Software-defined Security for Distributed Clouds

Maxime Compastié

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Software-defined Security for Distributed Clouds

THÈSE

présentée et soutenue publiquement le 18 Decembre 2018

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Maxime COMPASTIÉ

Composition du jury

Directeur de thèse :	Olivier FESTOR	Professeur, Université de Lorraine
Co-Directeur de thèse :	Rémi BADONNEL	Maître de Conférences, Université de Lorraine
Rapporteurs :	Nora CUPPENS	Professeur, IMT Atlantique
	Thierry GAYRAUD	Professeur, Université de Toulouse
Examinateurs :	Véronique LEGRAND	Professeure, CNAM
	Pierre-Etienne MOREAU	Professeur, Université de Lorraine
Examinateurs Invités :	Ruan HE	Chief Cloud Architect, Tencent
	Sok-Yen LOUI	Ingénieure de Recherche, Orange Labs

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

To my family.

Acknowledgement

Foremost, I would like to express gratitude to my supervisors, Pr. Olivier Festor and Dr. Rémi badonnel, for their guidance all along my PhD. Their precious advices and countless discussions have helped me in elaborating this work and making this PhD concrete. Their patience and their rigor have enabled me to a scientific level I couldn't have expected otherwise.

I am also accountable to Dr. Ruan HE and Ms. Sok-Yen LOUI for their time as industrial advisors. They were an inexhaustible source of inspiration, thoughtful comments and motivation. I would also thank my former project manager, Dr. Mohamed Kassi Lahlou, for contribibuting in bootstrapping this thesis and giving me the opportunity to join Orange.

I would like to thanks my PhD reviewers, Nora Cuppens and Thierry Gayraud, to have accepted to review my manuscript and their supporting comments. Besides, my thanks also goes to the whole members of my PhD commitee for having accepted to evaluate my work and attending my defense.

My thanks also go to the members APPCRM department at Orange Labs Service. Their warm welcome and good mood has given me the courage to begin my research and to move forward. Thanks to Hervé, Eliane, Roger, Sylvain, Guy, Guillaume, Alexandre D. Philippe K, Stéphane A., Stéphane C.. Rémi H., Bart, Jérôme, Christine, Thomas and Alioune. I also address very special thanks to Claire and Leo for our pinball contest.

Also, I want to express my gratitude to the members of the ARSEC department at Orange. They were source of support and fruitful discussion that have contributed to enrich the content of this work. Thanks to Ghada, Christophe, Matthieu, Philippe C., Philippe L.G., Jean-Philippe, Adam, Khaoula, Sebastien, Michael, Xavier G., Émilie, Marc, Xiao, Mahdi, Gilles, Alicia, Sandra, Alex, David, Benoît, Pascal, Paul, Kahina and Todor.

I also want to thanks my colleagues from the RESIST research team in the LORIA laboratory. Although I was located most of the time at Orange, I have greatly appreciated the time we spent together. Thanks to Nicolas, Pierre-Olivier, Soline, Thomas, Adrien, Alexandre M., Loïc, Daishi, Abdulqawi, Abir and Mingxiao

My final thanks are devoted to my friends and my family that have supported me during the whole of this experience.

Thanks to all those who has contributed directly or directly to this work !

Contents

List of Figures

Chapter 1

Introduction

Contents

1.1 Research Context

The development of the Internet has been leveraged by the deployment of large data centers providing computing resources (software applications, virtualized hardware equipments) that can be shared and combined to build elaborated services. These resources available in a metered manner are contributing to lower the capital and operational costs of infrastructures and their services. In particular, cloud computing introduces a new model for using and composing these resources deployed over the Internet. In that context, a cloud service provider (CSP) may offer different types of computing resources (with various levels of manageability) that can be exploited by customers for their own usage. According to the the NIST Institute [95], cloud computing can be characterized by five main properties:

- The *on-demand self-service* property enables customers to provision unilaterally resources for their services.
- The *broad network access* property leverages an ubiquitous access to a service through standard mechanisms, over the network, and more generally over the Internet.
- The *resource pooling* property contributes to an efficient management of resources that are shared to support services.

- The *rapid elasticity* property permits to adjust the resources to cope with the service workload. The resources can be easily instantiated or released in a dynamic manner.

- The *measured service* property characterized the on-demand nature of resources provided by cloud infrastructures. The fine-grained renting of these resources is in phase with resource management and billing optimization.

The cloud resources are typically exposed according to three different service models:

- The *Software as a Service (SaaS)* model provides whole applications to the customers exploiting the cloud infrastructure.

- The *Platform as a Service (PaaS)* enables the customers to build their own applications on a development and deployment environment provided by the cloud infrastructure.

- The *Infrastructure as a Service (IaaS)* permits to the customers a direct access to elementary resources related to the infrastructure, such as virtual machines and equipments.

These service models correspond to different levels of manageability for respectively cloud providers and cloud customers. The last service model is strongly dependent on virtualization techniques enabling the access and configuration of hardware resources by customers.

1.1.1 Involvement of Virtualization in Cloud Computing

Virtualization is an important foundation of cloud computing, that permits to isolate and share the software and hardware resources that are offered to customers. In particular, system virtualization enables the uncoupling of running software applications from hardware resources supporting their execution. This isolation in a virtual environment provides benefits in terms of resilience and security, as highlighted in [51]. The control, that an hypervisor (implementing the system virtualization) has over applications encapsulated in a virtual machine is an opportunity to frame their behaviors and detect any malicious activities that may relate to them. However, taking benefits of this situation also requires dedicated operations to handle the requirements and characteristics of applications, such as controlling the access to hardware resources, or inspecting their internal states.

1.1.2 Distribution of Resources over Cloud Environments

Cloud resources can be assigned to several customers with their specific requirements with respect to the operations to be performed over their resources. Each customer can be called a *tenant*, while the property for a given cloud to support multiple tenants is called *multi-tenancy*. Moreover, these resources can be distributed over several infrastructures. The property for multiple infrastructures to provide resources and collaborate with each others is called *multi-cloud*[1]. The conjunction of both multi-tenancy and multi-cloud defines the *distributed cloud*. The deployment of such a cloud increases the complexity of management. A typical example is the case of thousand of virtual machines belonging to different owners and hosted over dozens of data-centers, while each of these machines embeds their own applications and have dependencies with respect to data-centers on which they can be deployed on.

[1]The *inter-cloud* and *cloud-of-clouds* terminologies are considered as analog to *multi-cloud*

1.1.3 Exposure to Security Attacks

If the managed resources (or management directives) are related to security objectives of the cloud service, a mismanagement may jeopardize the security of tenant resources, as well as the security of the whole cloud service. This case is worsened by the broad network access property of cloud computing, as it increases the exposure of cloud services, making them an attractive target. A first axis to support security management is provided by the autonomic computing area [68]. It has opened new opportunities to tackle management and orchestration, by addressing the self-configuration, self-healing, self-optimization and self-protection of complex systems. It defines an approach for leveraging the automation of cloud environments, reducing the cost of their management and limiting error-prone manipulations. A second axis concerns the *software-defined* paradigm, which provides the matter to isolate resources from their control into two different planes. As an illustration, the *software-defined networking* uncouples the network equipments, corresponding to the data plane, from their management, corresponding to the control plane. The conjunction of these two axes provide new perspectives for automating and orchestrating the security of distributed clouds.

In that context, the main objective of this thesis is to design a unified and homogeneous security management plane for protecting distributed clouds, bridging the gap between virtualization and orchestration techniques.

1.2 Problem Statement

The building of a security management plane requires the identification of criteria to determine whether a resource is in an acceptable state and behaves correctly, or whether it requires management operations to comply with such a state. A security policy permits to formalize such criteria, and has then to be enacted by a management architecture and algorithms to enforce it. Such an architecture ensures that applications comply to the security policy. It detects cloud resources that are not aligned with the requirements of the policy, and forces them to be in conformance. In particular, we are relying in this thesis on the orchestration and configuration of security mechanisms that are software-defined. As depicted in Figure 1.1, several challenges have to be tackled to design such a security management plane with respect to the distribution of resources, the enforcement of security on them, and the adequacy with orchestration languages.

By definition, the protection of distributed clouds should take into account their multi-tenancy and multi-cloud properties. From a security policy perspective, multi-tenancy sustains the capability for different tenants to propose their own security policies, whose enforcement perimeters are limited to their sets of resources. It requires the security management plane to distinguish tenants during its processing. The multi-cloud property may also interfere with security enforcement. By deploying resources over infrastructures, it may leave some of them with no technical solution (due to the infrastructure) to conform with the security policy. The security management plane is therefore expected to consider infrastructure specificities, while orchestrating the security enforcement.

Distributed clouds also infer requirements over the security management plane. Cloud infrastructures come with a wide variety of allocated resources. The security management plane requires to provide an architecture able to handle indiscriminately all allocated cloud resources. Each resource comes with its own technical constraints, this impacting on how a security management operation should be performed to protect it. The management plane should remain agnostic to those specificities in its decision taking process, while the security enforcement should provide mechanisms to take them into account. The high availability constraint drives the operational

management of cloud resources. These resources may be exposed to security attacks all along their life-cycle. It is therefore required to control cloud resources, and maintain the security enforcement over them during their whole life-cycle.

Finally, the design of such a security management plane has to cope with the variety of protection requirements and related mechanisms. This supposes the extensibility of security policy languages, but also facilities to integrate enforcement mechanisms.

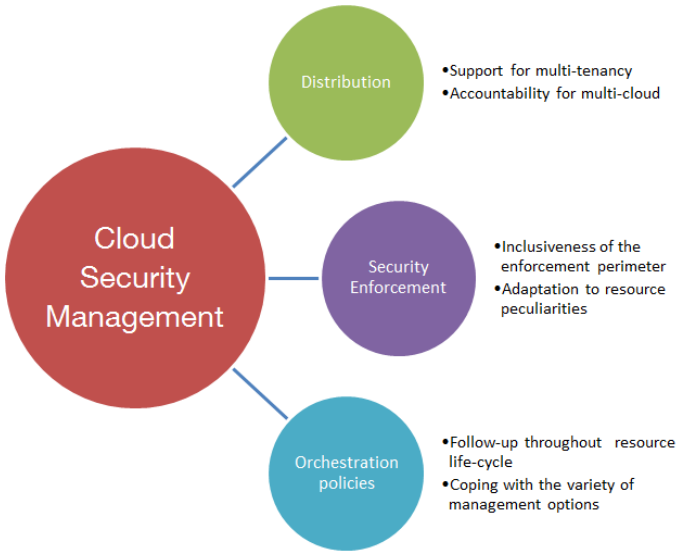


Figure 1.1: Challenges related to such a security management plane

In that context, this thesis addresses the design of an homogeneous security management plane for distributed clouds, with respect to the following questions: **(i) How can we design a dedicated security management plane to enact the automatic and homogeneous protection of distributed clouds? (ii) How can we ensure the compatibility of distributed cloud resources requiring protection based on this plane? (iii) How should the security requirements over cloud resources be specified to leverage a security orchestration from this plane?**

1.3 Contributions

This thesis proposes a software-defined security approach for distributed clouds.

We consider the programmability of security mechanisms that protect cloud resources, in order to leverage an homogeneous security management plane. This management plane serves as a support to orchestration activities. It is completed by tenant security plane conducting management operations adapted to each tenant, and supporting multiple infrastructures. This approach permits to tackle the complexity induced by the distribution of cloud resources. The security mechanisms are integrated within the resources requiring protection during the design of these resources.

This permits to ensure an holistic integration of security mechanisms within cloud resources, while exposing interfaces within the management plane. Moreover, securing resources before their allocation contributes to ensure a complete protection all along their life-cycle. In this thesis, we consider the case of distributed clouds composed of unikernel resources. We take benefits from unikernel properties to reduce the attack surface, and facilitate the integration of

security mechanisms at the resource design phase. The resources can be reconfigured or rebuilt to cope with security contextual changes.

In the meantime, we extend a cloud orchestration language, called TOSCA, to drive the building of these protected resources based on unikernels, and to support their configuration, according to different security levels.

In accordance with the proposed approach, the contributions of this dissertation are summarized as below.

1.3.1 Analysis of Virtualization Models for Cloud Security

As previously mentioned, virtualization constitutes a major building block for cloud computing environments, in order to uncouple computing resources from infrastructures supporting them. System virtualization based on hypervisors of type-I and type-II, has been reference solutions for decades. Nowadays, novel approaches for elaborating virtualized environments has been praised for improving performances and reducing technical management costs. The OS-level virtualization proposes to replace system virtualization by moving the abstraction layer to the interface between applications and their OS kernels. The unikernel-based virtualization consists in relying on system virtualization, but modifies the system architecture, through the insertion of a single application inside a dedicated OS kernel, in order to minimize unnecessary dependencies. In that context, we provide a description of the different virtualization models, and perform a comparative security analysis. In particular, we detail the vulnerabilities of these models with regard to existing threats, and evaluate their criticality by detailing related attacks. Finally, we point out several counter-measures and recommendations to avoid and mitigate these attacks in the context of cloud environments.

1.3.2 Software-Defined Security Architecture for Distributed Clouds

The management of resources in cloud environments is complicated by the multi-cloud and multi-tenancy properties that may appear when providing cloud services over distributed clouds. These two notions have an impact on security management, in order to specify and enforce security policies over resources that may be deployed over different infrastructures, and this with various tenants. This complexity is increased by the changes that may occur on resources in the context of cloud environments. Autonomic mechanisms permit to delegate some management tasks to the infrastructures themselves, contributing to the automation of cloud services. The software-defined paradigm also provides interesting perspectives for configuring security mechanisms and aligning security requirements over different infrastructures. In that context, our contribution consists in a logical architecture for supporting the programmability of security in the context of distributed clouds. We define a software-defined security approach aiming at a centralized and policy-driven control of the security in cloud environments. The proposed architecture addresses such software-defined security by taking into account the multi-cloud and multi-tenant properties of distributed clouds. The management is therefore performed at both the orchestration level and the tenant level, while relying on different types of policies. The first one coordinates the security management at the scale of a cloud service, while the second one addresses a specific tenant context, and is in charge of the security decision taking. This architecture serves a support for managing unikernel resources specifically built to address security requirements.

1.3.3 Generation of Protected Unikernel Resources

Enforcing a security policy over resources is subjected to the availability of adequate mechanisms to align their behavior with it. The lack of such mechanisms or a limited support for resources requiring this protection can result in (i) the inability to address all the resources in the enforcement perimeter, (ii) the inability to account the technical specificities of the resources, or (iii) the inability to enforce a protection all along their life-cycle. Unikernels constitute a thriving system architectural model that provides the necessary material for a comprehensive design of virtual machines. The limited tooling they carry at runtime imposes to properly build them, before their allocation. We therefore propose a framework for building unikernel images according to security requirements. We model these images as a set of modules which can undergo a configuration process and be built dynamically. This process serves as a basis for integrating security mechanisms in line with the policy requirements. Each change affecting these unikernels may imply the reconfiguration of images, their rebuilding, and the reallocation of their instances.

1.3.4 Extensions of a Cloud Orchestration Language

Defining a security policy at the scale of distributed clouds implies to cope with the heterogeneity of security models and mechanisms related to tenants and clouds. Each of them has to be taken into account, while considering inter-tenant and inter-cloud collaborations. In that context, the TOSCA language supports the specification and the orchestration of cloud services whose resources may be deployed over several infrastructures. The cloud service is described in the form of a topology with a set of resources and their relationships. It is possible to specify orchestration processes related to this service, these ones impacting on the state of resources and relationships. In our work, we propose two extensions for this TOSCA language, in order to drive the generation of unikernel resources and take into account security requirements. The first extension, called UniTOSCA, permits to detail the internal components of unikernel resources, and serves a support to automate the generation of adequate images integrating security mechanisms. The second extension, called SecTOSCA, enables the specification of security requirements, according to different security levels. An orchestration process can be associated to each security level. We provide a framework, based on these two extensions, capable of building and configuring protected cloud services. The software-defined security architecture brings the gap between the orchestraction activity and the generation of protected unikernel resources.

1.4 Outline of the Dissertation

The thesis dissertation is organized into seven chapters. Figure 1.2 describes their relationships.

We first conduct a state-of-the art on system virtualization with respect to cloud security, in Chapter 2. We compare different virtualization architectures and analyze their properties and criticalities to support cloud services. We elaborate several recommendations related to unikernels and security programmability, serving as requirements for the approach developed in this work.

In Chapter 3, we detail our software-defined security strategy capable of supporting the management of security mechanisms into a dedicated plane for distributed clouds. We detail the logical architecture implementing this approach, and supporting the multi-cloud and multi-tenant properties induced by distributed cloud environments. An evaluation of this architecture is performed, by confronting it to a set of realistic scenarios.

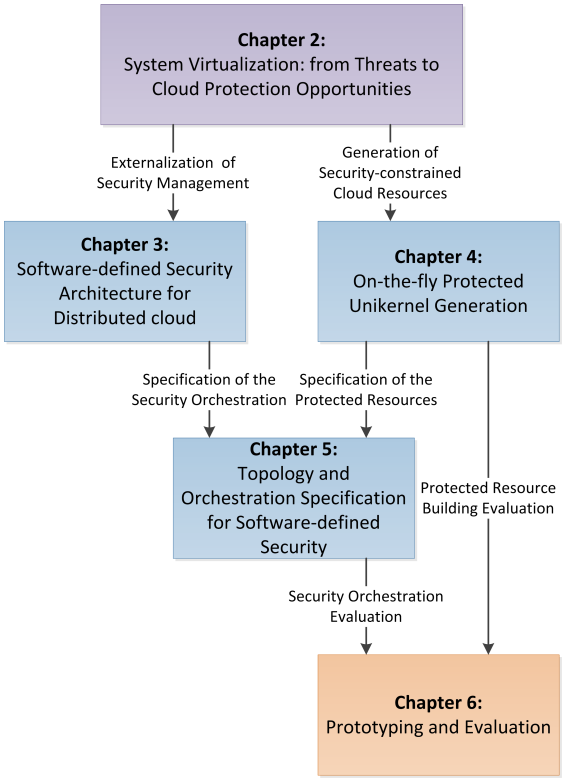


Figure 1.2: Organization of this document

We introduce in Chapter 4 the on-the-fly generation of unikernels, as a mean to leverage the security programmability of cloud resources. In particular, we present a framework for generating unikernel images that are constrained by security requirements. Their protection is enforced through the integration and configuration of dedicated mechanisms. We describe a proof-of-concept prototype and evaluate the approach in both a quantitative and qualitative manner.

Chapter 5 tackles the issue of the specification of security requirements. We rely on the TOSCA cloud orchestration language, and introduce two dedicated extensions. The first one addresses the internal description of unikernel resource,s while the second one supports the security requirements. This extended TOSCA language is then exploited by a framework to build and deploy protected cloud services. It drives both the generation of specific unikernel resources, and their configuration (or re-building) over time.

The overall approach is evaluated in Chapter 6. We describe several technical implementations of the framework, relying on the extended TOSCA language, the software-defined security mechanisms, and the generation of unikernel resources, and detail several series of experiments to quantify its performances.

Chapter 7 draws conclusions, wraps up the contributions of the thesis, and presents perspectives for future work.

Chapter 2

System Virtualization: from Threats to Cloud Protection Opportunities

Contents

2.1 Introduction

System virtualization is an important key to cloud environments [95]. In these environments, cloud service providers (CSP) expose resources to consumers, while these ones exploit these resources to run services or to compose new elaborated ones that can be offered to other ones. The multiplicity of stakeholders puts the security at stake at several levels and, consequently, questions the security of the underlying system virtualization: (i) the cloud service level agreement (SLA) specifies the availability of virtualized resources, (ii) the broad network access to cloud resources and the possible multi-tenancy requires the isolation of virtualized resources,

(iii) the growing involvement of governments through data protection regulation [43] puts also an additional pressure on the cloud service providers to ensure the confidentiality and integrity of resources. Recently, important efforts have been focused on performance improvement, with the emergence of new virtualization technologies capable of reducing virtualized environment footprints. On one hand, containerisation methods move the virtualization layer from the OS-hardware border to the OS-application one, in order to share the OS kernel. On the other hand, unikernel-based virtualization permits to simplify the complexity of virtual environments, by building minimal operating systems specifically built for dedicated applications.

The development and deployment of virtualization technologies are still in the early stage, while the technologies and products related are not yet mature. It therefore remains unclear how virtualization will fundamentally impact the landscape of cyber-defense, how it can help to improve security management, and more importantly, what kind of specific security threats it may introduce. We envision that the attack surface of virtualization could be significantly enlarged and multiplied due to the hypervisor isolation, cross-layer invocation, containerization, and so on. These factors make virtualization extremely difficult to establish in-depth defense security, requiring the critical security issues to be addressed in a holistic way. In particular, from a vertical perspective, applications inside a virtual machine (VM) are diversely incorporated with several layers and/or components, ranging from hypervisor to guest OS. Thus, any misconfigurations of a VM instance or hypervisor could eventually allow attackers to penetrate into the applications. From an horizontal perspective, as each layer is composed of heterogeneous components, the trust relationship between these components, either hardware or software, is hard to be established. It is therefore challenging to securely and seamlessly incorporate those components.

This state of the art addresses the security of system virtualization models used for cloud infrastructures. Contrarily to [137], we do not focus on the security question related to the operation of cloud environments. We provide an in-depth description of the virtualization models to conduct a security analysis. Contrarily to [123], the considered virtualization models are not limited to full-fledged VM system virtualization, but we also include OS-level virtualization and unikernel VM system virtualization. Our approach is close to [121], but our analysis is not bound to any cloud operation use-case.

To evaluate the benefits and limits of virtualization models for cloud security, the remainder of this chapter is organized as follows:

- First, we describe in Section 2.2 different virtualization models, including virtualization based on type-I and type-II hypervisors, OS-level virtualization and unikernel virtualization. The purpose is to give a basic understanding of their design principles and relationships, and to etablish a reference architecture that synthesizes these models and serves a support for the security analysis.

- Second, we analyze in Section 2.3 the security of the different virtualization models, based on this reference architecture. In that context, we identify and classify the vulnerabilities that may affect the components of this architecture. We then quantify their probability of occurrences, and detail the related security attacks, in view of existing security threats.

- Third, we infer in the last section different counter-measures and recommendations with respect to cloud security. This includes the generation of dedicated virtualized resources, the integration of security mechanisms at an early stage, and the opportunities offered by security programmability. These recommendations serve as a basis for the elaboration of our software-defined security approach for distributed cloud.

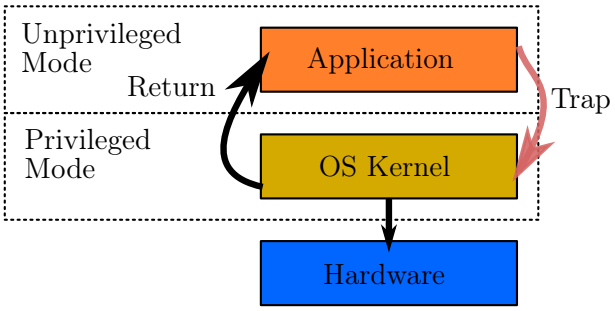


Figure 2.1: Regular system architecture schema

2.2 System Virtualization Models

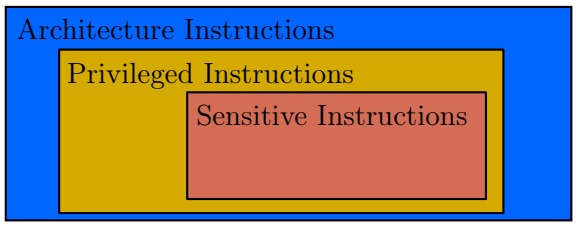
We will describe here different system virtualization models, in order to establish a reference architecture, that will serve as a basis for the security analysis.

