.amazon.com/Modern-Operating-Systems-3rd-Edition/dp/0136006639%3FSubscriptionId%

3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0136006639.

[205] Hendrik Tews, Tjark Weber, Marcus Völp, Erik Poll, Marko van Eekelen, and Peter van Rossum. Nova micro-hypervisor verification. *Robin project deliverable D*, 13, 2008.

[206] Hendrik Tews et al. Formal methods in the robin project: Specification and verification of the nova microhypervisor. In *C/C++ Verification Workshop, Technical Report ICIS-R07015*, pages 59–68. Citeseer, 2007.

[207] Christopher Thompson, Maria Huntley, and Chad Link. Virtualization detection: New strategies and their effectiveness.

[208] Annette Tolnai and Sebastiaan Solms. The cloud's core virtual infrastructure security. In Sérgio Tenreiro de Magalhaes, Hamid Jahankhani, and AliG. Hessami, editors, *Global Security, Safety, and Sustainability*, volume 92 of *Communications in Computer and Information Science*, pages 19–27. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15716-5. doi: 10.1007/978-3-642-15717-2_3. URL http://dx.doi.org/10.1007/978-3-642-15717-2_3.

[209] Sunay Tripathi, Nicolas Droux, Thirumalai Srinivasan, and Kais Belgaied. Crossbow: From Hardware Virtualized NICs to Virtualized Networks. In *ACM Workshop on Virtualized Infrastructure Systems and Architectures*, 2009.

[210] K. Twidle, N. Dulay, E. Lupu, and M. Sloman. Ponder2: A Policy System for Autonomous Pervasive Environments. In *International Conference on Autonomic and Autonomous Systems (ICAS)*, 2009.

[211] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.163. URL http://dx.doi.org/10.1109/MC.2005.163.

[212] Amit Vasudevan, Bryan Parno, Ning Qu, VirgilD. Gligor, and Adrian Perrig. Lockdown: Towards a safe and practical architecture for security applications on commodity platforms. In Stefan Katzenbeisser, Edgar Weippl, L.Jean Camp, Melanie Volkamer, Mike Reiter, and Xinwen Zhang, editors, *Trust and Trustworthy Computing*, volume 7344 of *Lecture Notes in Computer Science*, pages 34–54. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30920-5. doi: 10.1007/978-3-642-30921-2_3. URL http://dx.doi.org/10.1007/978-3-642-30921-2_3.

[213] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *IEEE Symposium on Security and Privacy*, 2013.

[214] Anthony Velte and Toby Velte. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010. ISBN 0071614036, 9780071614030.

[215] Virtual Organization Managment Service. http://www.globus.org/grid_software/security/voms.php.

[216] AMD Virtualization. Amd-v nested paging. *White paper.[Online] Available: http://sites. amd. com/us/business/it-solutions/virtualization/Pages/amd-v. aspx*, 2008.

[217] VMWare. Understanding full virtualization, paravirtualization, and hardware assist. URL http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf.

[218] VMware. Performance evaluation of intel ept hardware assist, 2009. URL http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf.

[219] VMWare. Understanding memory resource management in vmware esx server, 2009. URL http://www.vmware.com/files/pdf/perf-vsphere-memory_management.pdf.

[220] A. Wailly, M. Lacoste, and H. Debar. Towards Multi-Layer Autonomic Isolation of Cloud Computing and Networking Resources. In *Workshop on Crytography and Security in Clouds (CSC)*, 2011.

[221] A. Wailly, M. Lacoste, and H. Debar. Towards multi-layer autonomic isolation of cloud computing and networking resources. In *Network and Information Systems Security (SAR-SSI), 2011 Conference on*, pages 1–9, 2011. doi: 10.1109/SAR-SSI.2011.5931358.

[222] Aurélien Wailly, Marc Lacoste, and Hervé Debar. KungFuVisor: Enabling Hypervisor Self-Defense. In *EUROSYS Doctoral Workshop (EURODW)*, 2012.

[223] Aurelien Wailly, Marc Lacoste, and Hervé Debar. VESPA: Multi-Layered Self-Protection for Cloud Resources. In *International Conference on Autonomic Computing (ICAC)*, 2012.

[224] Feifei Wang, Ping Chen, Bing Mao, and Li Xie. Randhyp: Preventing attacks via xen hypercall interface. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 138–149. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-30435-4. doi: 10.1007/978-3-642-30436-1_12. URL http://dx.doi.org/10.1007/978-3-642-30436-1_12.

[225] J. Wang, A. Stavrou, and A. Ghosh. HyperCheck: A Hardware-Assisted Integrity Monitor. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2010.

[226] Y.-M. Wang, Doug Beck, Binh Vo, and Chad Verbowski. Detecting Stealth Software with Strider GhostBuster. In *International Conference on Dependable Systems and Networks (DSN)*, 2005.

[227] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy*, 2010.

[228] Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. Isolating commodity hosted hypervisors with hyperlock. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 127–140, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168850. URL http://doi.acm.org/10.1145/2168836.2168850.

[229] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: practical capabilities for unix. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association. ISBN 888-7-6666-5555-4. URL http://dl.acm.org/citation.cfm?id=1929820.1929824.

[230] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. Plug into the supercloud. *Internet Computing, IEEE*, 17(2):28–34, 2013. ISSN 1089-7801. doi: 10.1109/MIC.2012.145.

[231] Rafal Wojtczuk. A stitch in time saves nine a stitch in time saves nine: A case of multiple os vulnerability. 2012. URL http://media.blackhat.com/bh-us-12/Briefings/Wojtczuk/BH_US_12_Wojtczuk_A_Stitch_In_Time_WP.pdf.

[232] Chiachih Wu, Zhi Wang, and Xuxian Jiang. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2013.

[233] Xen.org security team. cve-2013-1922 - qemu-nbd format-guessing due to missing format specification. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1922.

[234] P. Zech, M. Felderer, and R. Breu. Towards a model based security testing approach of cloud computing environments. In *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*, pages 47–56, 2012. doi: 10.1109/SERE-C.2012.11.

[235] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[236] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382230. URL http://doi.acm.org/10.1145/2382196.2382230.

[237] Zhao Zhang, Qiao-Yan Wen, and Wen Tang. An efficient mutation-based fuzz testing approach for detecting flaws of network protocol. In *Computer Science Service System (CSSS), 2012 International Conference on*, pages 814–817, 2012. doi: 10.1109/CSSS.2012.208.

[238] Feng Zhao, Yali Jiang, Guofu Xiang, Hai Jin, and Wenbin Jiang. Vrfps: A novel virtual machine-based real-time file protection system. In *Software Engineering Research, Management and Applications, 2009. SERA '09. 7th ACIS International Conference on*, pages 217–224, 2009. doi: 10.1109/SERA.2009.23.

DECLARATION

You made it :)

Paris, September 2014

Aurélien Wailly