2.2.1 Context

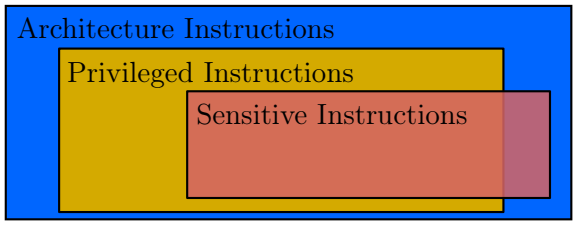
According to [125], system virtualization is the use of an encapsulating software layer surrounding an operating system, which conforms to the behavior expected from a physical hardware.

History. The system virtualization area has emerged since 1969, originally for general computing purposes such as multi-tasking or software-management. In the multi-tasking area, system virtualization has enabled hardware resource multiplexing, and thus, the concurrent execution of several OS, whereas the capability of enabling software execution conceived for one environment to run flawlessly on several others has paved the way for software management, at a time when hardware architectures were heterogeneous. However, more efficient approaches dealing with those issues have outrun the interest for system virtualization for decades. In 1990s, the emergence of personal computing and the limited compatibility of software appliances to one or a few number of operating systems has renewed the interest for system virtualization: the desktop virtualization permits the usage of an appliance developed on one operating system with another one. This is the main feature of products such as VMWare [154], founded in 1998. The emergence and spreading of cloud computing during 2010s has developed a new application field for the system virtualization: as the cloud paradigm relies on decoupling software resources from hardware resources, this one has naturally become a mean for enabling this paradigm.

Regular System Architecture. We define a system architecture according to a two-state mode execution environment (non privileged mode and privileged mode), as illustrated in Figure 2.1. Programs running in the *non privileged mode* rely on a subset of non-critical instructions sets (which will not impact the state of the machine, defined later), while programs in the *privileged mode* are allowed to employ the whole instruction set. If a program running in the non privileged mode wants to invoke a privileged instruction (instruction only allowed to run in the privileged mode), it sends an interruption called trap. Such a trap will trigger the execution of a predefined routine in the privileged mode. Practically speaking, on the x86 architecture, the privilege degree is divided into 4 levels called *protection rings*. The privileged mode corresponds to the value 0 of the *protection ring*, and the non-privileged mode is divided into ring 1, 2, 3. An



(a) Fully virtualizable architecture



(b) Not virtualizable architecture

Figure 2.2: Instruction set for a fully virtualizable architecture (a) and a not virtualizable architecture (b)

instruction is trapped if its ring value is higher than the ring value of the instruction it wants to invoke. Nowadays, most of the OS kernels are implemented in the ring 0 while user applications always resides in the ring 3. All user applications which are in the non privileged mode are loaded to the memory address space as *user space*. The programs in the privileged mode are then loaded to the memory address space as *kernel space*. One program can only run in one specific mode. When a user application wants to use a privileged instruction, it is then trapped through a specific interruption. The trapping mechanism stops the execution of the current application, stores the execution context, loads the corresponding routine, and executes the routine in the privileged mode. When the routine execution terminates, the execution context will be switched back to the user application which is in the non privileged mode. The set of *system calls* is defined as a subset of routines which can be invoked by applications in the non privileged mode. Appendix 8.2 presents an example of routines in charge of issuing a system call from an unprivileged application.

2.2.2 System Virtualization

The virtualization architecture derives from the regular system architecture, it enables concurrent execution of multiple OS. Within this architecture, there exist 2 types of instructions: sensitive and non-sensitive. Non-sensitive instructions can be executed directly on the hardware. However, since they modify the state of a machine which may impact a concurrent execution, they need to be controlled. The software module necessary to implement this virtualization architecture is called *Virtual Machine Monitor (VMM)* whose properties are defined in [125]. A VMM provisions an abstracted and isolated environment for each OS which is call *Virtual Machine (VM)*. It is a program interfacing between programs in each VM and hardware resources.

Fully Virtualizable Architecture

The fully virtualizable architecture is defined such that all its sensitive instructions are controlled. The VMM uses the trapping mechanism to realize this control. Hence, an architecture is con-

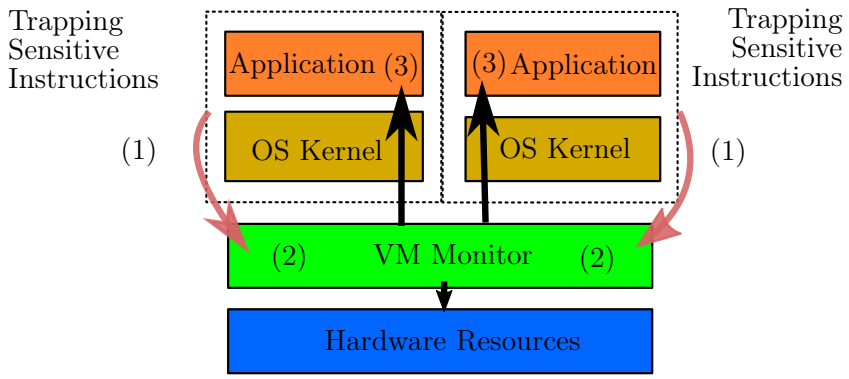


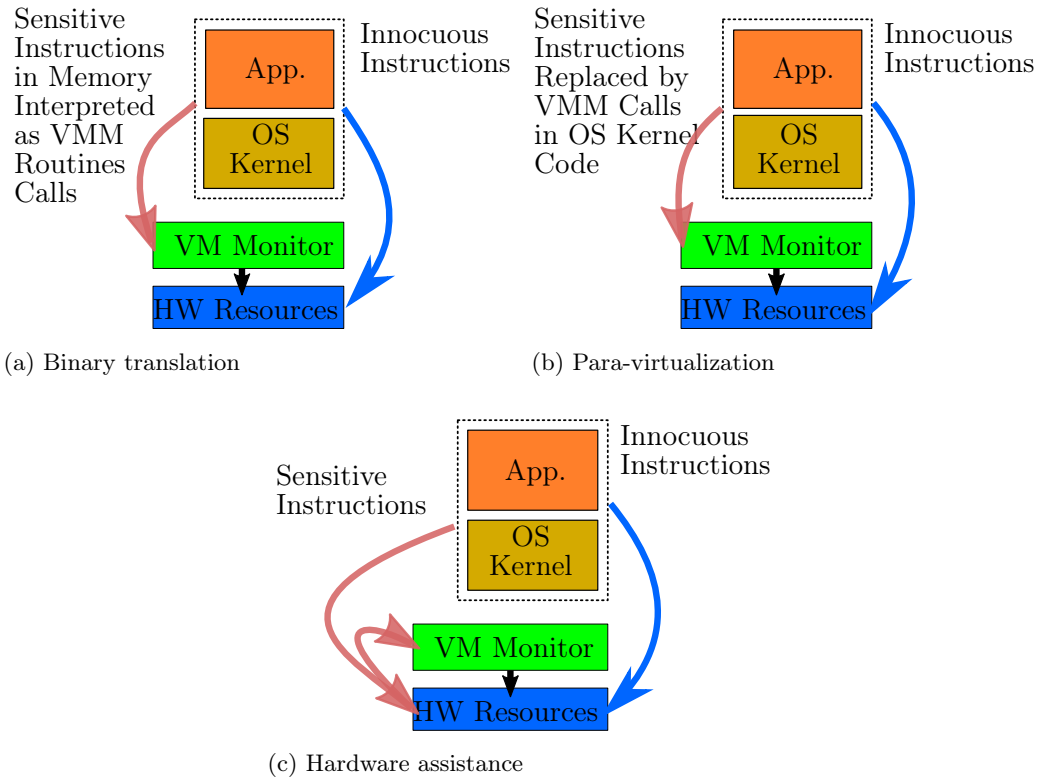
Figure 2.3: Virtual machine monitor

sidered as fully virtualizable if and only if all its sensitive instructions are part of the privileged instructions, as illustrated by Figure 2.2. Thus, intercepting the instructions from unprivileged mode permits to handle the execution of sensitive instructions through the privileged ones can be controlled through traps. Formally speaking, the VMM supporting the fully virtualizable architecture is expected to meet the following 3 properties: efficiency (the VMM allows innocuous instructions to be executed by the hardware directly, without intervention), resource control (no program can intervene in a hardware resource without a VMM allocator invocation), and equivalence (any program acts indistinguishably whether it is executed within a VM or not). Consequently, this architecture permits the existence of a VMM residing in privileged mode and lowering the whole virtual machine (including its privileged instructions) to the unprivileged mode: as illustrated in Figure 2.3, the VMM has to undertake (1) handling all trapping instructions from the VM, (2) simulating their effects according to sensitivity (innocuous instructions are directly executed, sensitive ones trigger VMM substitution routines), and then (3) gives back control to the VM. These three components enable the VMM to simulate privileged instructions: the *dispatcher* identifies the VMM routine corresponding to the instruction requiring simulation, the *allocator* allocates the resource required for the simulation, and the *interpreter* executes the VMM routines for the simulation.

Partially Virtualizable Architecture

The existence of sensitive but unprivileged instructions prevents the VMM from using trapping: such instructions do not trap and thus, do not trigger VMM instruction simulations, enabling the VM to access the hardware resources directly, contravening the VMM objectives. For instance, the x86 architecture does not comply with the fully virtualizable architecture [133]: some instructions including `SGDT` (Store Global Descriptor Table) and `SLDT` (Store Lobal Descriptor Table) are invokable in non-privileged mode, but access the sensitive CPU registers. Such an architecture refers as *partially virtualizable*, it is then necessary to treat those non-virtualizable instructions with other means than instruction trapping. Three methods have emerged as solutions for this issue: *binary translation*, *para-virtualization*, and *hardware-assistance*.

The **binary translation** method relies on a live interpretation of all sensitive instructions by VMM routines. Once a VM program has its code loaded into memory and its execution started, the VMM scrutinizes ongoing instructions to dynamically substitute sensitive ones by VMM routine calls before their usage. Those VMM routines are crafted to simulate their actions in a indistinguishably manner (*equivalence property*) in respect with VMM's *resource control* property.



(a) Binary translation

(b) Para-virtualization

(c) Hardware assistance

Figure 2.4: Virtualization methods in a partially virtualizable architecture

Non-sensitive instructions are not concerned by this process, ensuring the *efficiency property*. This process is performed at the binary level with compiled code. It is agnostic to the program source code and does not require any modification to it. But, the runtime translation induces an overhead due to the continuous memory access, rebounding on VM performances. Virtualization appliances implementing this method include *VMWare Workstation* [154], *QEMU* [127, 17] and *Oracle Virtualbox* [116].

The **para-virtualization** approach consists in the modification of the program source code employing sensitive instructions before their instantiations, substituting them by equivalent VMM routine calls (*equivalence property*) which are referred as *hypercalls*. Considering the case of an OS in a virtual machine, the para-virtualization related code modification are only located in the kernel of the VM, to make it cope with the VMM through hypercalls (*resource control property*). User applications remain unaffected (*efficiency property*). In practice on x86, both VMM and OS kernel reside in the protection ring 0 while unaffected user application resides in ring 3. As the para-virtualization integrates hypercalls in the program code before its execution, there is no more overhead due untimely memory access. This leads to better performance than binary translation. Its downside is the modification the program code to make it cope with the VMM. In numerous cases related to closed-source OS, its modification is not a considerable option, and consequently preventing the usage of this method. The Xen project [148] provides an appliance based on para-virtualization.

The **hardware-assisted virtualization** proposes a solution based on CPU capabilities. More precisely, if hardware-assisted virtualization is available, the CPU proposes a *root operation mode* intended to host the VMM and the *non-root operation mode* for VM code execution.

These modes are not related to the *privileged* and *non-privileged* ones. The root mode enables programs to define which instructions executed in the non-root one will trap back. Sensitive instructions executed in non-root mode can be configured to trap (*resource control* property), and to be simulated by a VMM routine in the root mode acting indistinctly (*equivalence property*). Innocuous instructions do not trap and are not affected by VMM (*efficiency property*). Using this method, virtualized programs are easier to build than those based on para-virtualization as they do not required source code modification to insert VMM routine calls. Moreover, the absence of dynamic translation contributes to better efficiency than the binary translation. On the other side, this method is based on CPU features that are not systematically implemented, bring the hardware issues into the prerequisite consideration for system virtualization.

VMM Implementation Types

As a common usage, VMMs (also called hypervisors) are implemented with extra-features enabling VM management (management console) and virtual hardware provisioning to VMs. In this state-of-the-art, we refer to a hypervisor as a VMM embedded with a management console, a memory manager, an input/output subsystem and a networking subsystem. We define a *host system* as the only set of OS and applications to have access to hardware resources not mitigated by the hypervisor. Conversely, *guest VM* alludes to the one with a restricted access to hardware resources. We distinguish two types of VMMs.

The **type-I hypervisor** alludes to a VMM directly operating the hardware resources. The host system runs alongside guest VMs, and both of them have to cope with the hypervisor. Their main difference at runtime is that the host system is permitted by hypervisor to access the hardware without restriction and have administrative privileges over the hypervisor while the guest VM are handled by virtual hardware environment. Thus, this approach corresponds to the classical VMM architecture. The Xen hypervisor [148, 12] and KVM [70] are two examples of type-I hypervisor implementations.

The **type-II hypervisor** runs as an application of the host system. It cannot directly access the hardware, but relies on the host OS routines to access hardware resources. Oracle Virtualbox [116] and QEMU [127] are two well-known type-II hypervisors.

2.2.3 OS-Level Virtualization

Hypervisors and VMs contribute to a proven virtualization architecture, currently being used in numerous use cases among which cloud computing, software testbed environments and software analysis framework. However, from an application isolation perspective, embedding OS kernel and virtual hardware environments with applications in a VM may be challenged from two optimizing issues. First, each VM embeds an OS kernel instance, meaning a resource cost not related to VM application exploitation. Second, the trapping mechanism induced by the VMM is a CPU-cycle greedy mechanism.

OS-level virtualization [2] attempts to increase efficiency by eliminating OS kernel from the isolation scope. It keeps only the set of applications and dependent libraries within a *container*. They run in parallel with other programs of the host system and both of them access the OS kernel and hardware resources, through the common OS kernel interfaces exposed to programs residing in user space (system calls) [73]. The host OS kernel is in charge of required virtual resources (i.e. filesystem access, networking) and container isolation enforcement. The container

[2] OS-level virtualization is also referred as *containerization*. However, some hypervisor-based solutions such as [67] have started as well to claim this terminology. For the sake of clarity, we dismiss its usage in this chapter.

execution environment benefits from kernel features and its configuration is performed by the *container engine*. This extra layer runs at the host system application level. By these means, OS-level virtualization enables almost-baremetal [45, 144] performances, outrunning the regular system virtualization ones, at the cost of coupling containers with a specific operating system. LXC [83], Docker [38] and gVisor [54] are three thriving examples of container engines supporting the Linux OS. Appendix 8.1 explores Linux kernel features implied in containerization.

2.2.4 Unikernel Virtualization

Containerization has brought an application-centric perspective in the virtualization debate, as the applications and their dependencies are the only embedded system parts in the portable environment. OS-level virtualization induces the dependency of in-container applications upon the host system OS kernel. It incidentally restricts the set of applications eligible to virtualization to the ones supporting this OS kernel. Additionally, in the performance area, containerized applications have to cope with host system OS kernel routines. Since containers cannot embed routines running in privileged mode, they cannot implement optimized privileged routines for dedicated purposes. Unikernel virtualization tackles these two issues, by transposing Library OS concept to the system virtualization era. Through the bypassing of legacy support constrains, it enables the refining of the system layer, reducing footprints of virtualized applications.

Library OS

A library OS exports the hardware resource management outside the OS kernel. In practice, this management relies on a set of libraries, implementable by every application whose access to the related resources is required. Each application is able to implement the most adapted resource managers to its mission while the dispatching of resource management amongst applications relieves the need of intermediate layers when accessing the hardware. In return, this architecture comes at the cost of a restricted portability and compatibility toward managed resources. The managers are expected to be resource-specific to attest their efficiency and application are not likely to implement as many managers as there are available hardware resources. The Drawbridge [126], the V++ cache kernel [29] and the Exokernel [42] projects are three architectures for library OS implementation exporting resource management in user space. The first converts Microsoft Windows 7 into a library-OS compliant architecture at the top of a VMM for process isolation purpose (*pico-process*) at the trade-off of keeping hardware management in the kernel space of the host OS to ensure compatibility. The second one only employs memory address space segregation in order to keep critical OS kernel features in the kernel space such as memory management, interprocess communications and thread management. The last one supports application multi-tasking and a secured hardware resource management, which is endorsed by libraries and is secured by a dedicated layer.Examples of Exokernel implementations include Aegis [42], ExOS [42] and XOmB [77].

Unikernels

Unikernels corresponds to an OS architecture featuring specialized, sealed, single purpose library OS. They can be run in virtual machine on the top of a hypervisor [91]. Each unikernel system addresses an application, its configuration, and the minimal dependencies required to run, and is configured and packed in an image, before their instantiation. Running inside a VM relieves those library OSes from supporting a vast amount of hardware configurations, and focus only on the virtual hardware environment exposed by an hypervisor. Additionally, the unikernel

architecture does not provide backward compatibility. This approach strengthens the lightness of unikernel VM instances compared to regular VMs. MirageOS [91] and IncludeOS [22] are two examples of thriving unikernel solutions.

A single unikernel image solely addresses the integration of one application, with its minimal set of dependencies. These dependencies address the application runtime, and libraries for both the software and the hardware resource management, in line with the library OS concept. Software components addressed by unikernel images can be tightly related to a development platform or remain independent. The first option imposes the usage of a dedicated tooling, paradigm and programming language to conceive a unikernel appliance. All the software components undergo the same processing (i.e. static analysis, code optimization, code compilation, linking and packing), contributing to the performances and the lightness of the resulting image. From the system architecture perspective, this option permits to include more system layer components in the assessment scope of programming language properties. The second option can be assimilated to the first one if a specific runtime environment is packed in the image. The *in-situ* software management capability is not provided in the unikernel image. Instead, they are performed externally, directly on the image before its instantiation, by the building of a toolstack, relying on a package manager, or be only uprooted on code inclusion mechanisms. This *ex-situ* approach contributes to lighten the unikernel footprint when instantiated.

An instantiated unikernel uses a single address space for its application and runtime, including the hardware management. This outwits the constraints and the implied overheads due to context switches issued by process scheduling or system calls and interrupts handling. Unikernel instances do not implement processes but still rely on multi-threading to support running parallel tasks. Built unikernel images can be conceived to be instantiable by a hypervisor, and cope with the related virtual hardware environment. This capability enablement relates on the inclusion of hypervisor-specific hardware resource management routines in unikernel images to address a specific virtual hardware environment. In a unikernel VM exploitation case, type-I hypervisors are usually privileged to minimize the layers mitigating the unikernel VM access to hardware resources. The routines of the unikernel VM are executed in the privileged mode to avoid the trapping overhead, when accessing the virtual hardware environment.

MirageOS

The MirageOS project associates the unikernel approach with the OCaml programming language and ecosystem [91, 90]. It provides a framework for turning OCaml-written appliances into unikernel images. This framework is constituted from a toolchain for preparing unikernel images, and the base runtime for executing the appliance. The project has initially undertaken the support of the Xen hypervisor, but the KVM and Bhyve hypervisors were also addressed latter.The provided toolchain encompasses development environment preparation through dependencies specification, solving and retrieval, and the unikernel image preparation through application code compilation, object linking and image packing. The prepared unikernel images are composed by both the appliance and the Mirage execution environment. As previously indicated, the first contains an application, its dependencies (libraries, runtimes and hardware management routines) and its configuration, and are closely tied to the OCaml eco-system. The applications are completely written in OCaml language, or partially written in C and tied back to the OCaml context with the `ocaml-ctype` binding. Dependencies are resolved thanks to the *OPAM* package manager before the mirage packing, and are written in the same fashion as applications. The configuration parameters are either hard-coded in application code or supplied at VM instantiation. The Mirage execution environment embedded in the image includes an OCaml

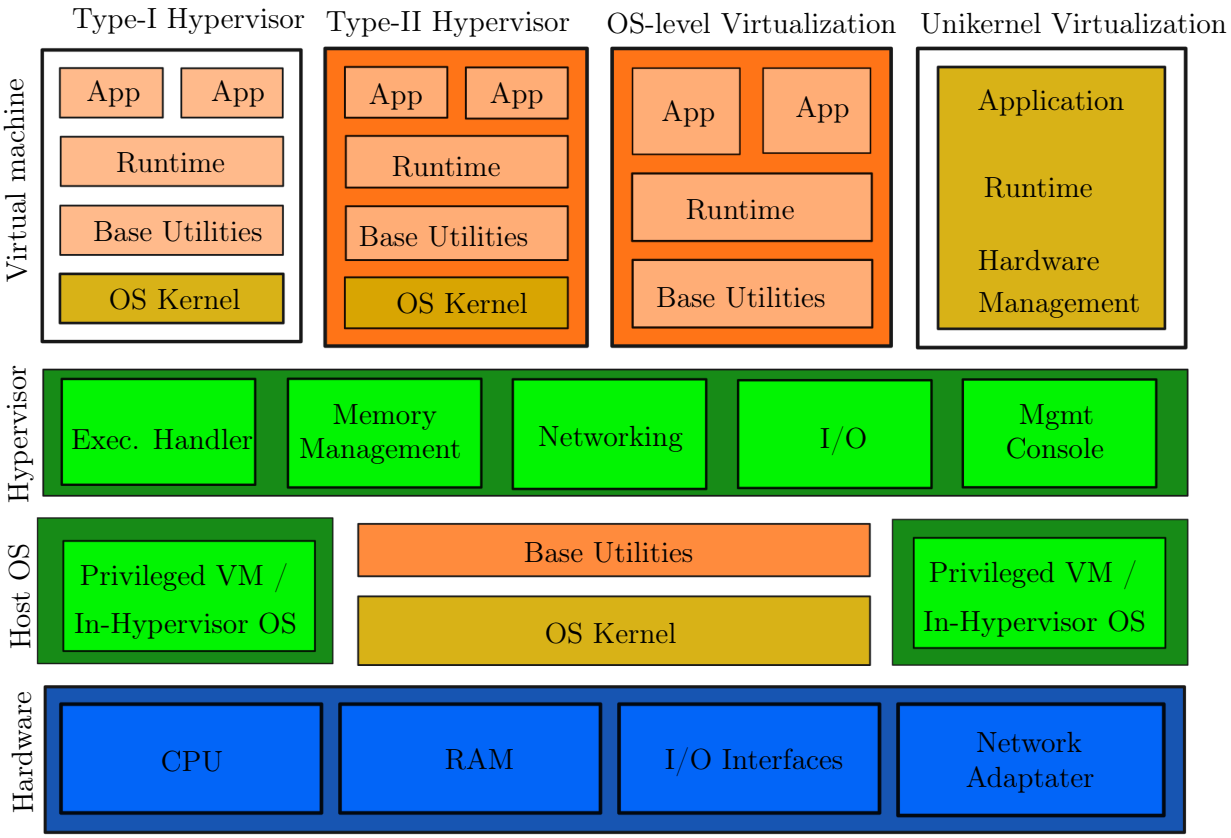


Figure 2.5: Virtualization reference architecture

runtime, and components depending of the targeted hypervisor. Mirage supports the Xen platform thanks to the PVBoot bootloader, the core library (*mirage-platform*) based on the mini-os project [134], and drivers for Xen `blkif` block storage and `vnetif` virtual network interface. The Solo5 project [159] contributes to KVM support (QEMU, ukvm [160]) and bhyve support in Mirage by providing compatible hardware management libraries: a core library (*mirage-solo5*) and interfaces management libraries (*mirage-block-solo5* for block devices,*mirage-net-solo5* for networking). The *syslinux* bootloader is employed.

2.2.5 Synthesis

We have presented different virtualization models, including virtualization based on hypervisor of type-I, virtualization based on hypervisor of type-II, OS-level virtualization and unikernel-based virtualization. We infer a virtualization reference architecture that synthesizes these virtualization models, and is depicted on Figure 2.5. This architecture will serve as a basis to our security analysis and is composed of four levels:

- The **hardware level**, represented at the bottom row of the figure, encompasses physical resource composing the host machine: The CPU, the volatile memory (RAM), the I/O interfaces (for persistent storage and extension cards) and the network adaptater for networking purpose.
- The **host OS level**, appearing at the so-called row in the figure, is exploited only for type-

II hypervisor and containers engines, and accounts for host OS kernel, base utilities and core libraries employed by those VM softwares. Type-I hypervisor and unikernel-related hypervisor may completely manage themselves hardware resources, or delegate complex management to privileged VM (e.g. storage or network devices handling).

- The **hypervisor level**, mentioned in the figure in the row above, referring as much to hypervisor as container-engine softwares. We assume it handles VMM capabilities (VM instructions trapping and sensitive instructions handling), VM memory management, VM networking (virtual network adaptater and inter-VM networks) and VM I/O interfaces. The hypervisor is administrated through a management console, enabling a system administrator or a service to manage the hypervisor and VM configurations.

- The **virtual machine level** refers to VMs, containers and unikernels, and is mentioned by the top row. In cases, this level includes applications, their configurations, their runtime environment and the libraries they depend on. Utilities are not systematically embedded in unikernel, since their features are embedded as application libraries, if they are required. In the same manner, OS kernel is neither integrated in containers (containers rely on host OS kernel) nor in unikernels (the OS kernel is reshaped in a set of libraries, provided with the application if required).

Virtualization provides important properties with respect to security [123], in particular:

- **Isolation:** VM access to physical resources is regulated by the hypervisor. This control affects inter-VM access as well, and confers resource isolation capability to virtualization. Moreover, by allocating quotas to the physical resources to VM, virtualization also promotes resource consumption isolation.

- **Oversight/introspection:** the hypervisor is able to inspect a VM resource usage and thus observes its internal state, leading to the oversight capability. As the hypervisor is also in charge of VM resource allocation, internal state modification may be performed, defining the introspection capability.

- **Snapshotting:** the hypervisor enforces access control amongst a VM and physical resources. This position permits the hypervisor to control the VM execution, by interrupting and resuming it. The content of allocated resources can be saved and reallocated as well, paving the way for VM internal state exportation and restoration (*VM snapshotting*). From a security perspective, this feature permits reversing back a VM in a insecured state to a previous secured one.

In the meantime, virtualization makes the system architecture more complex, by introducing new components (e.g. hypervisor) and redefining interactions between system architecture components (e.g. privileged instructions trapping). This may also introduce new security issues that are analyzed in the following section.

2.3 Security Analysis based on the Reference Architecture

We conduct a security analysis based on the reference architecture. First, we investigate the vulnerabilities of this architecture, draw a corresponding taxonomy, and evaluate qualitatively the criticality of each vulnerability with respect to virtualization models. We then determine the threats affecting this architecture, and analyze related attacks in view of identified vulnerabilities.

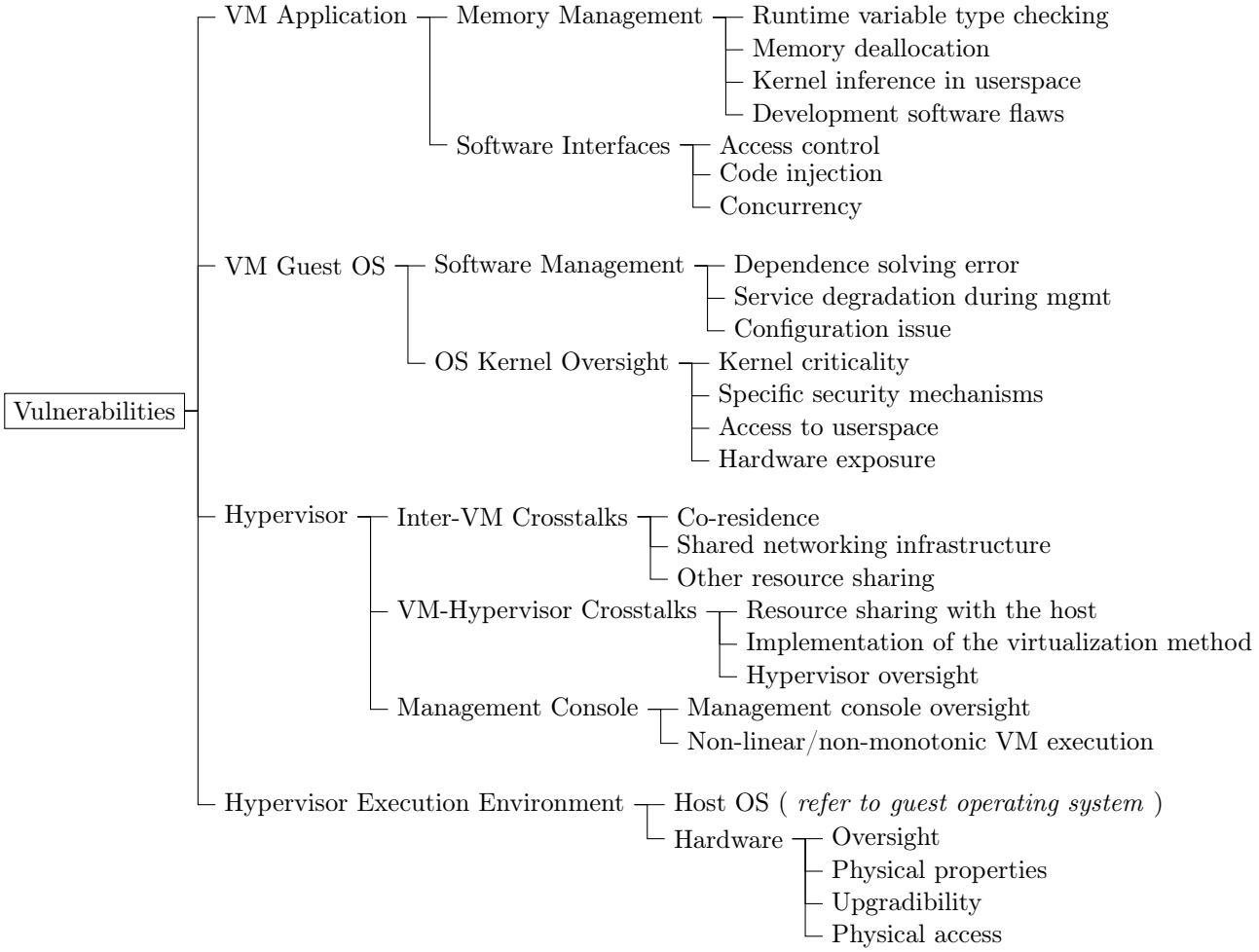


Figure 2.6: Classification of vulnerabilities

2.3.1 Identification of Vulnerabilities

We first identify and classify vulnerabilities related to the reference architecture, as depicted on Figure 2.6. The criticality of these vulnerabilities with respect to virtualization models is presented in Table 2.1. We investigate vulnerabilities carried by the VM as a part of a top-down study based on the reference architecture. We assume a VM hosting an application, its library dependencies and an OS kernel. We then consider the vulnerabilities affecting the hypervisor (or the container engine) running on the host machine. Nested virtualization is seen as a particular case of VM applications. In that context, we detail vulnerabilities related to (1) the VM application, (2) the VM guest OS, (3) the hypervisor, (4) the hypervisor execution environment.

VM Application

We describe vulnerabilities related to VM applications into two categories: memory management (with respect to runtime variable type checking, memory deallocation, kernel inference in user space, and developpement software flaws), and software interfaces (with respect to access control,

possible code injection, and concurrency).

Memory Management. An application instantiated in a VM relies on memory management routines provided by language interpreters, OS standard libraries and kernel internal routines. The lack of **type checking on variables at runtime** introduces trivial memory management based exploitation such as buffer overrun [57, 128] or integer overflows. This vulnerability is emphasized by the size of the code base of the applications deployed on virtual environments. Consequently, type-I and type-II hypervisors are the most subjected to this vulnerability, while OS-level virtualization benefits from OS kernel removal. As unikernels have their code base highly constrained, their virtualization model is the least affected. **Memory deallocation** also induces security issues. Except for interpretor-based execution environment equipped with memory garbage collector, no memory space is ensured to be automatically deallocated when stopped being used. This may lead to process memory leaks [57, 128, 21], and data persistence issues [143]. This vulnerability is related to traditional system architecture with multiple processes, and impacts type-I and type-II hypervisors as well as OS-based virtualization. The single application support from unikernel leaves physical memory cleaning to the hypervisor at application termination. **Kernel inferences in userspace** may also be possible. OS kernel is in charge of system management, by providing routines that application processes exploit. Nevertheless, the kernel is also able to manage this address space for various purposes, such as process swapping, free space reclamation [74], and exception handling. All virtualization models are affected by this vulnerability. While the vulnerability is obvious in the traditional system architecture, unikernels rely on a built-in runtime for the complete support of application. Finally, we should remind that any program comes with **software flaws** and bugs issued from their development cycles, inducing new exploitable vulnerabilities. This echoes to the unikernel pleading to limited code base, while OS-level containers benefit from not embedding an OS kernel.

Software Interfaces. In VM, programs rely on interfaces to communicate with the users and other programs. These ones are able to emit and to receive data through several channels including memory buffers, I/O interfaces and network sockets. In the area of **access control**, the improper implementation and configuration of authorization and authentication mechanisms can generate several vulnerabilities such as bad privileged assignment (configuration) or weak authentication [105, 63]. While virtualization models employing traditional system architecture have to carry multiple applications and base utilities with their own interfaces, unikernel models solely expose interfaces of the application it supports the execution of. Consequently, the access control vulnerability may be considered as less critical than in other virtualization models. Software interfaces may be used to corrupt the execution of a program, when it does not proceed to the necessary checks on the input data. This can lead to the exploitation of inconsistencies in data structures [104, 100], non-controlled string format exploitation [139] and, more directly, **code injection** [103, 65] (and former execution). If an interface can be accessed by several programs at a time, employing synchronization mechanisms is mandatory. Otherwise, it may be concerned by **concurrency** vulnerabilities [103, 112]. As all virtualization models support multi-threading in their virtualized environment, they are all equally affected by these vulnerabilities.

VM Guest OS

We consider two main categories of vulnerabilities related to guest OS: software management (including dependency solving, service degradation and configuration issues) and OS kernel

	Type-I Hypervisor	Type-II Hypervisor	OS-level Virtualization	Unikernel Virtualization
Runtime variable type checking	●	●	◖	○
Memory deallocation	●	●	●	◖
Kernel inference in userspace	◖	◖	◖	◖
Development software flaws	●	●	◖	○
Access control	●	●	●	○
Code injection	●	●	●	○
Concurrency	◖	◖	◖	◖
Dependence solving error	●	●	●	○
Service degradation during management	◖	◖	◖	○
Configuration issue	●	●	○	○
Kernel criticality	◖	◖	○	●
Specific security mechanisms	●	●	○	○
Access to userspace	●	●	○	●
Hardware exposure	●	●	○	◖
Co-residence	◖	◖	●	○
Shared networking	◖	◖	◖	◖
Other resource sharing	◖	◖	◖	◖
Resource sharing with the host	○	○	●	○
Implementation of virtualization method	◖	◖	●	◖
Hypervisor oversight	◖	◖	●	◖
Management console oversight	◖	◖	◖	◖
Non-linear/ non-monotonic VM execution	◖	◖	◖	◖
Host OS - Dependence solving error	◖	●	●	◖
Host OS - Service degradation during management	◖	●	●	◖
Host OS - Configuration Issue	◖	●	●	◖
Host OS kernel - Kernel criticality	◖	●	●	◖
Host OS kernel - Proper security mechanisms	◖	●	●	◖
Host OS kernel - Access to userspace	◖	●	●	◖
Host OS kernel - Hardware exposition	◖	●	●	◖
Hardware oversight	◖	◖	◖	◖
Hardware physical properties	◖	◖	◖	◖
Hardware upgradibility	◖	◖	◖	◖
Hardware physical access	◖	◖	◖	◖

Table 2.1: Occurence of vulnerabities with respect to virtualization models

oversight (including kernel criticality, specific security mechanisms, access to user space, and hardware exposure).

Software Management. When provisioning a virtual machine to prepare a service, the application is installed over its execution environment, and may be upgraded to a new version to apply security fixes, and eventually benefits from newly available features. In practice, such software management is performed by a package manager (e.g. APT [5]) or by a dedicated installer (E.g. 0install [1]) especially shaped for dedicated appliances. Nevertheless, **dependency solving** may generate vulnerabilities [37]. This affects particularly virtualization models supporting legacy system architectures. Software management can be performed on several points (e.g. base utilities, runtime, application itself) and imply multiple providers, increasing the vulnerable surface. On the opposite side, the software management in unikernels employs a pool of software components provided by the unikernel itself, and is performed only at build time. Besides, software management processes may lead to the **degradation of services**, as the upgrade usually requires to restart and reload configurations. This can be exploited to make the service unavailable during a certain time period. In every virtualization models, this vulnerability can be mitigate by performing upgrades on virtual environments that are not in production. This mitigation is natural for unikernels, as software management is performed ex-situ. Finally, software management may also induce **configuration** vulnerabilities. This may be due to packages whose initial configuration is too permissive, as shown by [60]. It may also relate to the update of configuration files during upgrades, with inconsistencies related to the new version of an application. Type-I and type-II virtualization models are the most affected due to the complexity of the architecture, while OS-level virtualization benefits from the absence of OS kernel in containers, and unikernels are protected by their inherent constrained nature.

OS Kernel Oversight. The OS kernel is a software running in privileged mode in charge of basic system resource management. The **criticality of the kernel** toward the system contributes to vulnerabilities with respect to tasks that it supports. These vulnerabilities are emphasized on monolithic kernel architectures (such as Linux), while they are further limited in micro-kernel architectures [81]. These vulnerabilities are bounded to the resilience of OS kernels and its isolation with the applications. OS-level containers are out of the scope, as they do not embed any OS kernels. Type-I and type-II virtualization models are the least affected, as the OS kernel exploits the system architecture features (i.e. privilege levels) to isolate itself from applications. On the opposite, unikernels do not propose strong barriers to isolate hardware management routines from the application ones. The OS kernel carries its own design principles, which may lead to the **inapplicability of security mechanisms** from the OS kernel to protect applications. For instance in Linux, process address space isolation is a non-sense in kernel space, as the notion of process is not defined there [21]. Unikernel is one known exception, as its design is based on a common framework for both the development of hardware resource management and applications [90]. Therefore, this vulnerability affects most significantly the type-I and type-II hypervisor virtualization models, while neither the OS-level nor the unikernel virtualization models are concerned. The **access to user space** is also another source of vulnerabilities. For instance, Linux kernel contains routines to read or write memory allocated to applications without restriction, (`copy_from_user()` and `copy_to_user()`), enabling it to intervene with no control [21]. The kernel has thus access to the application internal structures supporting both the application data and code. Finally, the OS kernel has an unrestricted access to hardware resources, since it runs in privileged mode. These resources can be either physical or virtual.

The **hardware exposure** constitutes another vulnerability. All the virtualization models (except OS-level one) are concerned. Unikernels are packed only with the hardware management routines that are necessary to the proper application operation, reducing potential risks.

Hypervisor

The hypervisor is subject to several vulnerabilities related to : inter-VM crosstalks (such as co-residence, common networking infrastructure, and other resource sharing issues), VM-hypervisor crosstalks (such as resource sharing with the host, implementation of virtualization, and hypervisor oversight issues) and the management console (such as hypervisor oversight and non-linear VM execution issues).

Inter-VM Crosstalks. By controlling VM access to physical hardware, the hypervisor enforces inter-VM isolation. Each VM is not able to access the resource or communicate with another, except if it is tolerated by the hypervisor. This isolation is especially challenged when several VMs are sharing the same hypervisor on the same hardware. This situation is referred as VM **co-residence** [132]. Indeed, VMs rely on the same resources (CPU, memory, I/O, networking interfaces), and the same hypervisor. Compromising one of these shared resources paves the way for creating a hidden channel mitigating the hypervisor isolation, as stated by [16]. This vulnerability affects the virtualization models in different manner: VMs with type-I hypervisor, type-II hypervisor and unikernels rely on a virtual hardware environment to interact with each other. These interfaces are specifically provisioned to each VM to let the hypervisor enforce the isolation. Unikernels restrict the interfaces, by supporting the least necessary virtual hardware resources for the unikernel execution. On the contrary, co-located containers in OS-level virtualization are part of a common host OS, that provides a wider pool of shared resources. This leaves them more vulnerable to these isolation issues. In the area of **networking infrastructures**, the hypervisor may provide a networking feature to support communications amongst hosted virtual environments and external resources. It can either rely on a virtualized network sustained only by the hypervisor or by an external program (e.g OpenVSwitch [115]). The networking configuration has to enforce the isolation amongst resources to only authorized communications. A misconfiguration leads to potential vulnerabilities enabling to bypass the isolation. Hypervisors may also feature **other resource sharing**, whose misconfiguration enables communications amongst VMs. The most obvious one is persistent storage sharing amongst several VMs (*volume*). Several VM may be an instance of a common image issued from a registry. The tampering of this image can compromise the related allocated VMs. This vulnerability affects equally all the considered virtualization models.

VM-Hypervisor Crosstalks. The hypervisor controls the execution of VMs, while interfacing between them and the host OS. It enforces the isolation between the host system supporting the hypervisor and the VM. Hypervisors and VMs communicate through interfaces. The virtual hardware environments is composed of virtualized hardware resources exposed to the VM and serving as a communication medium with the hypervisor. These resources may be subject to vulnerabilities [102, 94, 76]. Hypervisors may also provide private communication channels (with their VMs), that may not enforce proper security checks [98]. Vulnerabilities may also be due to the **implementation of virtualization methods**. From a software engineering perspective, hypervisor routines can be flawed, and carry software vulnerabilities [99]. It may also affect their implementations [129]. System virtualization may exploit hardware mechanisms that can mitigate the effects of software flaws. This contributes to make them a bit more secured.

Finally, the hypervisor controls the execution of VMs, and has an holistic view over the usage of virtualized resources. The **oversight of hypervisors** can be seen as a potential vulnerability, as they can access and modify the exposed virtual hardware resources [109]. Models related to system virtualization are less prone to these vulnerabilities. OS-level virtualization exposes the whole system call interface of the host OS kernel to containers, jeopardizing its isolation.

Management Console. For administration and monitoring purposes, each hypervisor proposes an interface for handling it and the hosted VMs. The management console can be used by administrators or other entities, such a cloud orchestrators.The first vulnerability relates the **oversight based on the management console**. The users of the management console have a complete control over VM life-cycles, and can modify the configuration of the virtual hardware environment. This control can be performed independently from what the software in VMs can support, affecting its execution. Users are also able to build new and unsecured communication channels [79]. All virtualization models are equally affected by this vulnerability, as each one proposes a management console. This control also permits to affect the **monotonicity of VM executions**, as shown in [123, 47]. For instance, performing a suspend-and-resume operation against a VM enables potential time and replay attacks. All virtualization models are equally affected by this vulnerability. These features are quite common in hypervisors, but may also be performed over container-based solutions.

Execution Environment of the Hypervisor

The execution environment (host OS or hardware) of hypervisors is also concerned by different vulnerabilities.

Host OS. Hypervisors supporting the execution of VMs have to rely on an OS to access physical hardware resources. This OS can be an internal part of the hypervisor (e.g VMware ESXI), be a VM itself controlled by the hypervisor (e.g. Dom0 in Xen), or be out of the control of the hypervisor (e.g. Oracle Virtualbox). The hypervisor is dependent of the behavior of this OS. The OS-level virtualization is affected as well by these vulnerabilities as it is an application supported by an host OS. We consider that Host OS vulnerabilities are the same than the ones detailed in Section 2.3.1. These vulnerabilities are predominant in type-II hypervisor and OS-level virtualization models, as host OSes are necessarily present. Type-I hypervisor and unikernel virtualization models are less affected, as the OSes can be managed and combined in the hypervisor.

Hardware. As nested virtualization is considered as out-of-scope of this analysis, we consider cases where the hypervisor (and eventually the host OS) are running on a bare-metal hardware. The hardware layer is affected by vulnerabilities, independently from the considered virtualization model. The hypervisor is a software component requiring a hardware infrastructure to operate. This hardware has an **oversight** above the running hypervisor, the running VMs and the current communications of the virtual interfaces exploiting the physical one. The persistent storage of the hardware infrastructures also affects the stopped VM, and their own storage. This makes hardware an attractive target to then compromise VMs. The **physical properties** of hardware can be exploited as side channel sources [72]. Hardware manufacturers embed firmwares with their devices, such as CPU microcodes and extension card firmwares. Some of them do not accept firmware **upgrades**, because of the design constraints or by the lack of support from manufacturers, leaving security flaws unpatched. Other ones support it, but may require the

system to run in a limited mode [162]. Finally, the **physical access** to hardware resources constitutes an important vulnerability, as components may be added or modified in the hardware layer, altering the behavior of underlying hypervisors and VMs.

2.3.2 Considered Threats and Attacks

In accordance with the methodology exposed in [149], we analyze the different attacks related to our reference architecture, according a set of six threats, namely spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privileges. In that context, we assume several hypotheses regarding the reference architecture and the attacker:

- The instantiated virtual machine applications expose interfaces that are accessible remotely.
- The attacker is located remotely, and can use the legitimately exposed interfaces.
- The hardware layer is not accessible to the attacker. We consider physical intrusion attacks out of the scope of system virtualization.
- The assets targeted by the attacker are applications and data located in virtual machines.
- The objective of the attacker is to impact on the availability (denial of service), the integrity (alteration) or the confidentiality (retrieving) of an asset.
- A component is considered as secured if not compromised, but may contain inherent software flaws known by the attacker. The attacker can exploit them to compromise a component, and gain influence over it.

We classify attacks with regard to threats, according to the notion of compromises: we consider a component as compromise, when the attacker is capable of executing an arbitrary code. We introduce a three-level classification based on the extension of [137]: compromise-free attacks (that are not related to compromises), compromising attacks (that enable compromises), and compromise-based attacks (that are based on compromises). Figure 2.7 provides an overview of this classification, while Table 2.2 details the relationships with the threats and the reference architecture.

2.3.3 Compromise-Free Attacks

We first detail compromise-free attacks, that do not aim at or do not require the compromise of a component of the architecture. As targets of these attacks, we mainly focus on virtual machines and hypervisors. However, the execution environment of hypervisors might also be considered to some extent, as an objective for an attacker.

Virtual Machine As a Target

We consider in this category **VM denial of service** attacks, which consist in blocking or disrupting the operation of a virtual machine. These attacks do not aim at gaining control over the resource nor exfiltrating data, but only affect its availability. Such an attack can be performed from the hypervisor (exploiting the management console oversight vulnerability), by using the management console (or a service controlling it) to shutdown the targeted virtual machine, as detailed in [79]. A more discret manner to proceed such a denial of service is to reconfigure the virtual hardware environment to cause the VM dysfunctioning (e.g. reducing allocated RAM)

	Spoofing	Tampering	Repudiation	Info. disclosure	Denial of service	Privilege elevation
Application	• Hyperjacking • VM Mobility	• Software Bad Design • Application Software Exploitation • VM Hopping • VM Escape to the VM	• VM Monitoring from the VM • VM Monitoring from the Host	• Application Software Exploitation • VM Monitoring from the VM • VM Monitoring from the Host • Inter-VM Com. Introspection	• VM Denial of Service	• Application Software Exploitation • VM Hopping • VM Escape to the VM
Runtime	• Hyperjacking • VM Mobility	• Software Bad Design • Application Software Exploitation • VM Hopping • VM Escape to the VM	• VM Monitoring from the VM • VM Monitoring from the Host	• Application Software Exploitation • VM Monitoring from the VM • VM monitoring from the Host • Inter-VM Com. Exploitation	• VM Denial of Service	• Application Software Exploitation • VM Hopping • VM Escape to the VM
OS Kernel	• Hyperjacking • VM Mobility	• Software Bad Design • OS Kernel Exploitation • VM Hopping • VM Escape to the VM	• VM Monitoring from the VM • VM Monitoring from the Host	• OS Kernel Exploitation • VM Monitoring from the Host • Inter-VM Com. Exploitation	• VM Denial of Service	• OS Kernel Exploitation • VM Hopping • VM Escape to the VM
Hypervisor		• Hypervisor Exploitation • VM Escape to the Host	• Command Control Channel Exploitation • Hypervisor Exploitation	• Hypervisor Exploitation • Computation Res. Monopolisation	• Hypervisor Denial of Service	• VM Escape to host
Host OS Kernel		• VM Escape to the Host			• Hypervisor Denial of Service	• VM Escape to the Host
Hardware		• Firmware Exploitation		• Firmware Exploitation	• Hypervisor Denial of Service	• Firmware Exploitation

Table 2.2: Relationships amongst threats and attacks

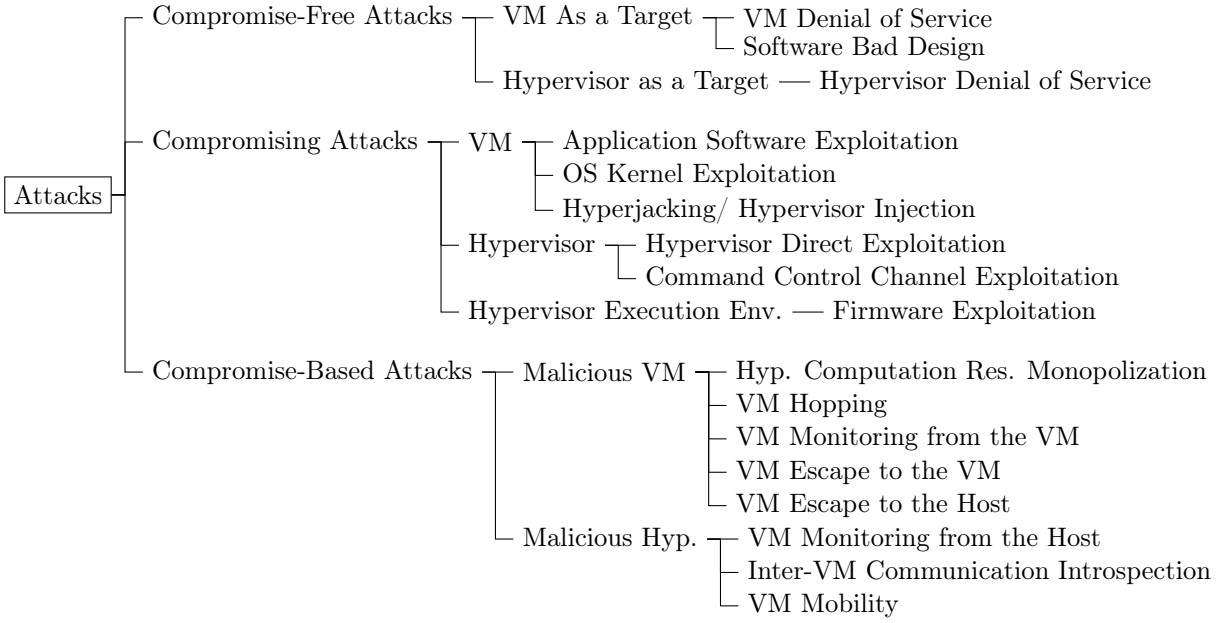


Figure 2.7: Classification of attacks

to reduce the footprint of the attack. The network may also be used to perform such attacks. Based on the *hardware exposure* vulnerability of guest and host OS kernels, a massive workload from the network may make the VM applications and the VM OS kernels unaivaible. This can typically consist in a distributed denial of service attack. Software flaws are also sources of software failures (e.g. [101] related to a given appliance) and can be exploited to perform denial of service attacks. VM appliances may be attacked using their intrinsic flaws, or the ones of their dependencies. These attacks exploit mostly the *memory management* and *concurrency* vulnerabilities, but might also rely on *software interface* vulnerabilities. The OS kernel of the VM may also be used for that respect (*kernel criticality* vulnerabilities). According to [53], linux kernel faults are mainly due to unsecure software development.

We then focus on attacks exploiting **software bad design**. Software applications may carry flaws in their design, but not specifically related to their code. These attacks can be performed without requiring the execution of arbitrary code, and thus, without requiring a compromise. They relates to the tampering threat of in-VM applications, their runtime, the base utilities and the in-VM OS kernel. For instance, misconfigurations are typically exploited to perform such attacks, as detailed in [82, 13]. They may the consequence of initial inadequate settings, or changes due to software upgrades or manual administration. The impact of such attacks can be high, when they relate to authorization and authentication rules (*access control* vulnerabilities). Software management may also be exploited for that purpose. Outdated dependencies, required by VM appliances, can introduce vulnerabilities. The upgrade process can also lead to man-in-the-middle (MITM) attacks during the collection of packages, as pointed out in [87, 24]. In particular, it is possible to prevent the execution of software updates, including security patches. These attacks relate to the *dependency solving error* vulnerabilities and the *memory management* vulnerabilities. Finally, attacks may also be based on the *non-linear/non-monotonic execution* vulnerability. As exposed in [123], an attack might restore a VM to a vulnerable state, by using the hypervisor facilities.

Hypervisor as a Target

The compromise-free attacks also concern denials of service related to hypervisors, similar to the previous ones:

- The management console represents an attack vector, since it can shut down the hypervisor (management console oversight vulnerability).
- Network protocols can be used to flood the hypervisor and/or its execution environment based on DDoS attacks.
- Hypervisors are also composed of software components, having their own vulnerabilities that are exploitable by attackers.

We can also notice that VMs can indirectly contribute to hypervisor denial of service. For instance, authors of [123] point out VM sprawling flaws, enabling the rapid spread of misbehaving VMs. This leads to the exhaustion of the hypervisor resources.

2.3.4 Compromising Attacks

The second main category of our classification is compromising attacks. This compromise often serves as a basis for more elaborated attacks. We analyze here attacks leading to the compromise of components of the reference virtualization architecture. We distinguish attacks affecting the virtual machines, the hypervisor and its execution environment.

Virtual Machines

The attacks compromising virtual machines first include **software exploitation** attacks, that consist in forcing an application in a VM to execute an arbitrary code. As the application runs in an unprivileged mode, the injected payload cannot rely on privileged instruction sets. This affects in-VM applications, their runtime and base utilities. Considered actions include the tampering of resources, the privilege elevation of an attacker, and the disclosure of information. This attack may exploit several vulnerabilities. In addition to *memory management* vulnerabilities [124], the *code injection* in software interfaces may also be used for the insertion of payloads. The *dependency solving error* [3] in software management may also prevent software upgrades, in order to maintain security flaws. Compromised OS kernels can also contribute to such attacks (*access to userspace* vulnerabilities). Authors of [28] show applications can be subverted by an OS kernel based on forged returns from system calls. Other flaws, such as misconfigurations and additive dependencies, may be used from unprotected interfaces. The work in [87, 24] also investigate a MITM attack against the software management system, with the objective of inserting or maintaining unprotected interfaces on applications, accessible by the attacker.

We can also consider **OS kernel exploitation** attacks aiming at the execution of an arbitrary code by the kernel. These attacks benefits from the privileged execution mode, which extends the attack surface. A concrete example of such attack is the use of rootkits [114], enabling an attacker to gain the whole control over a system. The related threats include tampering, privilege elevation, and disclosure of information. A compromised application in the user space may be sufficient, but not necessary for performing these attacks. Moreover, modular OS kernels increase the risks, as the module loading process constitutes an additional payload vector.

Finally, we can point out **hyperjacking / hypervisor injection** attacks. A typically example is the VM-based rootkit (VMBR) attack, which permits to integrate a malicious hypervisor

at the bottom side of the operating system. Its introspection capability enables it to overpass any security mechanisms [123]. Subvirt [69] proposes persistent VMBRs targeting both Windows OS and Linux and hosting malicious services undetectable from these OS. The blue pill attack, described in [25], is a VMBR attack capable of infecting an host without any reboot. The VMBR attack takes advantages from the *hypervisor oversight* vulnerability to get introspection capabilities, and to be not detectable by OS-level detection mechanisms.

Hypervisor

The compromising attacks may also target the hypervisor of the virtualization architecture. Amongst these attacks, we can highlight **hypervisor direct exploitation** attacks. Even if they are not expected to be exposed as VM appliances they host, hypervisors remain sensitive to these direct attacks. The threats related to these attacks affect the hypervisor itself, by tampering it, repudiating the traceability of its behavior, disclosing information of its configuration and the VM configuration, and elevating the privilege of a VM user to those of the hypervisor administrator. For instance, the work in [161] illustrates the Xen hypervisor (and the privileged domain) compromise by a rootkit using a weakness in the DMA implementation and the Xen debug register. [117] also identifies several common attacks against hypervisors, by considering the case of local attackers. Hypervisor extension mechanisms (*extension packs*, *module framework*) are also critical components for the hypervisor, as they are capable of loading code, such as a malicious payload, in its instance. Such attack is described in [117], where the *Xen loadable module framework* permits to load arbitrary code in the Xen address space. These attacks take advantage of the *resource sharing with host* and *virtualization implementation* vulnerabilities.

Another important compromising attacks correspond to **command/control channel exploitation** attacks. The command/control channel is a privileged communication medium between the hypervisor and a VM, as depicted in [155]. This threat targets the hypervisor, through the repudiation of malicious activities in a VM. Also, the work in [26] investigate how to use the command channel testing feature, in order to recognize a virtual machine environment.These attacks may typically be related to the *resource sharing with host* vulnerability.

Execution Environment of the Hypervisor

Compromising attacks may also include **firmware exploitation** attacks against the execution environment of the hypervisor. Hypervisors are running at the top of a hardware layer, except in the case of nested virtualization. They are therefore constrained by the hardware layer, which is composed of devices with their own firmwares. These firmwares may carry their own flaws, providing the necessary material to an attacker to compromise them. The corresponding threats are the hardware tampering, the information disclosure based on side channels, and the elevation of privileges to get control on the software components running over the hardware. For instance, such an attack is showcased in [163], in order to compromise the firmware of network interface card. These attacks are emphasized by the hard constraints regarding the upgrade procedure of these firmwares. Such upgrades are typically limited by the degradation of services during the upgrade process, or by the lack of support from manufacturers. These attacks are mainly based on the *hardware oversight* and *hardware upgradability* vulnerabilities.

2.3.5 Compromise-Based Attacks

The last category of attacks are compromise-based attacks. The two first categories (compromise-free and compromising attacks) are often the first steps towards establishing a more sophisticated

attacks targeting a virtualization architecture. Consequently, we consider that compromise-based attacks require the prior compromise of a component to be performed. These attacks typically include the case of malicious virtual machines attacking the remainder of the virtualization architecture, and the case of a malicious hypervisor trying to snoop on the virtual machines it is in charge of.

Malicious Virtual Machines

The first category of attacks relies on malicious virtual machines. An hypervisor hosts several VMs, which share the same physical resources. They can be competing to access these resources. The **hypervisor resource monopolization** attacks consist in a malicious VM taking control on an hypervisor to gain an exclusive access to these resources. This leads to the violation of the hypervisor resource isolation. These attacks on physical CPU are described in [168]: a Xen scheduler vulnerability is used by a malicious virtual machine to steal compute cycles to other co-located machines. They typically exploit the *co-residence* and the *virtualization method implementation* vulnerabilities.

The **VM hopping** attacks consist in a malicious VM that directly targets another VM from the virtualization environment. The related threat concerns the VM applications, their runtime and utilities, and their OS kernel. They permit the tampering and the elevation of privilege in the virtualization architecture. These attacks can typically use *common networking infrastructure* and *other resource sharing* vulnerabilities to access the targeted VM. It can exploit *software interfaces*, *configuration* and *hardware exposure* vulnerabilities to perform a compromise.

The attacks regarding the **VM monitoring from a VM** consist in collecting information about a VM, without compromising it. They can typically rely on a passive monitoring of another virtual machine. The related threats are about VM applications, their runtime and their OS kernel, through the disclosure of information and the breaching of the non-repudiation principle. These attacks are mainly conditioned by the exploitation of side channels, and the co-residence of VMs. [14] illustrates the by-passing of an OS protection based on hypervisor side channels. The work in [165] makes use of co-residence and physical properties to affect memory pages owned by other VMs, to proceed to in-memory information leakage or even to build a hidden channel between both VMs. The co-residence issue is especially challenging in a public cloud infrastructure. The work in [132] details the steps to reach a VM co-residence with the targeted VM. Finally, authors of [167] explore how co-residence can contribute to cryptography key leaks based on CPU L1-cache. These attacks are based on the *hardware physical properties*, the *virtualization method implementation* and the *inter-VM crosstalks* vulnerabilities.

The **VM escape to a VM** attack aims at compromising the hypervisor, in order to access another VM. It is very similar to a VM hopping, and corresponds to the same threats, but relies on the hypervisor compromise to break the isolation. *VM-hypervisor crosstalks* vulnerabilities are typically involved in these attacks. The **VM escape to an host** relies on the hypervisor compromise from the malicious VM, in order to gain further control. In most of the cases, these attacks target legitimate interfaces between the hypervisor and the malicious VM, in order to compromise the hypervisor, such as the VENOM attack [48] and the Cloudburst attack [76]. These attacks take advantage of the *VM-hypervisor crosstalks* vulnerabilities.

Malicious Hypervisor

The second category of attacks relies on a malicious hypervisor. The **VM introspection / monitoring** attack from the host exploits a malicious hypervisor to analyse the behavior of a

	VM Denial of Service	Software Bad Design	Hyp. Denial of Service	Application Software Exploitation	OS Kernel Exploitation	Hyperjacking	Hyp. Direct Exploitation	Command Control Channel Exploitation	Firmware Exploitation	Hyp. Computation Resources Monopolization	VM Hopping	VM Monitoring from VM	VM Escape to the VM	VM Escape to the Host	VM Monitoring from the Host	Inter-VM Communication Introspection	VM Mobility
Runtime variable type checking	✔	✔		✔													
Memory deallocation	✔	✔		✔													
Kernel inference in userspace	✔	✔		✔													
Development software flaws	✔	✔		✔													
Access control	✔	✔									✔						
Possible code injection	✔			✔							✔						
Concurrence vulnerability	✔										✔						
Dependence solving error		✔		✔													
Service degradation during mgmt																	
Configuration issue		✔		✔							✔						
Kernel criticality	✔																
Proper security mechanisms	✔				✔												
Access to userspace	✔			✔													
Hardware exposition	✔										✔						
Co-residence										✔		✔					
Common networking infrastructure			✔								✔	✔					
Other resource sharing			✔								✔	✔					
Resource sharing with host			✔				✔	✔					✔	✔	✔	✔	
Virtualization method implementation							✔			✔		✔	✔	✔			
Hypervisor oversight						✔							✔	✔	✔	✔	
Management console oversight	✔		✔														✔
Non-linear/monotonicity VM execution		✔															✔
Host OS - Dependence solving error																	
Host OS - Service degradation during mgmt																	
Host OS - Configuration Issue																	
Host OS kernel - Kernel criticality			✔														
Host OS kernel - Proper security mechanisms			✔														
Host OS kernel - Access to userspace			✔														
Host OS kernel - Hardware exposition			✔														
Hardware oversight									✔								
Hardware physical properties												✔					
Hardware upgradibility									✔								
Hardware physical access																	

Table 2.3: Relationships amongst attacks and vulnerabilities

VM and infer its internal state. Different introspection techniques are presented in [109] to track the activities of a virtual machine. For instance, [79] demonstrates an approach for extracting a secret key from a memory dump. Authors of [107] propose a strategy to extract secrets from a AMD SEV-protected VM: a malicious hypervisor manipulating the mapping between guest physical memory and the host physical memory enables a malicious distant client to obtain secret data. A shown in [79], the management console can also serve as a support to perform such a malicious monitoring. These attacks are typically based on *resource sharing with the host* and *hypervisor oversight* vulnerabilities.

The **inter-VM communication introspection** attacks may also provide interesting information regading VM communications with other hosted VMs or hosts, through I/O or networking subsystems that are handled by the hypervisor. This raises privacy issues related to communication interception and introspection. The threat affects the OS kernel, the runtime environment and the applications in VMs, causing potential disclosure of information. The *hypervisor oversight* and *resource sharing with host* vulnerabilities contribute to these attacks. The textbfVM mobility attacks consists in an attacker using the export feature of the hypervisor to obtain the VM storage device, the virtual hardware environment configuration, and the current state of the memory and vCPU related to a VM. They can typically be based on the *management console* vulnerabilities.

As a synthesis of this section, Figure 2.3 describes the relationships amongst the vulnerabilities of the reference architecture and the attacks that they leverage.

2.4 Counter-Measures

After having classified security attacks, we propose different counter-measures and recommendations with regard to the reference architecture. We consider two major requirements with respect to these counter-measures. They should have a minimal impact on the operability of protected resources, and should not impact on the security benefits brought by the virtualization itself, in particular in our cloud computing context. We provide a threefold classification of security counter-measures. It includes counter-measures consisting in integrating security mechanisms at the design of a resource, counter-measures aiming at reducing the attack surface of the resource, and counter-measures enabling a higher adaptation through security programmability.

2.4.1 Integration of security mechanisms at design time

A first category of counter-measures consists in addressing the protection of resources (or components) at design time through the integration of security mechanisms. This may concern both the virtual machines and the hypervisor of the architecture.

Protection of virtual machines

The protection of virtual machine resources can be performed at various levels. Counter-measures can be considered at the **OS kernel**. For instance, the address space layout randomization (ASLR) method permits to prevent the exploitation of memory management vulnerabilities [140]. From an architectural viewpoint, monolithic OS kernels may facilitate security attacks due to the lack of isolation amongst kernel subsystems. PerspicuOS [36] addresses this issue, by fragmenting the OS kernel code with an isolation of privileges. The nested kernel is the OS kernel subpart whose memory access is privileged, while the outer kernel corresponds to subsystems relying on inner kernel API for memory access. These APIs virtualize the MMU (Memory Management

Unit) to enforce protection on the outer kernel memory access. The isolation of the OS kernel components argues in favor of unikernels, since the minimalism of their architecture leads to the dispatching of hardware management routines accross several unikernel VMs.

The counter-measures also concern the **applications** of virtual machines, as they represent a major entry point. The variety of applications and runtime environments to be considered in that context overpasses the scope of this state-of-the-art. This often supposes specialized security mechanisms dedicated to a given application. Protecting an application from an untrusted execution environment is not a novel issue. Solutions such as XOMOS [80] address this issue, but suppose important requirements regarding the architecture. Shielding the application execution is also an interesting approach to protect applications from malicious OS. For instance, Haven [15] exploits the Intel SGX technology to provide a protective layer against OS-based and physical attacks. The usage of unikernels constitutes also a solution to have a minimal code base and reduce the attack surface. In addition, as the runtime environments are constrained by the unikernel framework, this restricts the complexity of their support from a security perspective.

From a **software management** viewpoint, package managers may be securized and may contribute to prevent attacks on virtual machines. Approaches such as [24] introduce package signature mechanisms to deal with man-in-the-middle attacks, as well as package alterations. Complementarily, [3] exploits SAT solving techniques to cope with dependency solving issues. In the case of unikernels, the software management is performed outside the virtual machines, reducing the consequences of its flaws to the supported appliance.

Protection of the hypervisor

The hypervisor provides an **execution environment** to the VMs, enabling them to run in accordance with the VMM properties [125]. In that context, [146] proposes the use of hardware to protect VMs from the host, by enforcing isolation through hardware resource access management and privacy based on a trusted platform module (TPM). However, this counter-measure supposes important prerequisites on the hardware architecture, which may not be considered in practice. Additionally, [79] introduces a method for filtering hypercalls and checking their integrity at the invokation. Virtual CPU context integrity checks as well as virtual memory encryptions are enforced during the execution of the hypervisor. In the area of OS-level virtualization, the Intel safeguard extension is used by SCONE [7] to protect containers against untrusted execution environments. This framework provides the necessary building blocks to design enclaved linux containers, that are resilient against tampering attacks from the OS, with a limited trust computing base (TCB). SCONE uses a limited pool of threads that are mapped with ones from the untrusted OS. The system calls are performed asynchronously with the assistance of a module outside the enclave. Complementary, the exploitation of command channels can be avoided by obfuscation methods, as detailed in [26]. [168] also describes a solution to CPU monopolization attacks, by switching virtual CPU scheduling to alternative methods (e.g. randomized scheduler, Bernouilli scheduler and uniform scheduler) to limit flaw exploitations.

The **granularity** of the hypervisor architecture also impacts on security. In order to avoid a monolithic architecture, [34] uses hypervisor virtualization features to decompose management OS capabilities across dedicated service VMs. These VMs are framed by coercitive security constraints, such as hypercall restrictions, security audits and frequent reboots. However, this approach only focuses on the management console segmentation, and does not address the internal subsystems of the hypervisor. The NOVA [145] hypervisor goes further, by proposing a micro-hypervisor architecture. A minimum trusted computing base (TCB) resides on the host kernel (*micro-hypervisor*), while each virtual machine dedicated VMMs are executed in the user

space, meaning host device drivers and remaining services. The lack of maturity of this approach introduces limitations regarding guest OS compatibities.

It is also important to cope with **networking and storage** security. In that context, the enforcement of access control security policies on VM resources is explored by the sHype hypervisor [138]. This solution, based on Xen, enables the enforcement of policies related to the mandatory access control (MAC) on VM resource access, including network communications, standard I/O communications and shared memory. More precisely, the reference monitor enforces the policy on event channel operations, shared memory requesting and domain management operations. The presented solution is limited to access control enforcement with the MAC model on a single hypervisor instance. TVDSEC [150] addresses the access control policy enforcement over several hypervisor instances. In return, only the enforcement over control flows is considered. Networking may also serve for performing resource quarantines: the NICE framework [32] exploits a network controller to put in a quarantine state VMs that are potentially compromised.

2.4.2 Minimization of the attack surface

A second category of counter-measures concern the minimization of the attack surface, by verifying the properties and restricting the capabilities of resources. In particular we focus on verification and capability dropping techniques.

Formal verification can be applied to both the OS kernel and the applications of virtual machines. It permits to assess the security properties of the components of the architecture, but also to verify the design of security mechanisms (e.g. cryptographic libraries). For instance, Klein et al. [71] introduces the design and the implementation of a micro-kernel, SEL4, featuring formal specifications. It relies on the Haskell purely functional programming language. The source code can be automatically translated into a formal specification that is then checked. The Hyperkernel [111] project proposes an OS kernel that can be assessed using the Z3 SMT prover. This OS makes use of the hardware-based virtualization feature to enforce an isolation between the kernel residing in root-mode and the process running in non-root mode. These approaches may imply an important cost for modifying/extending the code framed for formal specifications, and often suppose to define a modular architecture to prove the properties of components independently. Alternative approaches have also emerged to protect applications and guarantee security properties, without implying formal checking. Typically, virtual ghost [35] is a framework to protect user applications from an untrusted operating system. It enables the applications to control their own operating system-proof sandbox, and a layer interleaving the OS and the hardware is responsible for enforcing the sandbox isolation.

It is also possible to apply **capability dropping** techniques. Hardening methods are part of them and permit to reduce the attack surface of a system, by imposin a system to reduce its attack surface, by constraining parameter values and imposing (security) software components. Most applications come with their specific recommendations with respect to hardening. In addition to authorization restrictions and unnecessary software dropping, address space layout randomization enforcement is recommended. The images of resources can also be hardened in such a way that their applications are dedicated to a specific purpose, while considering their lifetime is restricted in accordance. A directory of unikernels is proposed in [89], facilitating the usage of constrained and ephemeral virtualized resources. The **outsourcing of in-VM software management** contributes also to this hardening. Most operating system images are provided with their software management tools. Therefore, the presence of those tools in the VM image can be questioned, as most Delegating the software management to the image manufacturing process contributes to protect VM applications against related attacks and compromises.

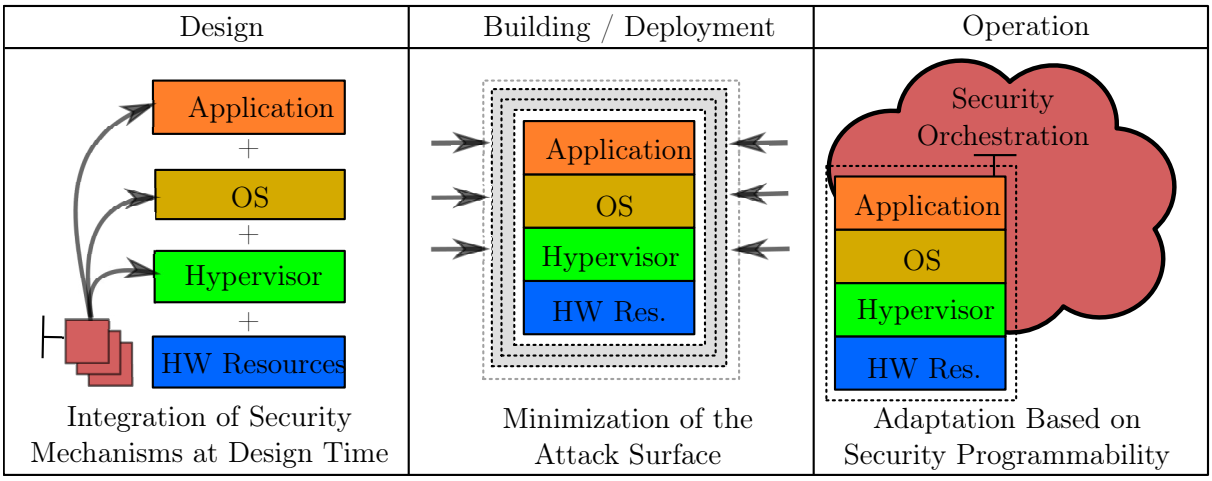


Figure 2.8: Illustrated Synthesis of Recommendations with Respect to Cloud Protection.

An example of such outsourcing is given in [24], in accordance with security criteria. However, this approach supposes that (i) the VM image manufacturing chain is trusted and (ii) the VM instances cannot receive security updates, leaving long-leaving VM instances vulnerable to new attacks, and arguing in favor of short-living VM instances. In that context, we can also restrict the virtual hardware environment. Embedding unused hardware increases the exposure to attacks. An example of such protection is given in [160], where the authors export the virtual hardware environment stack to the VM itself. The solution is based on a modified version of the KVM hypervisor delegating its monitor to the VMs. It increases the isolation between the hypervisor and the VM, and permits a finely tailored virtual hardware environment for VMs. However, this implies important modifications in the code base of the hypervisor, requiring a per-hypervisor engineering, which has not be performed on other hypervisors than KVM.

2.4.3 Adaptation based on security programmability

Counter-measures are efficient at reducing the attack surface, but they have to constantly be adapted over time to cope with new threats and attacks. The programmability of resources and their security mechanisms contribute to this required adaptation. It can be driven by an orchestration activity relying on monitoring results.

The **monitoring** of resources can be performed based on introspection techniques. For instance, VMwatcher [64] permits to determine the internal state (memory and filesystem) of virtual machines. This enables to outsource some security mechanisms such as in-VM malware detection. Authors of [30] propose similar techniques, but applied from the guest OS during the execution phase. Active monitoring approaches, such as the *LARES* architecture [122], are also applied to observe virtual machines based on dedicated agents or probes. Detection approaches, such as the NICKLE framework [131], permit to identify and prevent rootkits. Finally, the hypervisor introspection is also applicable to VM virtual hardware environments, as developed by Slick [10] and TVDSEC [150]. The first one relies on a modification of the hypervisor (experimented in KVM) to track the activity of VMs on virtual storage, while the second one focuses on tracking VM networking flows. The monitoring may also consist in the identification of vulnerable configurations related to components of the virtualization architecture. For instance, the approach in [13] proposes a SAT solving method for supporting resource management. It permits to detect the presence of configuration vulnerabilities in preventive manner, considering

the maintenance operations that are available.

The analysis of monitoring results can then drive **orchestration** activities in virtualization and cloud environments. An efficient and consistent orchestration is an important aspect to deploy and coordinate security mechanisms. Some efforts, such as [120], exploits orchestration languages to support access control policies in the area of virtualized network functions (VNF). The security policy mining contributes to the coherency of the orchestration by extracting high-level policy from entities enforcing low-level ones. Work in [55] applies this approach to extract network-wide access control policy from the configuration of multiple firewalls. Orchestration depends on the **programmability** of resources and their security mechanisms. To enforce security decisions taken by an orchestrator, the resources of the virtualization architecture can be dynamically reconfigured through programmability facilities. These reconfigurations enable to reduce the attack surface (e.g. setting up authentication) or to mitigate the attacks (e.g. isolating a compromised host through firewalling). Considered resources include the different components of the virtualization architecture, but also security mechanisms. We can distinguish different cases: (1) components supporting reconfigurations at runtime can be managed through their configuration interfaces, (2) statically-configured components that are restartable, can be modified by changing the static configuration alteration and restarting the instance, (3) components that are non restartable (nor reconfigurable at runtime) can be addressed through dedicated methodologies, such as dynamic software updates developed in [156] using IncludeOS unikernels. We can also notice some recovery techniques, such as the TASER intrusion recovery system [50], capable of tracking the activity of a virtual machine, and recoverying its configuration to clean state after an attack.

2.5 Conclusions

In this chapter we have described different virtualization models, namely virtualization based on type-I and type-II hypervisors, OS-level virtualization and unikernel virtualization. We have detailed their design principles and relationships in order to infer a reference architecture, that serves as a support to our security analysis. We have then analyzed the different vulnerabilities that may affect the components of this architecture, and identified related attacks, in view of existing security threats. We have then highlight several counter-measures and recommendations with respect to our cloud security context.These recommendations are synthesized on Fig. 2.8, in line with different phases related to the lifecycle of cloud resources. These recommendations concern more specifically the integration of security mechanisms at the design time, the minimization of the attack surface, and the programmability of security mechanisms. They serve as a basis to the elaboration of our software-defined security approach for distributed clouds. Unikernel virtualization offers different opportunities to reduce the attack surface based on their simplified architecture, and to integrate security mechanisms at an early stage during the building of unikernel images. The programmability of resources, in phase with the security orchestration, can drive the generation of such unikernel resources. In the next chapters, we will first introduce an architecture supporting security programmability in a multi-cloud and multi-tenant environment. We will then define different mechanisms for the on-the-fly generation of protected unikernel resources. Finally, we will extend an orchestration language to drive the generation of the protected resources, in accordance with different security levels.

Chapter 3

SDSec Architecture for Distributed Clouds

Contents

3.1 Introduction

Cloud computing permits to build elaborated services and applications based on multiple computing resources, such as virtual machines, network devices, software components, themselves provided as services that can be easily deployed through the Internet. It supports an as-a-service scheme with a transparent access to resources and an outsourcing of management activities to the cloud provider. This separation of roles permits to optimize the allocation and usage of resources, but it may also introduces additional management complexity due to the cloud distributed nature. We remind that the cloud infrastructure and its applications may typically be divided into isolated sets of resources called *tenants*, corresponding to different ownerships and requirements, defining the *multi-tenancy* property. Another property comes to the facts that the resources may be distributed among several infrastructures, as each of them may be specialized

in a dedicated processing. Distributed cloud can be defined by the conjunction of the *multi-tenancy* and *multi-cloud* properties. In this context, security management has become a major challenge. The dynamics of cloud infrastructures induced by their on-demand self-service, rapid elasticity and distribution has outrun traditional security management, while the ubiquity and high availability of cloud resources make them attractive targets for attackers [4].

Exploiting autonomic and programmability mechanisms opens new perspectives for enabling such a security management. Autonomic strategies permit to address the scalability issues induced by large and distributed cloud infrastructure resources, by delegating part of the management tasks to the environment itself. In particular, this may concern the management tasks related to self-protection and self-configuration. The goal being to maintain a security level for services in a distributed cloud, based on the activation or deactivation of available countermeasures. In addition, network programmability has already shown its advantage for software-defined networking by separating the network infrastructure into two separate planes, i.e. the data plane and the control plane, and contributing to its dynamic configuration and adaptation. Similarly, there is an important need for supporting *software-defined security* in distributed clouds.

In this chapter, we propose a software-defined security architecture for distributed clouds, and detail its main components and their interactions. We also provide an analysis of this solution based on a set of validation scenarios corresponding to a realistic use case. The architecture permits to specify security policies, and enables their automatic enforcement in a multi-tenant and multi-cloud environment. Security mechanisms are dynamically configured based on changes that may occur in the distributed cloud. This architecture serves as a first building block for our approach, and is complemented by the on-the-fly generation of protected unikernels (detailed in the next chapter), and the extension of the TOSCA orchestration language (described in Chapter 5).

The remainder of this chapter is organized as follow: Section 3.2 gives an overview of existing work related to our architecture, complementarily to the state-of-the-art. We then describe the concept of software-defined security in Section 3.3. The proposed architecture, its components and their interactions are detailed in Section 3.4. We then evaluate this architecture through a set of validation scenarios, and discuss implementation considerations in Section 3.5. Finally, we summarize the contribution of this chapter in Section 3.6.

3.2 Related Work

The security of cloud infrastructures has already been largely explored in the literature, in particular through different architectures and frameworks. [157] focuses on challenges related to policy-based security management in that context. It includes the specification of a cloud security policy, the support for security decisions, as well as the certification of security components. In the same manner, the *TCloud* framework [153, 18] proposes to enforce a security policy with a trusted cloud stack (*trustworthy openstack*). This policy does not mainly focus on the protection of cloud resources, but rather consider the resilience through the use of multiple infrastructures. The framework provides infrastructure-level and platform-level security components, such as an access-control-as-a-service subsystem, a bayesian-fault-tolerant middleware and a relational database service exploiting other Tcloud components as building blocks. They might be compatible with multi-cloud environments, with a hardened build of OpenStack environments. However, these solutions do not specifically address self-configuration mechanisms, nor the management issues induced by multi-cloud and multi-tenancy properties. The *Iceman* architecture [40] discusses a secure federated inter-cloud identity management approach. It promotes identity self-service

for each cloud of a federation, increasing the granularity of the access control in those multi-cloud environments. The authors of [135] proposes a cloud management framework able to deal with multi-tenancy, based on the XACML architecture (described later in this section). The security policy model combines benefits from the role-based access, the attributes-based access and the task-based access models. But this one is limited to access control policies and cannot support other security features. Similarly, PaaSword [152] is a framework employing the XACML architecture to ensure the privacy of user data manipulated by cloud applications through data encryption and access control. The framework proposes an holistic and proactive definition of entities involved in the security policy through the annotation of the model source code. It is however not designed to support additional security features or the modification of the entity used by the security policy without a complete cloud application redesign. The proposed architecture is independent from the available security mechanisms and addresses their self-configuration in a distributed cloud. The SDAC framework [58] generates security policies for the access control in multi-tenant and distributed infrastructure environments: the architecture uncouples access control enforcement in a dedicated plane from its logic. The later is organized in three planes. First, the application plane specifies and customizes the cloud-wide security policy. Second, the control plane hosts the cloud-wide policy taking into account each tenant of the cloud service. Third, the policy plane contains tenant specific policies, and interfaces with the enforcers. The main shortcoming of this architecture is its support for security features, which is limited to access control.

In the area of programmability, software-defined networking (SDN) permits to distinguish the *control plane* making decisions about where the traffic should be sent from the *data plane* forwarding packets. This paradigm enables a dynamic and adaptive policy enforcement. It may also serve as a framework to leverage infrastructures security, as stated in [56]. For instance, the *Flowtags* framework described in [44] enables the integration of middleboxes for chaining security functions whose composition is supported by SDN controller. [75] proposes a framework for enforcing a network security policy through a set of middleboxes. The solution considers an exhaustive packet tagging to direct network traffic to a set of middleboxes corresponding to the security policy. But, this solution only considers middleboxes for instantiating security mechanisms. Authors of [61] explore how to exploit the SDN paradigm to build a chain of security functions, including intrusion detection systems and firewalls, to protect smart devices. IETF has also explored the interfacing of SDN-based security services with network security functions [62]. The NICE framework [32] exploits the switch programmability to mitigate network intrusions and to manage compromised hosts in cloud environment. The logic of the vulnerable hosts and intrusions is dedicated to an IDS (*NICE-A*) and a control center. The later stores VM profiles regarding vulnerabilities, alerts and traffics, while an an attack analyzer supports the decision taking process and a network controller drives the programmable network equipments. Such approaches take advantage of SDN with respect to security policy enforcement. Important efforts have also focused on the verification of security chains. For instance, *VeriCon* [11] combines a language for specifying SDN policies with an approach to check whether a policy verifies invariants expressed in predicate logic. In the same manner, *FlowChecker* [141] represents the network as a binary decision diagram (BDD), whereas properties are expressed in computation tree logic (CTL). However, the model based on BDDs requires a certain expertise of formal methods, which cannot be generally expected from network operators. In our context, we are focusing on a software-defined security framework to protect distributed cloud, in line with software-defined networking, but not limited to network enforcement considerations. The autonomic computing paradigm provides a framework for self-management activities, and relies on several main areas: self-configuration, self-optimization, self-protection and self-healing [68].

Although it does not bring a formal distributed cloud support, it may introduce the negotiation among independent components. This approach may deal with exhaustive enforcement issues, as autonomic components can continuously enforce the security policy and adapt to the changes in their action perimeters. The two previous paradigms do not directly deal with distributed cloud issues. But, they provide important building blocks for supporting security policy enforcement and defining a security management architecture in that context.

With respect to security policies, the OASIS consortium introduces two standardized languages: XACML (eXtensible Access Control Markup Language) for representing and exchanging security policies [19] and SAML (Security Assertion Markup Language) for specifying security statements [92]. The XACML specification also defines a reference architecture for the specified policy. It isolates the components in charge of the enforcement (referred as policy enforcement point) from those hosting the policy (the policy retrieval point) supporting enacting logic (the policy decision point and the policy information point) and the policy administration interface (the policy administration point). The main limitation are twofold. First, the XACML specification is closely related to the access control security feature and makes other feature hardly addressable. Second, this architecture is dedicated to a reactive enforcement of a security policy. It does not provide any workflow to proactively enforce a security policy through Policy Enforcement Points (PEP). This approach remains relevant, as the XACML defines modular components for the security enforcement. Primelife [6] presents an extension of XACML accounting for user privacy during the decision taking process. It relies on an access control decision function (ACDF) component, in order to minimize the transmitted informations to the access control monitor. Besides, an architecture and use-cases featuring XACML and SAML in distributed environments have been detailed in [86], validating the usability of XACML in distributed systems. However, it raises some limitations, such as the need for a high granularity of sub-policies and the difficulty of maintaining an encoded security policy. Moreover, those two languages address specifically the access control policy specification and enforcement. The languages and formats introduced by the SCAP framework constitute also an interesting support. They cover seven complementary format specifications that are exploitable for automating security management in distributed cloud [158]. The description of the platform requiring protection can rely on the *common platform enumeration* and *common configuration enumeration* formats. The scope of addressed vulnerabilities is tackled by the *common vulnerability enumeration* format and the *open vulnerability assessment language*. Their detection can be supported by the *extensible configuration checklist description format* and the *open checklist interactive language*. The impact of vulnerabilities can be evaluated based on a scoring, with the *common vulnerability scoring system*. The SCAP framework is closely tied to vulnerability management and system hardening.

Table 3.1 synthesizes and compares the different architectures and frameworks mentioned below. In particular, we have determined their compatibility with the multi-cloud and multi-tenant properties. We have also indicated the scope of protected cloud resources, as well the types of security features they enforce. Of course, the scope of resources restricts the enforcement perimeter for the security features. We have also assessed the support for the self-configuration of resource protections, and the level of technical coupling with respect to security policies. Although several existing work, such as the SDAC and PaaSword approaches, are close to address the different criteria of the table, their enforcements remain strongly focused on specific types of resource protection.

	Multi-tenancy	Multi-cloud	Protected resources	Security features	Self-configuration	Policy technical coupling
TCloud [18, 153]	✘	✘	Infra. platform., and service cloud assets	Resilience	✘	Low
Iceman [40]	✔	✔	Unspecified cloud resources	Access control	✘	Low
Runsewe [135]	✔	✔	Unspecified cloud resources	Access control	✘	High
SDAC [58]	✔	✔	Infrastr.-level cloud resources	Access control	✔	Low
Koorevar [75]	✘	✘	Networking resources	Network stream processing	✘	High
NICE [32]	✘	✘	Infra.-level cloud resources	Intrusion detection and vuln. manage-ment	✔	High
XACML [19]	✔	✔	Unspecified cloud resources	Access control	✘	Medium
Primelife [6]	✔	✔	Unspecified cloud resources, consumer identity	Access control	✘	Medium
PaaSword [152]	✔	✔	Data storage of cloud applications	Access control, encryption	✔	High
SCAP [158]	✘	✘	Infra.-level cloud resources	Vuln. mgmt, system hardening	✔	High

Table 3.1: Comparison of security architectures and frameworks for distributed clouds

3.3 Software-Defined Security Overview

Software-defined security (SDSec) can be seen as a strategy enabling the specification and the enforcement of security policies through a decoupling into two planes, similar to a sofware-defined networking approach. The policy-related decisions are performed in a security control plane, while being enforced by the programmability of a security resource plane.

3.3.1 Objectives

The SDSec approach is motivated by the four main objectives described below.

Consistency of security management. The SDSec approach promotes a consistent security management amongst resources that require protection. It aims at facilitating the alignment of configurations related to security mechanisms with respect to a security policy. The scope of this policy is not limited to a local technical context, but can address an arbitrary set of resources representing a whole service and the infrastructure supporting it. The decision-taking process defining resource security configurations is integrative, and takes into account the current state of resources in the enforcement perimeter.

Independence with respect to the resource technical context. The security enforcement of the SDSec approach is agnostic from the technical nature of the resources to be protected, and relies only on the exposed interfaces. This is particularly important in a multi-cloud and multi-tenant environment, where security mechanisms may be implemented differently.

Support for multiple security features. The SDSec approach is capable of supporting multiple security features (e.g. cryptography, access control, intrusion detection). To achieve this, each security feature has to provide its own logic to the security management, and the corresponding security mechanisms capable of enforcing the decisions.

Decreasing the operating expenses. From an exploitation perspective, the deployment of a SDSec approach aims at reducing the operating expenses with regard to the exploitation of protected services. It facilitates the deployment and the maintenance of the protection over cloud services.

3.3.2 Design principles

To meet these objectives, the SDSec paradigm is based on several design principles for a framework or a platform implementing it.

The SDSec approach imposes the decoupling of security management into two planes. The security control plane is in charge of interpreting the security policy and performing the decision taking process. It may rely typically on a security orchestrator. The security resource plane corresponds the resources requiring protection, and the security mechanisms in charge of protecting them. These mechanisms are security enforcers, that are usually not involved in the decision taking.

These two planes impact on the deployment and operation of security features. The logic behind a security feature is left to the security control plane, corresponding to orchestrator capabilities. Dedicated mechanisms in the security resource plane provide the necessary material

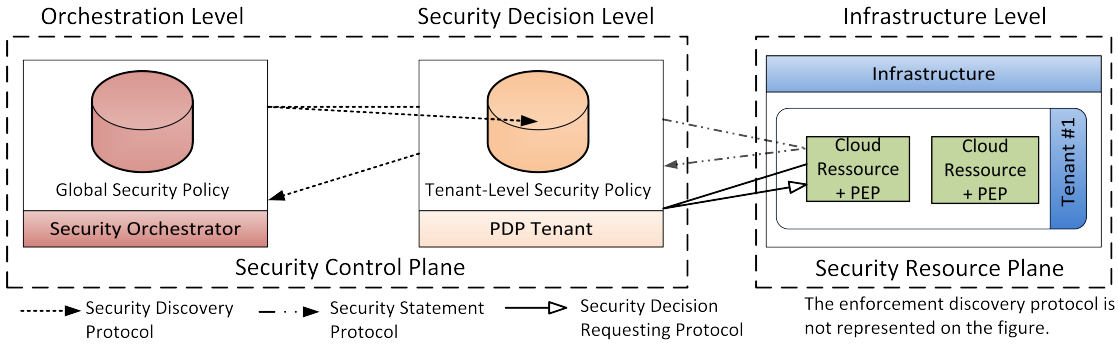


Figure 3.1: SDSec architecture in a single-infrastructure single-tenant environment

to enforce this logic. Therefore, each security feature can be abstracted as a capability for the security orchestrator and a set of security mechanisms.

The interactions between these two planes are restricted to the configuration interfaces of security mechanisms. These interfaces, enabling security programmability, are exploited to transmit the necessary information, so that the configuration of security mechanisms are aligned with the security policy.

3.4 Software-Defined Security Architecture

We propose a software-defined security (SDSec) architecture for protecting a distributed cloud. This architecture is consequently composed of two main planes, called respectively the security control plane and the security resource plane (as depicted on Figure 3.1). It exploits autonomic mechanisms within distributed cloud infrastructures, to enable cloud resources to be dynamically protected according to a given policy. More specifically, the security policy model includes a *global security policy (GSP)* which formally defines the security objectives of cloud resources. It is then translated into several *tenant-level security policies (TLSP)*, providing the security statements / rules that must be verified by specific resources at the tenant level, within the distributed cloud. These security statements are then enforced on cloud resources, i.e. virtualized infrastructures, virtual machines and software products. They aim at constraining the behavior of these components and protecting them based on counter-measures available within the distributed cloud. The enforcement should be performed dynamically, and meet adaptation and automation properties. Any changes on the protected resource state or in the infrastructure are taken into account in the enforcement. No human operator interventions are needed to maintain the enforcement. The policy decisions related to the enforcement are automatically performed according to contextual and security criteria.

The components of the architecture, part of the security control plane, include the *security orchestrator* and the policy decision points (PDP). The orchestrator hosts the global security policy (GSP), and exposes it through a dedicated interface to PDPs specific to tenants, in the form of tenant-level security policies (TLSPs). It receives enforcement feedbacks from the policy decision points (PDP). These interactions are supported by the security discovery protocol, enabling the PDP to identify the security orchestrator, and to fetch its security policy. The components, part of the security resource plane, correspond to the policy enforcement points (PEP). They correspond to programmable resources enforcing the security statements / rules, using the *security statement protocol*. This protocol permits a proactive security policy enforcement on

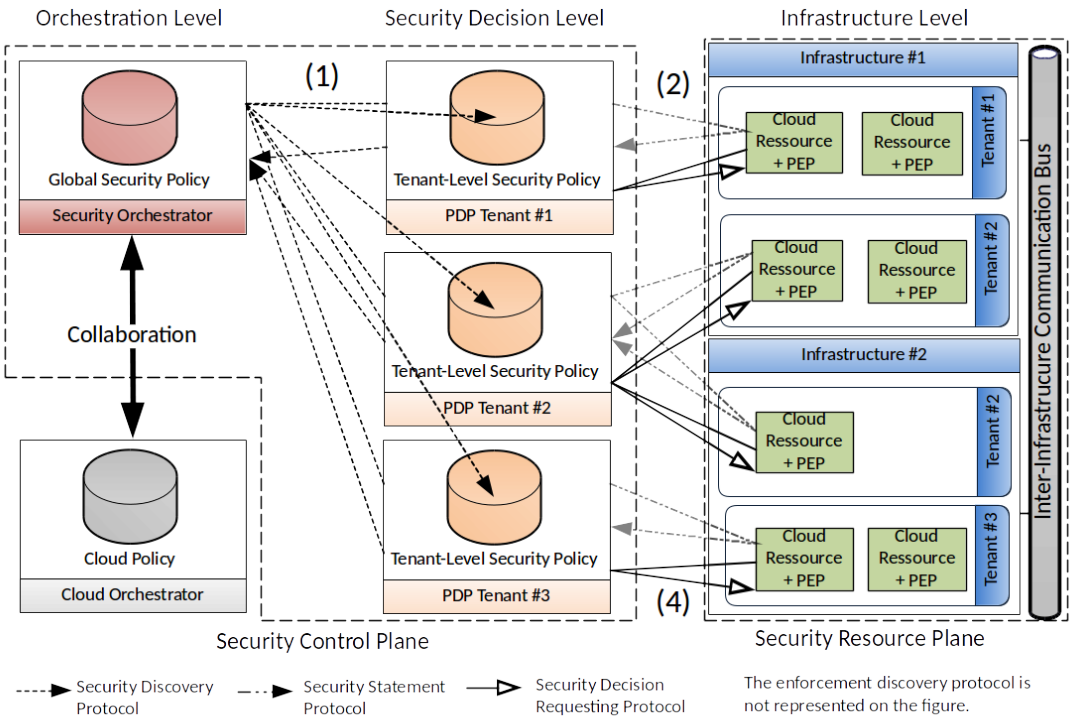


Figure 3.2: SDSec architecture in a multi-cloud and multi-tenant environment

specific categories of cloud resources. The PEPs may also solicit a PDP for taking a security decision, corresponding to a reactive enforcement, with the *security decision requesting protocol*. The automated configuration of security mechanims enable a lower coupling with respect to the orchestration. Contrarily to a regular orchestration addressing requests and expecting feedbacks, the security orchestrator adopts a more passive approach. It is supported by the PDPs, that are capable of taking specific decisions in accordance with their enforcement contexts. This architectural design, which is relatively generic, permits to cope with the multi-cloud and multi-tenant properties of distributed cloud. It addresses potential heterogeneous infrastructures that collaborate to provide a given cloud service, and the case of multi-tenancy that leads to define specific security requirements for each tenant of a given infrastructure.

We detail the roles and functioning of the components of this architecture on Figure 3.2. This figure makes the assumption that each PDP is dedicated to a single tenant, which is a simple interpretation of software-defined security in this multi-tenancy context. We consider the presence of a regular cloud orchestrator (CO) in charge of managing cloud resources. Even though this component is not meant to be a part of the proposed security architecture, its supposed presence allows taking into account the changes on cloud resources. This component may also corresponds to the manual intervention of system administrators, or to the automatic changes due to one or several orchestrator(s).

3.4.1 Security Orchestrator

Amongst the architectural components, the security orchestrator (SO) is responsible for the management of the GSP, its interpretation and distribution in the form of TLSPs. The GSP

policy is meant to be enforced on the distributed cloud, and so, on multiple collaborating cloud infrastructures with different tenants. The interpretation is influenced by feedbacks provided by the enforcement. In line with the XACML terminology [19], the security orchestrator is similar to a Policy Administration Point (PAP), allowing the storage of the GSP policy, and its translation into TLSPs. The changes operated on the GSP policy have to be propagated to the whole enforcement perimeter. Contrarily to the cloud orchestrator, the security orchestrator is not meant to manage cloud resources. Consequently, the instantiation, the removal or the reconfiguration of cloud resources is not endorsed by the security orchestrator.

However, this highlights the need for the security orchestrator and the cloud orchestrator to collaborate. For instance, the security orchestrator requires to be noticed in case of deployments of new cloud resources, in order to enforce the GSP policy on them. In the same manner, the cloud orchestrator have to remove a resource and reconfigure its workflow, when the security orchestrator requests its removal for security purpose. This collaboration is represented on the left part of Figure 3.2, by the double arrow between the two orchestrators. An overview of the activity diagram of the security orchestrator is given on Figure 3.3. The orchestrator does not push the TLSPs to the PDPs. This design choice sustains the isolation of tenants among each others and with the cloud administrator, as required to fulfill the multi-tenancy property. These TLSPs must be attached to meta-data to enable PDPs to fetch only the policies they are concerned with, by discriminating each TLSP according to enforcement context criteria. The policy has to be exposed through a dedicated interface accepting incoming connections from PDPs (with the use of the security discovery protocol). Another interface assumes the reception of all PDP enforcement feedbacks. The determination of the exposed TLSPs (as well as the notification sent to the cloud orchestrator) is correlated to the GSP policy, the PDP feedbacks and the notifications that may be sent by the cloud orchestrator.

3.4.2 Policy Decision Points

The Policy Decision Points (PDPs) play a central role in this software-defined security architecture. They serve as intermediates between the security orchestrator and the PEPs enforcing policies on cloud resources. More precisely, the PDPs are in charge of fetching and hosting the TLSPs using the policy security discovery protocol, and locating their PEPs by invoking the enforcement discovery protocol. They support the interactions with PEPs by collecting their feedbacks and responding to security requests in accordance with the hosted TLSPs. In our SDSec solution, they enact the decision taking process of the security control plane. Delegating this activity to a tenant-specific component prevent any interference of other tenants on the enforcement of security features. This is particularly critical, when enforced security features are directly related to the confidentiality of the cloud resources in the tenant, such as cryptography or access control. According to the XACML architecture [19], the PDPs assume different roles: the role of PDPs providing authorization decisions, but also the role of policy administration points (PAPs) with respect to TLSPs. Figure 3.4 draws a comparison between the XACML architecture [19] and the PDP. All the components from the XACML architecture fit in our SDSec architecture, showing its ability to handle the access control security feature. However, it diverges on the support of a pro-active enforcement over resources requiring protection. PDPs have to take into account external informations, during the interpretation of their TLSPs. For instance, a time-regulated access control policy requires an access point to a system clock. As this parameter cannot be generalized to all PDPs of the enforced perimeter, the PDP has to propose an extensible interface able to communicate with third-party security information providers. These providers are part of the security control plane. With the XACML architecture, these

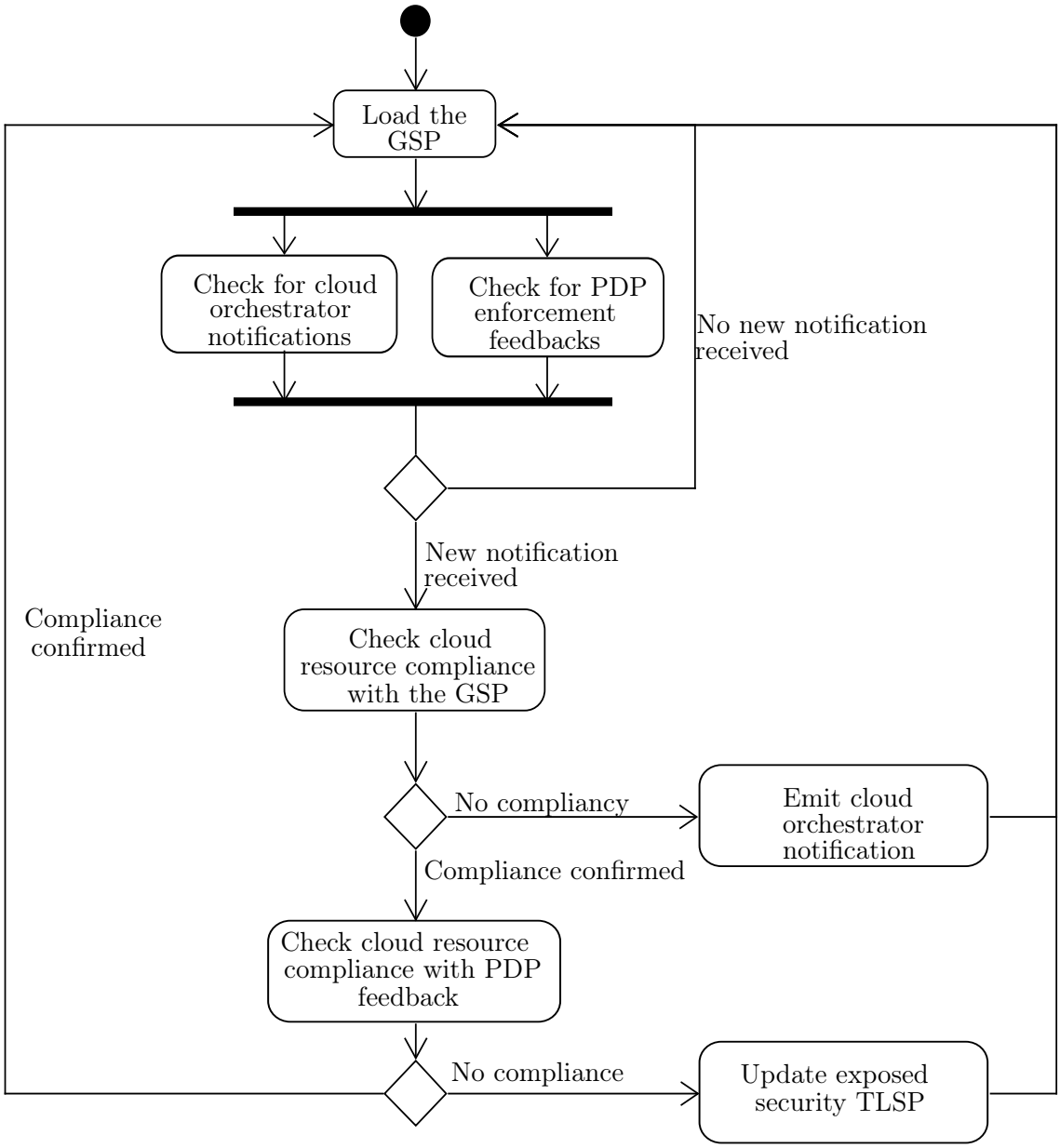


Figure 3.3: Activity diagram of the security orchestrator

third-party resources are assimilated to Policy Information Points (PIPs). Besides, the PDPs maintain several meta-datas describing their decisional capabilities, which are directly related to their enforcement context. These meta-data are important for the security policy discovery protocol, as they enables the PDP to determine the TLSP policy adequate to its enforcement perimeter. Consequently, the security statements intended to the PEPs are directly related to the stored TLSPs, modulated by the feedbacks given by the PEPs, and eventually, by the PIP contents. Figure 3.5 gives a summary of the behavior of a PDP.

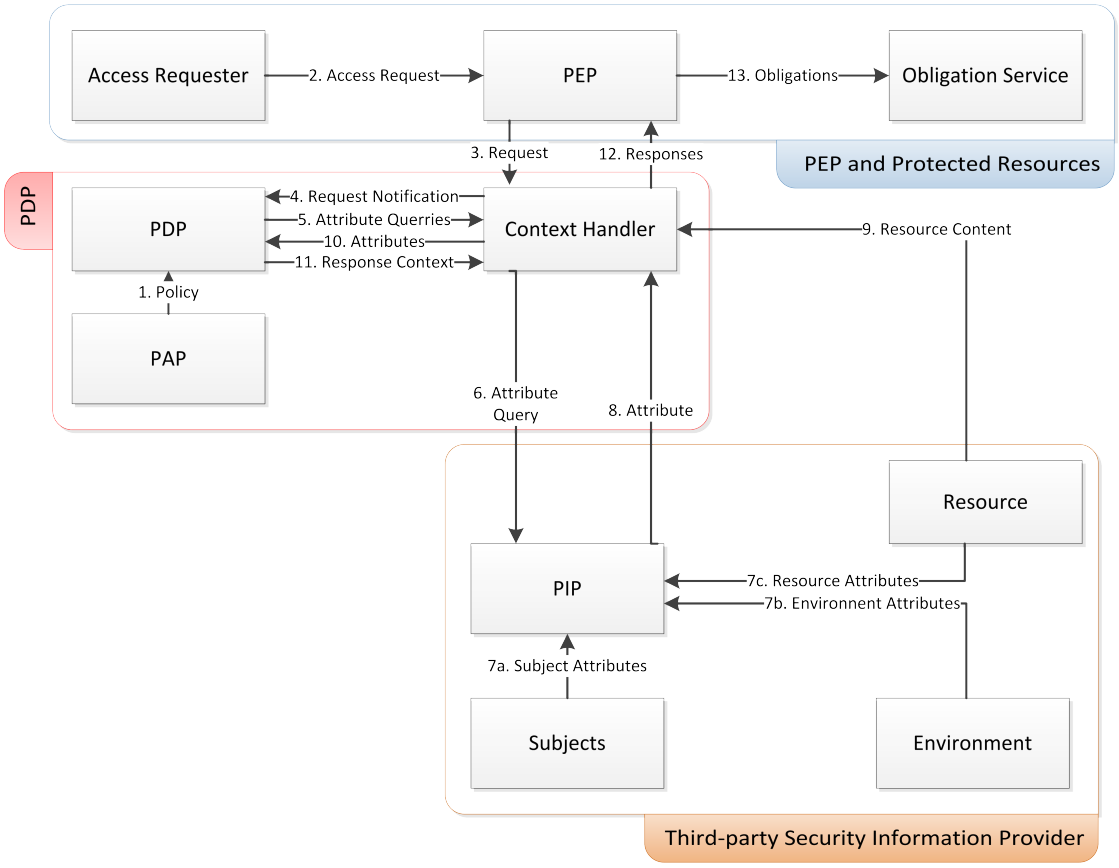


Figure 3.4: Comparison of our SDSec architecture with the XACML architecture

3.4.3 Policy Enforcement Points

The Policy Enforcement Points (PEPs) are in charge of the enforcement of TLSPs for a dedicated cloud resource. More precisely, a cloud resource refers to an instantiated resource on a cloud infrastructure. Virtual machines, cloud services, sets of files or network functions are examples of such resources. The considered enforcement is twofold. In one hand, the control and modification of security parameters on the resources according to the security statements / rules enable a pro-active enforcement. On the other hand, the insertion of security event hooks to handle state changes and prepare associated security decisional requests paves the way for a reactive enforcement. Examples of such security event hooks include incoming connection attempts, configuration modifications, and the exceeding of a threshold. Besides, the objectives of the PEPs in the architecture are not limited to their counterparts in the XACML architecture. Not only the PEPs are able to trigger the PDPs for reactive enforcement, but the PDPs can also start interacting with the PEPs on their own for a proactive enforcement.

Consequently, the PEPs have to expose an interface to the PDP for receiving security statements, and be able to contact the PDPs to return feedbacks (after the execution of a security statement or after an event hook), and to transmit a security decisional request. The configuration of security parameters is directly dependent on received security statements. The feedbacks are defined based on received security statements, states of considered security parameters and event hook states. Security decisional requests are emitted by the PEPs based on event hook states.

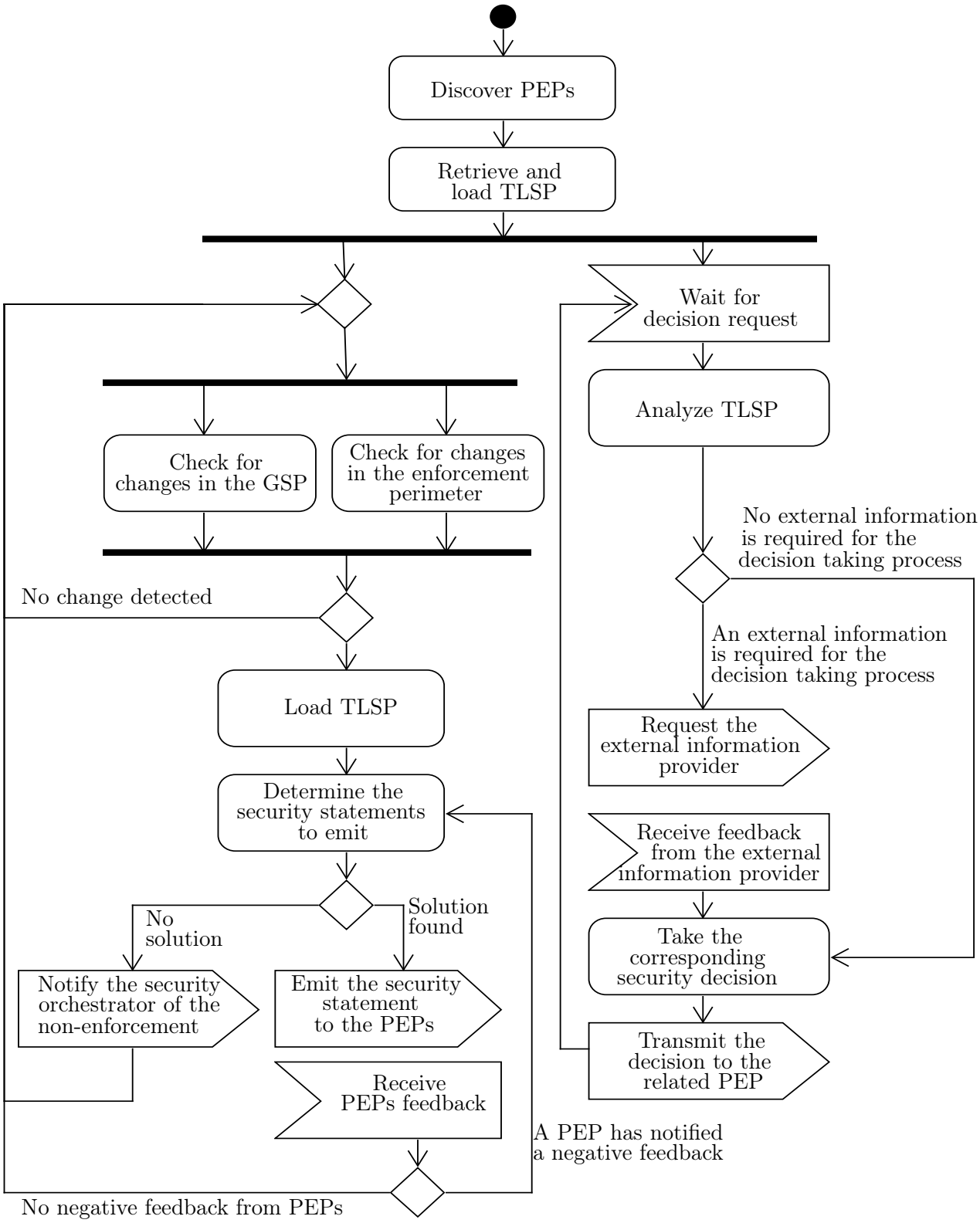


Figure 3.5: Activity diagram of the PDP

3.4.4 Interactions Amongst Components

The interactions amongst the components of this software-defined security architectures is supported by different protocols, that are synthesized below:

- The *Security Policy Discovery Protocol* is a discovery protocol invoked by a PDP to discover the security orchestrator and fetch a TLSP policy. The discovery process takes as inputs the PDP meta-data, and gives back the required TLSP policies. Because of the criticality of this protocol, its specification has to integrate technical mechanisms to protect the integrity of informations and have to be tamper-proof.
- The *Enforcement Discovery Protocol* enables a PDP to discover available PEPs in its enforcement perimeter. It therefore permits the PDP to determine its enforcement capabilities. More precisely, these capabilities are expressed by available PEPs through their enforcement meta-data, and brought back to the PDP, which determines their potential contributions to the security enforcement. To prevent security policy information leaks to an intruder, or prevent an intruder to weaken the security enforcement by providing false security assessment feedbacks, the protocol has to enable the PDP to verify the authenticity of the discovered PEPs.
- The *Security Statement Protocol* supports the security programmability in this architecture. It enables the PDPs to generate security statements, and send them to the PEPs in their enforcement perimeters. The feedback has to be emitted asynchronously, in case of enforcement execution time-outs.
- Finally, the *Security Decision Requesting Protocol* offers to the architecture its dynamic enforcement properties. Indeed, this protocol enables the PEPs to solicit the PDPs for handling a security decision. This security decision request occurs when a security hook of a PEP is triggered, and the security statement cannot be handled by the PEP itself.

3.5 Architecture Evaluation

In order to analyze and validate our proposed architecture, we have considered a set of scenarios based on a realistic use case, corresponding to a cloud service provider (CSP) proposing a *Platform-as-a-Service (PaaS)* solution to customers, based on world-wide infrastructures. These scenarios are inspired from operational use-cases from Orange as cloud service provider. They are bound to tasks that are manually performed by operators, but are expected to be automated. This analysis permits to evaluate the applicability of the architecture to real-world issues in a multi-tenant and multi-cloud context. From a practical viewpoint, the multi-tenancy typically corresponds to the use of the same infrastructure by several independent customers, while the multi-cloud property may come from the world-wide location of cloud infrastructures. We consider the use case depicted on Figure 3.6. To protect its solution, a cloud service provider enforces a security policy on its own infrastructure, and on its cloud resources instantiated by tenants. In that context, we consider the case of a customer, deploying two virtual machines (VM) for hosting web applications. The first VM corresponds to the European version of his application, while the second VM corresponds to the American version.

3.5.1 Validation Scenarios

The scenarios inferred from this use case, make the following assumptions:

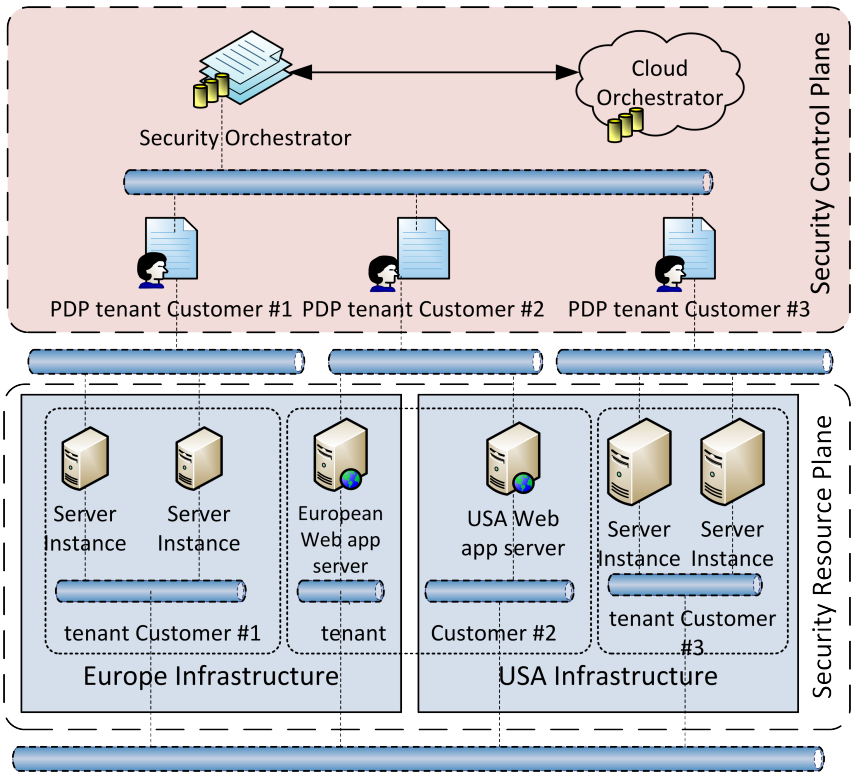


Figure 3.6: Use-case supporting the validation of our SDSec architecture

1. The CSP has implemented the business/operational processes using a cloud orchestrator.
2. Each customer request is endorsed by the cloud orchestrator.
3. The customers are unable to remove the PEPs of the cloud resources.
4. The communications amongst the PEPs and PDPs are not interrupted.
5. The deployment of software stacks in the PaaS resources is governed by the cloud orchestrator and embed the related PEPs.
6. The cloud resource manager comes with its own PEP, which is managed by the tenant PDP.

We have analyzed a set of five scenarios, corresponding to the deployment of a new virtual machine instance for a customer, the update of the security policy by the cloud service provider, an attack (DDoS) targeting an instantiated virtual machine, an inter-resource access request, and the removal of a virtual machine instance. These scenarios permit to address the whole life-cycle of a resource in a cloud service, while covering both proactive and reactive security enforcement.

Resource instantiation scenario

In the first scenario, depicted on Figure 3.7, the customer sets up a dedicated server associated to his tenant, in order to synchronize and back up the informations of the instances of his web application. The virtual machines hosting its web applications are Linux-powered, embeds a

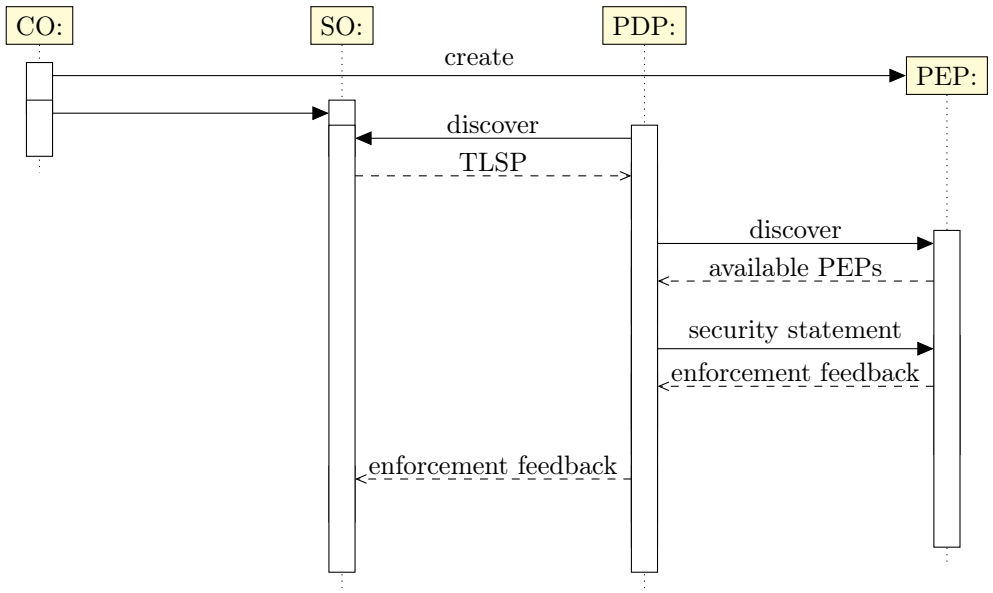


Figure 3.7: Sequence diagram of the instantiation scenario

SSH server for administrative tasks and a web server. The chosen technical solution consists in using a SQL server and a SFTP server in a dedicated VM stored in the European infrastructure, which will accept connections from the two web application servers. The cloud orchestrator (CO) processes the deployment of these two services with their respective PEPs and notifies the security orchestrator (SO). As SFTP and SQL are newly deployed services in the tenant, the security orchestrator assumes that the TLSP of the tenant PDP is not adapted anymore, and modifies the exposed TLSP to this PDP. The PDP discovers the two new PEPs, fetches the newly available TLSPs from the security orchestrator, and sends the security statements to the PEPs. Finally, the PEP transmits a positive enforcement feedback to the PDP, and the PDP acts likewise with the security orchestrator. This prevents the security orchestrator from requesting the cloud orchestrator to take counter-measures against the tenant resources.

Security policy update scenario

In the second scenario shown on Figure 3.8, the security administrator of the CSP updates the security of its infrastructure, by restricting the access of critical services only to the local network and the CSP VPN. The criticality of a service is not defined in the GSP policy, but is delegated to the PDP. After the update of the GSP policy and its processing by the security orchestrator, the PDP of each tenant detects and collects updated TLSPs. All the PDPs interpret their TLSPs into security statements restricting the critical service access.The PDP associated to the consider customer has deduced that all SSH and SQL servers were critical. It requests their PEPs to restrict their access and notifies the security orchestrator of the effective enforcement. If one of the PDPs receives a PEP negative feedback and has no other counter-measure to apply, it notifies the security orchestrator which will in turn notify the cloud orchestrator to disable vulnerable services.

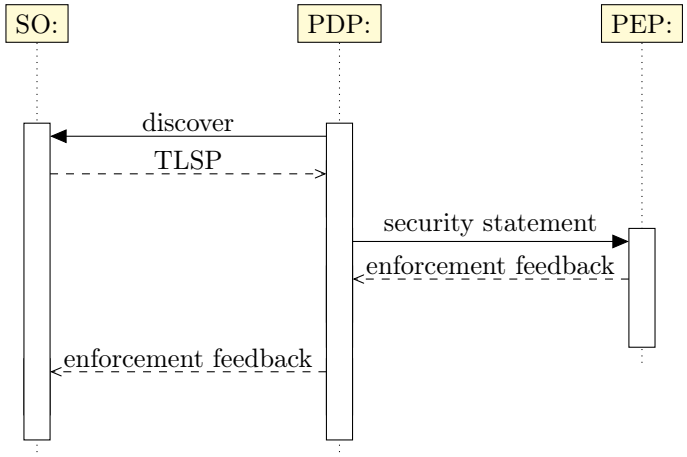


Figure 3.8: Sequence diagram of the security policy update scenario

Attack scenario

The attack scenario is depicted on Figure 3.9. The VM in charge of the European version of the web application hosting is exposed to a Distributed Denial of Service (DDoS) attack targeting the web server application. This VM is protected by a network appliance capable of IP address filtering, but still has access to the effective source address of the HTTP requests. This mechanism supports live configuration of the IP addresses whose connections have to be blocked. It can interface with the VM webserver to retrieve the most encountered IP addresses from the client. In-VM PEP detects the incoming DDoS attack through the increasing memory and CPU consumptions from the webserver process, and notify the PDP when they exceed a threshold specified in the TLSP policy. The PDP requests the IP filtering mechanism to block the 200 most active IP addresses that are interacting with the web server. Once proceeded, the mechanism replies with a positive enforcement feedback. Once the DDoS attack is countered, the webserver resource consumption reverts back to a normal state, and the VM PEP notifies about the effective TLSP enforcement with a positive enforcement feedback.

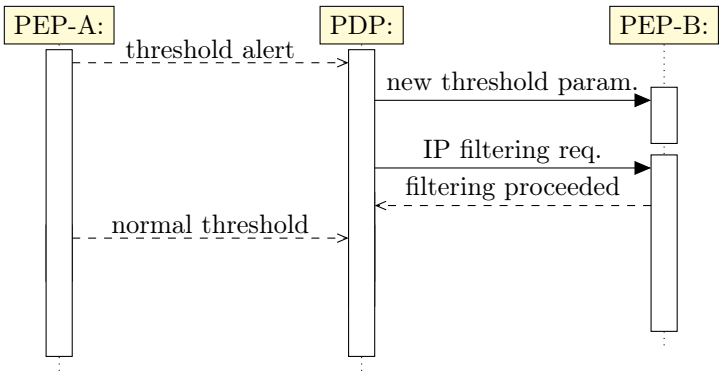


Figure 3.9: Sequence diagrams of the attack scenario

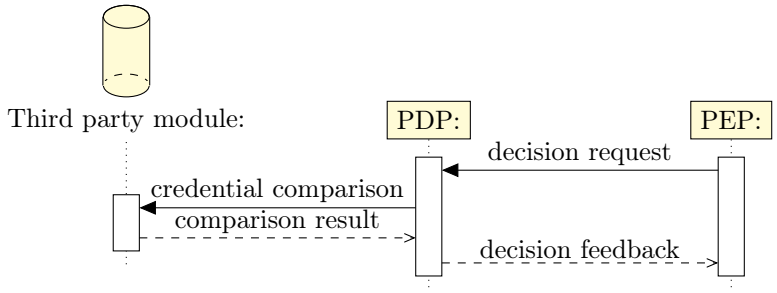


Figure 3.10: Sequence diagram for the access request scenario

Access request scenario

The next scenario corresponds to an access request depicted on Figure 3.10. The cloud service provider has defined in its GSP policy, that the used credentials for the connections among cloud resources have a limited lifetime, and have to be regularly changed. The verification of the validity is enforced by the PDP using a third-party module. Meanwhile, the client has set-up an automatic back-up process between the backup server hosted in the European infrastructure and the production server located in the USA, by using SQL and SFTP transactions. The production server authenticates to the backup server using a dedicated password. When the production server connects to the back-up server, the connection attempts trigger the connection hooks of PEPs related to the SQL and SFTP servers. Both of them block temporarily the connection attempts, and make decision requests to the PDP, providing hashes of used credentials. As the TLSP policy imposes the verification of the credential life-time, it uses its third party module to check it. As this module has no precedent records of hashes, it concludes that the transmitted credentials are newly created and are allowed to be used. The PDP responses to both security decision requests are positive, and incoming connections are authorized by respective PEPs.

Resource removal scenario

The last scenario, shown on Figure 3.11, corresponds to a resource removal. The client wants to update the virtual machine supporting the American web application by proceeding to a fresh installation. To meet this objective, the client wants to completely remove it and reconfigure a new virtual machine. He uses the cloud orchestrator to remove this virtual machine, which is notified to the security orchestrator. The security orchestrator updates its GSP, to take into account the removal of the cloud resource and checks its consequences on the enforcement: The TLSP is updated. The PDP of the customer fetches the new TLSP policy, and stores it. Through the cloud orchestrator, the security orchestrator starts deallocating resources to the American VM and the PEP addresses a security decisional request to its PDP for allowing the removal. According to its TLSP, the PDP grants the request. The PEP lets the cloud orchestrator to complete the resource removal.

This analysis shows that all the presented scenarios can be addressed by our proposed software-defined security architecture. However, some limitations with respect to the considered use case should be highlighted. First, the use case has dealt with a GSP policy set by one security orchestrator. The case of multiple security administrators, with different enforcement parameters, is also interesting to be investigated, while we can abstract it through the single security orchestrator case. Second, the use case assumes that one PDP is allocated to one tenant, corresponding to one customer. This is however only one possible interpretation of the

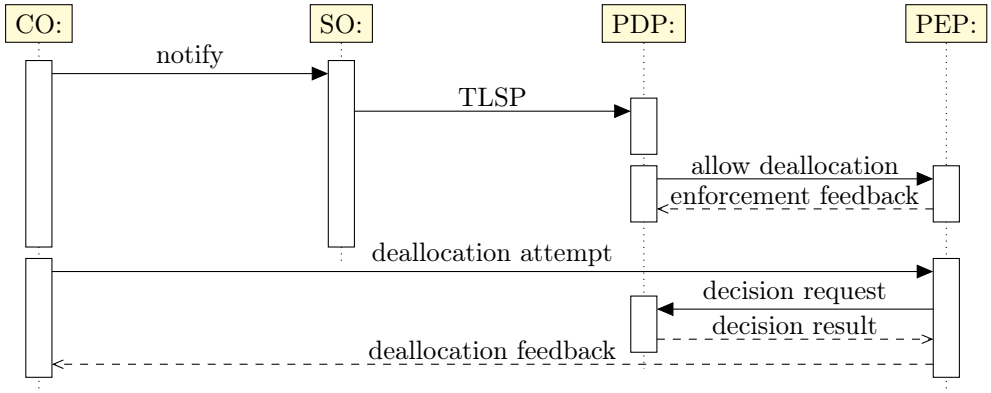


Figure 3.11: Sequence diagram for the resource removal scenario

multi-tenancy notion, but other ones would have made the use case unnecessarily more complex.

3.5.2 Practical Considerations

After analyzing different validation scenarios, we are discussing here practical considerations with respect to this software-defined architecture.

Cloud environment

Before considering a software-defined security stack for our architecture, we focus on the environment and the resources we want to enforce. We address distributed cloud infrastructure security. The retained technical solution should be a proven solution in the multi-tenancy area as well as the multi-cloud one. Moreover, as suggested by the network filtering appliance in the third validation scenario, some of the counter-measures are likely to rely on infrastructure configuration. This highlights the need for an extensible cloud stack. In both cases, the OpenStack cloud suite is an attractive solution, as it supports multi-tenancy through the users and region management, and it supports the integration of additional components.

Considering the orchestration, we have to distinguish the need of a security orchestrator based on a security policy ruling, and a regular cloud one whose actions are driven by customer solicitation or CSP management tasks. The security orchestrator will be further analyzed in the next subsection. The cloud orchestrator has no specific security expectation except its capability to handle cloud orchestration notifications, and reciprocally emits notification to it. These two requirements are related to common orchestrator features as both are linkable to basic messaging between cloud appliances, each one issuing a request to the other and waiting for a feedback. Therefore, no more prerequisite other than distributed cloud support is expected from them.

In the cloud resource area, our architecture is designed to be resource agnostic in the sense that the PEPs are the only agents of the architecture depending on cloud resources. Their interactions are based on resources programmability, inspection and event handling. These features gains interest as they are related to dynamic and complex resources. In this context, virtual machines operating systems and applications are well-suited for exploring this kind of enforcement, but cannot be generalized as the only type of resources to be protected. Besides, their nature directly influences the way PEPs are implemented: an executable cloud resource opens the debate about whether the PEP should be totally, partially or not at all included in it while a non-executable one excludes it.

Architectural components

Considerations are also raised by the implementation of the framework itself. The security orchestration permits the coordination of the PDPs with each others and the cloud infrastructure (through the cloud orchestrator). As such, it is a highly critical single point of failure in charge of supervising several tenants and infrastructures. Such a criticality raises technical issues about redundancy or distribution among the infrastructures, but also policy concerns such as handling enforcement state transition due to GSP modification: if the modification process is not properly handled, as cloud tenant-level security policy and cloud-resource statement are not instantly propagated (due to network or processing overhead), we can conceive that a subset of resources of the cloud infrastructure managed by the security orchestrator can be trapped into a inconsistent security state. This eventuality must urge the orchestrator to check the consistency of intermediate enforcement states, at the infrastructure level (resource enforcement state can conflicts) and at the policy-decision level (concurrent low-level security policy can as well conflicts).

Moreover, the privacy concerns is risen with the PDP. Indeed, it can access all the PEPs it is in charge of, and any data leak may allow an attacker to collect resource data or metadata. Incidentally, the confidentiality of the communications between PEPs and PDPs is as critical as the isolation between PDPs is. This statement decides the question of the relationship amongst PDPs and tenants. To enforce a proper isolation between PDPs, it is necessary that each of them address one unique tenant. Otherwise, one tenant could compromise a multi-tenant PDP, and use it to fetch data from the other tenant resources.

Finally, the variability of the resources addressed by this security architecture leads to the question of PEP design. Building one PEP for each type of resource to enforce a TLSP policy in a cloud is not a sustainable approach as the workload for a sufficient enforcement coverage would go too far. Thus, we should consider a more generic approach allowing an automatic adaptation to cloud resource. An adaptive design and instantiation of PEP is a interesting response element, as the core logic of the PEP could be specified, before being compiled and adapted on-the-fly to the particularities of the resource to protect. Moreover, such an approach could possibly take advantage of the cloud resource build environment: if this PEP design and integration process is able to extract the required information from cloud resources being constructed, it would lead to an automatic and adaptive design of PEPs tied to cloud resource dynamics.

3.6 Summary

We have proposed in this chapter a software-defined security architecture for protecting distributed clouds. It relies on the programmability introduced by software-defined security, and addresses the constraints induced by multi-tenancy and multi-cloud properties. We have detailed the different components of this architecture, including a security orchestrator, policy decision points (PDPs) and policy enforcement points (PEPs), interacting according to a dedicated set of protocols. Based on the specification of a security policy, the architecture supports the dynamic configuration of security mechanisms to adjust to contextual changes, based on available resources and counter-measures. It enables a low coupling with respect to the orchestration, through the use of PDPs. We have evaluated the proposed solution and discussed practical considerations, through a set of validation scenarios inspired from operational use-cases from Orange as cloud service provider. The proposed approach has raised several challenges with respect to the design of the considered components, and the specification of security policies in a multi-cloud and multi-tenant context.

This software-defined architecture is only the first building block of our solution. In the next

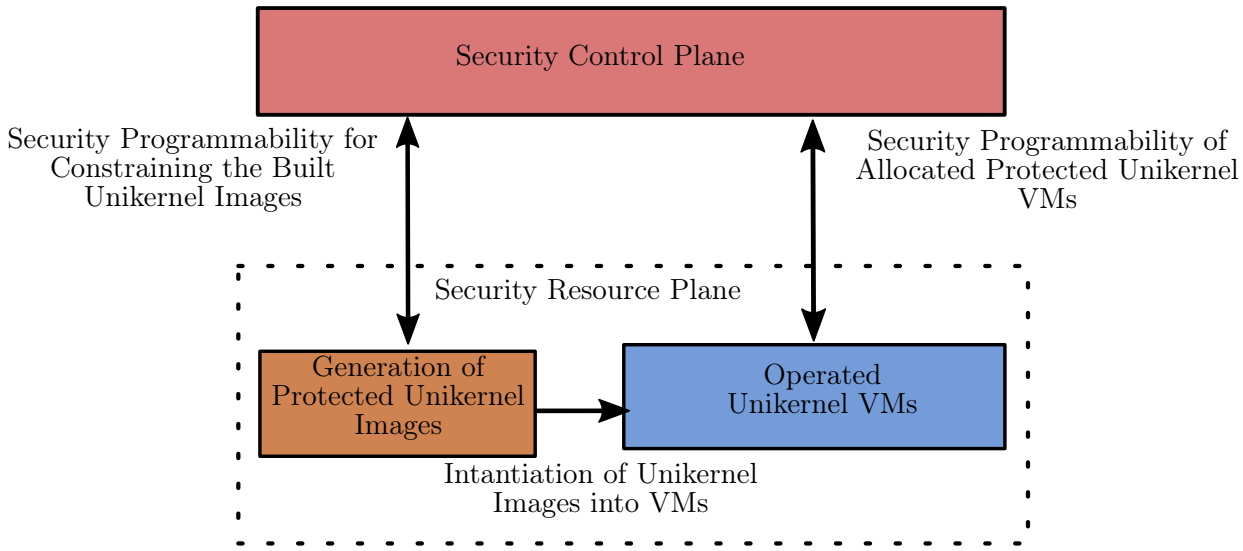


Figure 3.12: Integration of the SDSec architecture with our unikernel generation framework

chapter, we will focus on the second building block, corresponding to the on-the-fly generation of protected unikernels. The objective is to leverage security programmability in cloud environment through the generation of these unikernels, whose benefits have been highlighted in the state-of-the-art. Figure 3.12 overviews the integration of our SDSec architecture with the unikernel generation framework.

Chapter 4

On-the-Fly Protected Unikernel Generation

Contents

4.1 Introduction

In the previous chapter, we have shown that applying the software-defined paradigm offers new perspectives to tackle the issue of security management complexity in distributed clouds. Software-defined security (SDSec) permits to decouple security policy enforcement from their specification by abstracting resources technical considerations. The security control plane is in charge of security decisions through business-logic constraints, while the security resource plane represents the resources to be protected and dedicated programmable security mechanisms accounting for technical constraints. The provided security should take into account the changes affecting the resources and their context, in order to dynamically adjust the enforcement and maintain the compliance to security requirements. The adequacy of these mechanisms with allocated resources directly impacts the coverage and efficiency of security management over the infrastructures to be protected. Unfortunately, the heterogeneity of addressed resources requires more and more specialized mechanisms, increasing the management costs and reducing the exhaustiveness of protection. Performance improvement efforts have led to several initiatives in

the area of system virtualization, in order to reduce the footprint of appliance execution environments. The thriving adoption of containerization has successfully contributed to this objective, as well as refining the granularity of cloud scalability, improving their portability and promoting new appliance lifecycles (e.g. DevOps [66]). However, containerized applications have to eschew OS kernel modifications, banning several applications that have still to comply with complex execution driven by legacy support constraints.

We present in that context a software-defined security strategy exploiting unikernels for protecting cloud infrastructures. These unikernels correspond to lightweight virtual machines specially built for a dedicated application at the expense of backward compatibility, while their performance enables a short lifespan-based usage [91]. They promote a simplified and analyzable internal functioning, putting on an equal footing an application, its requirements and the OS routines they rely on. We exploit their properties to reduce the attack exposure of cloud resources, through the on-the-fly generation of highly constrained configurations with the strict necessary for a given time-limited period.

Our main contributions, described in this chapter, are (i) specifying a software-defined strategy based on unikernels to protect cloud infrastructures, (ii) designing a management framework with a dedicated modeling to support the generation of unikernels with security mechanisms, (iii) implementing a proof-of-concept prototype using MirageOS, and (iv) providing a performance evaluation based on extensive series of experiments. This chapter contributions tackle the compatibility issue between the security management plane and the resources requiring protection. Therefore, the evaluation work addresses rather the impact of this compatibility over protected resource performances than the leveraged security management. The evaluation of the framework is complemented by experimental results presented in Chapter 6.

The remainder of this chapter is structured as follows: we present related work in Section 4.2 and provide a background on unikernels in Section 4.3. Section 4.4 describes our software-defined security strategy based on unikernels, with the formalization of unikernel image generation over time. We evaluate in Section 4.5 the performances in comparison to legacy virtualization, through an implementation prototype. We conclude the contribution of this chapter in Section 4.6.

4.2 Related Work

We have already presented work related to virtualization techniques in the state-of-the-art chapter. System virtualization can be exploited for enabling third party security mechanisms to inspect in-VM applications and communications. For instance, LARES [122] is an architecture enforcing an active monitoring of a VM through the injection of dedicated mechanisms in-situ. On the contrary, VMWatcher [64] addresses a semantic view reconstruction from VM memory, to enable a non-intrusive monitoring. In the same manner, Livewire [47] is an programmable IDS inspecting VM states. Slick [10] focuses on VM storage I/O to detect intrusion attempts. Specific architectures also contribute to reducing the attack surface. Microkernels [81] have sustained an OS compartmentalization approach by dispatching non critical OS features across isolated services residing in their own memory space, but at the cost of a serious overhead. Exokernels [42] correspond to a library OS supporting multiple process executions, but promotes routine injections to prevent inconsistent behavior in hardware resource management due to process switch. Considering system virtualization, Drawbridge [126] is another library OS embedding instances of system components and applications in a VM, but collaborating with a host OS to access hardware resource securely and provide a seamless user interface. UKVM [160] is a monitor dedicated to the VM it runs to embed only required features to its execution. [23] sustains the usage

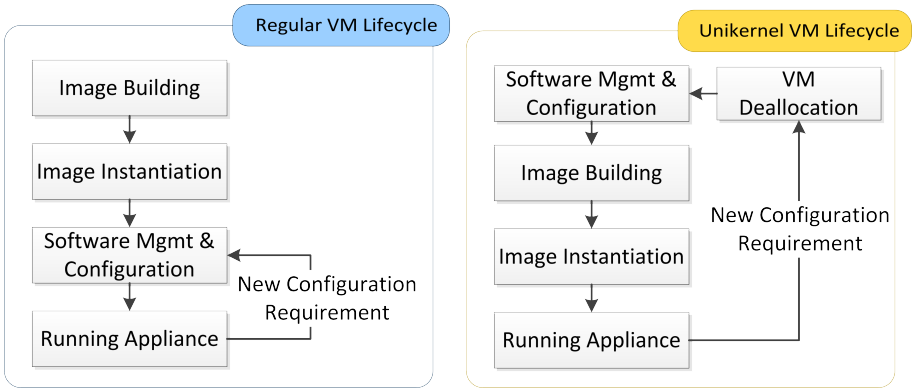


Figure 4.1: Comparison of regular and unikernel VM lifecycles

of unikernels in cloud environment from a security perspective. Our purpose is to take benefits of unikernels and their properties to support security programmability in VM cloud resources.

4.3 Background on Unikernels

We remind some important background related to unikernels and their building. Virtualization enables hardware to be abstracted from programs by a software layer. Hypervisors are in charge of provisioning dedicated virtual hardware environments to each program and interfacing with hardware. Amongst virtualization models, unikernels offer new perspectives for minimizing the attack surface.

Overview. The complexity of system architectures in VMs, relying on a full-featured OS kernel and application runtime, gives however a glimpse of an optimization through the simplification of these dependencies. Unikernel VMs simplify the in-VM system architecture by reducing legacy features from the OS (e.g. multiple users support, multiple hardware resource support), restricting each VM to one application, and embracing the library OS, to enable an efficient virtual hardware resource management. The library OS implements the hardware resource management to a set of regular libraries that are statically linked to the applications. In return, this restricts applications to cope with the hardware resources whose support is implemented by linked libraries.

A unikernel system image embeds a single application with its dependencies. Those encompass a minimal runtime together with software and hardware resources management libraries, in accordance with the library OS concept. The software management of unikernel is performed externally through provided building tool-stack, directly on system image before instantiation. This approach alleviates unikernel from the in-situ software management constraint, contributing to their lightness. Management can still rely on package management mechanisms (e.g. OPAM [113] for Mirage) or solely on source code inclusion mechanisms (e.g. IncludeOS). This constrains refreshes on a VM configuration process performed before the VM allocation. Its lifecycle is depicted on Figure 4.1. From our security perspective, unikernels offer interesting properties to reduce the attack surface, by generating highly constrained configurations limited to the strict necessary in a dynamic manner.

A unikernel instance uses a single address space for its application and its runtime. It does not implement processes, but rather uses threads, outwitting the context switch overhead during

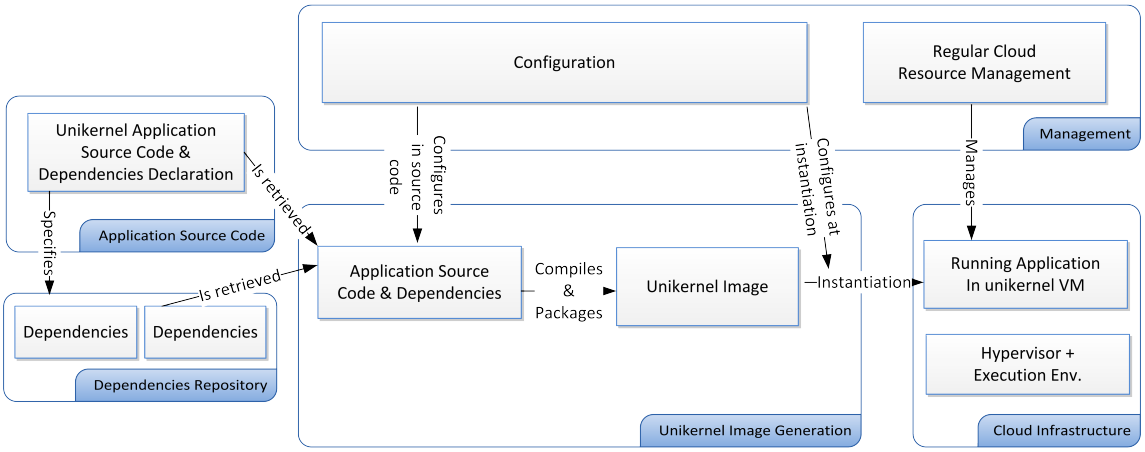


Figure 4.2: Unikernel generation architecture

concurrent processing. The light footprint of unikernel VMs refreshes on their fast boot time, and contributes to operation of short-living unikernel instance exploitation. Naturally, unikernel images can be designed to cope with an hypervisor, be launched in a VM and interact with a virtual hardware environment.

Building architecture. Figure 4.2 presents an architecture supporting the unikernel image generation and allocation in a cloud infrastructure. We can distinguish different steps.

1. The building environment is provisioned with the source code of the unikernel application to be build. This source code is expected to specify the core logic of the application and its dependencies through the libraries to be statically linked. Its design is constrained by the unikernel platform: usable libraries are partly or totally provided while the unikernel platform while employed programming language are framed by it.

2. Source code dependencies are managed through packages gathering libraries by a package manager. This one handles their retrieval from remote repositories and the management of their versions installed locally. This software can be dedicated to the unikernel platform or be bound to the programming language related to the unikernel platform.

3. Once all the dependencies are fetched, a toolchain is able to compile the source code and package it into a unikernel image. The tooling is partly or totally provided by the unikernel platform. The remainder is provided by the programming environment it is related to. The produced image is self-sufficient, as it does not rely on any external dependencies to be bootstrapped, except for the hypervisor they are designed to cope with.

4. This unikernel image can therefore be allocated in a cloud infrastructure, as one or several unikernel VMs. Their execution is supported by the hypervisor and its execution environment. It exposes a virtual hardware environment compatible with the hardware resource support of the unikernel. Those instances are driven by a regular cloud resource management. The concision requirement over unikernel VMs imposes the management to be external.

5. The process image building and allocation can incorporate a configuration at two levels. First, the configuration can be incorporated statically in unikernel images by altering its

source code to constrain its execution. Passing configuration parameter through variable definition or configuration file are examples of such configuration method. Second, the configuration can be set dynamically to a unikernel instance by passing boot arguments during unikernel VM allocation process.

Table 4.1 gives a synthetic comparison of regular executables supporting applications with unikernels.

	Regular executable	Unikernel
Source code	Regular application source code	Regular application source code
Dependencies	Libraries to be linked statically and dynamically	Libraries to be linked statically
Generated output	Regular executable file	VM image
Instance	OS process	VM instance
Required runtime	OS providing dynamic libraries	Hypervisor
Configurability	• During build process with source code modification. • At startup through command-line arguments and configuration files.	• During build process with source code modification. • At startup through VM kernel boot arguments.

Table 4.1: Comparison between a regular executable and a unikernel

This building architecture serves as a basis for our framework generating secured unikernels.

4.4 Software-defined Security Framework Based on Unikernels

We propose a software-defined security framework that exploits unikernel properties for protecting cloud infrastructures. This framework permits to generate on-the-fly unikernel images that integrate protection mechanisms. Unikernels have a less flexible configuration in comparison to regular virtual machines, but allow for significant reduction of the attack surface. In that context, we exploit unikernels to build highly-constrained configurations limited to the strict necessary and with a time-limited validity. Configuration changes are supported by the generation of new unikernel images in a dynamic manner.

As depicted on Figure 4.3, the framework components are layered on the unikernel regular building chain. The source code of a unikernel defines the main behavior of the considered appliance and its libraries dependencies. Those are stored in a dedicated repository. Security mechanism requirements are interpreted from source code dependencies declaration, and those are stored on a dedicated repository. In the unikernel image generation, every unikernel application has its dependencies and required security mechanisms fetched and its source code compiled and assembled in a bootable VM image. This VM can be instantiated in a cloud infrastructure, on the top of an system virtualization environment. The management of the building process and the VM launch are performed by a dedicated plane: while the regular cloud resource management addresses the running application in the unikernel VM, a specific configuration management

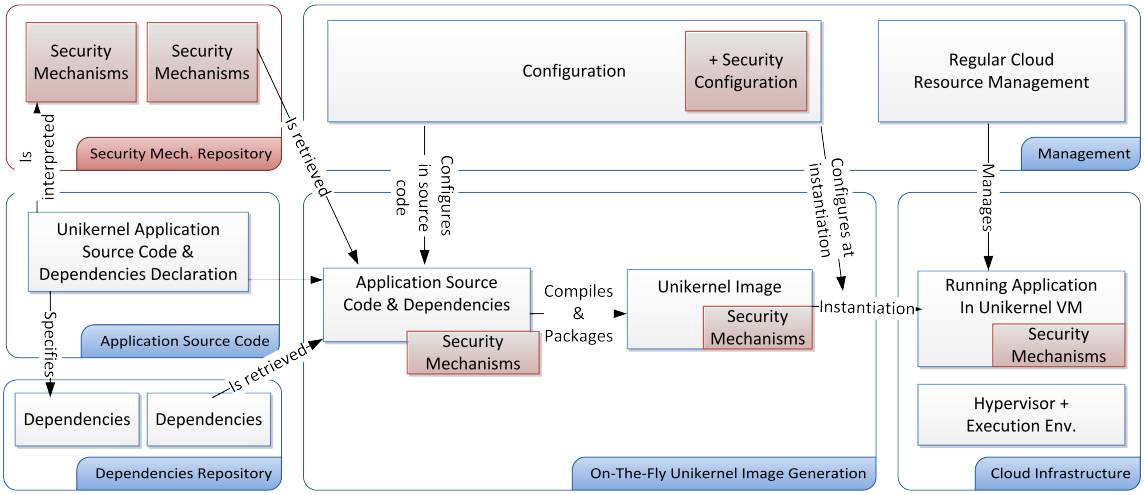


Figure 4.3: Software-defined security framework for cloud infrastructures with on-the-fly generation of unikernel images

aims at the application compilation or booting processes. We added the capability to handle the configuration of security mechanisms to meet security requirements.

This strategy based on unikernels enables a more comprehensive protection perimeter. These mechanisms are expected to address the protection of all the accessible resources of the unikernel VM, beneficing of the library OS paradigm to include the runtime components hardware management in its scope. This extends the potential protection perimeter of security components over resources in an instantiated unikernel VM. This also enables a highly-coupled enforcement, by deploying security mechanisms at the building step of unikernel images. We consider the installation of extra modules aside the application and their dependencies, but also consider minor modifications of their source code (i.e. code patching or additional source code inclusion). Unikernels provide the necessary environment to build images from source code and binary objects, and closed-source libraries requiring modifications (e.g. security hook insertion in routines) can be evaded with another implementation, taking benefits from the library OS philosophy. This tight integration dwarfs the enforcement overhead by enabling explicit collaboration between protected resources and mechanisms. Moreover, addressing the modification of every component of the unikernel extends the variety of potential enforcement features. We consider here the pro-active programmability of security mechanisms: we restrict the alteration of their configuration before their instantiation (e.g. configuration file edition, source code modification and boot parameter provisioning).

4.4.1 On-the-fly Unikernel Generation

We formalize the unikernels and their on-the-fly generation supporting our software-defined security approach for cloud infrastructures. We consider in that context that software component operations are enabled by software manager capabilities currently used in the unikernel ecosystem. We also take into account the programmability constraints related to software-defined security by addressing configuration management through a non-binding model, in respect with the lack of consensus on the configuration specification model in the unikernel ecosystem. We introduce a reference to a unitary software component integrable in a unikernel image as a module noted m. This terminology can refer to both software packages or injectable source code files,

which is consistent with considered software inclusion mechanisms for unikernels. The whole set of usable software modules is identified as $\mathscr{M}$, with $m \in \mathscr{M}$. Practically speaking, this matches all the packages in the configured repositories or all the libraries in the include directories. We also refer to c as a configuration option, and $\mathscr{C}(m)$ as the whole set of options applicable to a given module m. We expect c to be a condition that can be met by the code of the module m. Each constructible unikernel image u is encompassed in a set $\mathscr{U}$. We point out u_0 to be the nil element of this set, alluding to be the minimal image on which anyone is based.

This elementary modeling enables the definitions of basic operations on unikernels. The *Install* operation is the insertion of a module m in a kernel image u, resulting in a new kernel image u', as given by Equation 4.1.

$$\begin{aligned} Install :& \mathscr{U} \times \mathscr{M} \longrightarrow \mathscr{U}, \\ & (u, m) \longmapsto Install(u, m) = u' \end{aligned} \tag{4.1}$$

In the meantime, the *Configure* operation, given by Equation 4.2, permits to modify or alter the code embedded within a module m installed in a unikernel image u to meet a configuration option c, resulting in a new image u'

$$\begin{aligned} Configure :& \mathscr{U} \times \mathscr{M} \times \mathscr{C}(\mathscr{M}) \longrightarrow \mathscr{U}, \\ & (u, m, c) \longmapsto Configure(u, m, c) = u' \end{aligned} \tag{4.2}$$

Complementarily, we introduce observation operations to describe unikernel images. In particular, we define the *Modules* operation to get the set of inserted modules, and the *Configuration* operation to get the set of activated configuration options for a given module on a unikernel image. These operations are respectively given by Equations 4.3 and 4.4.

$$\begin{aligned} Modules :& \mathscr{U} \longrightarrow \mathscr{P}(\mathscr{M}), \\ & u \longmapsto Modules(u) \end{aligned} \tag{4.3}$$

$$\begin{aligned} Configuration :& \mathscr{U} \times Modules(\mathscr{U}) \longrightarrow \mathscr{P}(\mathscr{C}(\mathscr{P}(\mathscr{M}))), \\ & (u, m) \longmapsto Configuration(u, m) \end{aligned} \tag{4.4}$$

Several constraints are inferred from the unikernel properties. First, the installation operation does not remove any of the modules previously installed/inserted. This is a necessary condition to comply with the insertion mechanism, which only increases addressable routines in in-compilation objects and package managers that do not support conflict management. In that context, unikernel source code patching can be seen as a module insertion method. We also assume that the re-insertion of a module does not commit any modifications to the image, to layer the behavior of most package manager. This assertion is acceptable in the case of code inclusion, as most libraries protect themselves against multiple inclusions based on preprocessor directive usages. This constraint on modules is given by Equation 4.5.

$$\forall (u, m) \in \mathscr{U} \times \mathscr{M}, u' = Install(u, m) \implies Modules(u) \subseteq Modules(u') \quad (\text{equality if } m \in Modules(u)) \tag{4.5}$$

In addition, we conversely assume that fulfilling a particular configuration option can prevent meeting the requirement of another one. A typical example is expecting a unikernel application

variable to respect a value, while reconfiguring it with another one. This second constraint is given by Equation 4.6.

$$\forall (u, m) \in \mathscr{U} \times \mathscr{M}, \\ u' = Configuration(u, m) \not\Rightarrow u' \subseteq Configuration(Configure(u, m, c), m) \quad (4.6)$$

We also assume that the minimal unikernel image does not embed any module, as provided by Equation 4.7. This condition is required to permit the building of any unikernel image with solely the additive installation operation.

$$Modules(u_0) = \oslash \quad (4.7)$$

Finally, we consequently state the composability of every conceivable unikernel as the installation of multiple modules to the nil unikernel image, as described by Equation 4.8.

$$\forall u \in \mathscr{U}, \exists (m_i)_{1 \leqslant i \leqslant n} \in \mathscr{M}^{\mathbb{N}} / u = Install(Install(Install(u_0, m_1), \dots), m_n) \quad (4.8)$$

Our modeling introduces also two types of relationships to express the compatibility among software components in a unikernel image. These are inherited from package managers [93]. The dependency relationship is addressed by the $requires(m)$ operation, identifying the set of modules expected in a unikernel image before the installation of a given module m. The $conflicts(m)$ operation points the set of modules that prevents the installation of the module m. As a consequence, we consider that a given module is installable in a unikernel image when all its dependencies are already installed and no conflicting ones are present, as given by Equation 4.9.

$$IsInstallable(u, m) \Leftrightarrow \\ (Modules(u) \supseteq requires(m)) \wedge (Modules(u) \cap conflicts(m) = \oslash) \quad (4.9)$$

The generation of unikernel images relies on this modeling, and is given by Algorithm 1. It describes the different phases related to the building of unikernel images with security mechanisms. It takes as inputs the *imageSpecs* enumerative data structures providing the module and configuration dependencies, and *securityRequirements* enumerating additional security requirements on them. The algorithm also admits the procedures *Insert(datastructure,element)* for the insertion of *element* in the data structure *datastructure* and *Instantiate(u)* the unikernel instantiation. On lines 2 and 3, the algorithm fetches the updated version of the *imageSpecs* and *securityRequirements* data structures. Then, from lines 4 to 9, it exploits these two data structures to get the specification of the image embedding the protection mechanisms. The algorithm generates a new image from line 10 to 20, by initiating a blank one, and iterating over the specifications to install required modules and configure them. The module compatibility with the unikernel image is assessed with the *isInstallable*() function, which implements the *Module*(), the *requires*() and the *conflicts*() operations. The instantiation is set on line 21. The algorithm finally loops over these instructions each time a change in the *securityRequirements* data structure is notified.

The security mechanisms are therefore directly integrated to unikernel VM instances, as part of the application dependencies. This minimizes the enforcement disturbances due to external factors (e.g. network connectivity issues), and contributes to dwarf the attack surface, by limiting vulnerable communications. This approach also prevents the semantic gap [64] issue with respect to security enforcement. The security mechanisms do not have to construct a semantic view of

Algorithm 1 Unikernel image generation with security mechanisms for cloud software-defined security

```
1: repeat
2:     imageSpecs ← unikernel image specification
3:     securityRequirements ← unikernel sec. requirements
4:     for m ∈ securityRequirements do
5:         Insert(imageSpecs, m)
6:         for c ∈ securityRequirements[m] do
7:             Insert(imageSpecs[m], c)
8:         end for
9:     end for
10:    u ← u_0
11:    for m ∈ imageSpecs do
12:        if IsInstallable(u, m) then
13:            u ← Install(u, m)
14:        else
15:            throw exception
16:        end if
17:        for c ∈ imageSpecs[m] do
18:            u ← Configure(u, m, c)
19:        end for
20:    end for
21:    Instantiate(u)
22: until securityRequirements changes
```

in-VM data structures from the resources accessible to the hypervisor, in order to enable in-VM resource observations and changes.

Moreover, the additional code is integrated at unikernel compilation time, as regular source code. This permits an intrinsic integration of protective and protected mechanisms as they both undergo the same code optimization and linking processes, leading to minimize security mechanism overhead. This integration before unikernel VM instantiation goes in favor of security-by-design properties, on the condition that the remainder of the application code supports it. Currently, the configuration of security mechanisms themselves is performed before their instantiation in unikernel VMs in a pro-active manner. This means before the source code compilation (e.g. parameters inserted in source code) or at image instantiation (e.g. booting arguments). The configurations are highly constrained, requiring image re-building to cope with a new given context. This however contributes to reduce the complexity of a further security orchestration [119, 88].

4.4.2 Benefits of Unikernels for Software-defined Security

Independently from the integration of security mechanisms at compilation time, unikernels provide multiple benefits for software-defined security, in comparison to legacy VM architectures and containers. We detail them below.

Reduced attack surface. By requesting legacy features, by preventing unused interface and hardware management, and by focusing on applications and their sole dependencies, unikernels reduce the required code base for applications. This effort is also supported through the direct implementation of hardware resource management in applications, in accordance with the library OS principle, which evades the intermediate hardware abstraction layers. This coercive code consolidation reduces unikernel attack surface by circumventing software flaws and exploitable entry points. Besides, limiting the unikernel code base restricts the available capability of an attacker having succeeded in jeopardizing one instance. The restricted interfaces with other VMs minimize propagation vectors. The evacuation of unnecessary routines, legacy tools and legacy program support prevents an attacker from easily weaponizing an instantiated unikernel VM, or injecting regular exploitation tooling in it. It results in the clearance of any memory isolation in unikernel VMs. The only remaining barrier is provided by the system virtualization. This approach dismisses the multi-tenancy support in each VM instance, leaving unikernel VMs as the atomic resources to address tenancy and infrastructure location. This statement is acceptable as long as unikernel embeds a single application instance for one tenant. It even simplifies distributed cloud security management by oughting the assimilation of applications to VMs in the security policy specification.

Image building sanitization. The image construction tooling takes as input the unikernel source code to process it into a bootable VM unikernel image through source code static analysis, compilation and objects linking. The static analysis step attests the coherence of the code and its compliance with the programming language properties. The nature of the latter directly impacts the processing of this verification. For instance, a language featuring a strong typing system induces a deeper memory management assessment at compilation time. By scoping the codes of an application, its dependencies and its runtime, unikernel transposes these verifications to a broader, more inclusive field than a regular system stack. The choice of programming language through the retained unikernel solution therefore impacts the type of assessment conferable to a VM (i.e. dependability, safety or performance). From a performance perspective, addressing the

compilation tooling to a wider scope than only application contributes to reducing the weight image footprint. On one side, applying the same software management to the whole system architecture permits the evacuation of unneeded dependencies in areas not concerned by traditional software management. On the other side, the linking process affects wider stack and the architecture, obsoleting legacy interfaces between system component (i.e. process switch, sockets usage, system calls), circumventing the related overhead.

Code portability. As each unikernel image embeds an application and all the necessary software components for its execution, its outwits traditional compatibility issues between in-VM components. This restricts compatibility issues to the ones among the unikernel VMs and the hypervisor and their execution environment (i.e. computing resources, storage and networking prerequisites). Configured accordingly, several hypervisors can support the execution of the same VM. By considering hypervisors across several infrastructures, unikernels can embrace the multi-cloud constraint. In addition to the multi-tenancy viewpoint, this attests the compatibility of unikernels with distributed cloud environments.

4.4.3 Reactivity Improvement through Image Pooling

As an extension of the current framework, we incorporate the feature of unikernel image pooling to limit the triggering of the image generation process during a cloud service exploitation. We propose the building of several versions of the unikernels image, each one accounting for a different set of values for in-source-code configuration parameters. Each version of a unikernel image diverges from the others on the value affected to a parameter. The building is performed proactively to any unikernel image allocation. When a unikernel reconfiguration concerns a configuration parameter whose new value has been accounted in the image pooling process, the management plane solely have to deallocate the current VM, then proceed to the allocation of the image supporting the new value of the configuration parameter. This evacuates the building process for several reconfiguration case, dwarfing the delay for configuration update.

The components of the architecture have to undergo only minor modifications to support this extension: the unikernel image generator only has to support image pooling while the regular cloud management have to select to corresponding image at unikernel instantiation time.

The reconfiguration algorithm is modified to support it. Algorithm 2 presents the new version of the generation algorithms:

- The algorithm relies on additional structures: The *imagePool*, defined on line 32, stores already built unikernel images and enumerates them according to the image specification they correspond to. The *proactiveImageSpecs*, defined on line 33, gathers all the specification of the image to be built before any VM allocation. The function $IsAlreadyPooled()$, defined on line 25 to 30, takes as input a reference to the image pooling structure and an image specification and returns a boolean stating if a corresponding image has already been built and is available in the pool.

- The image process is altered on line 22 to systematically store the generated image in the pool structure with its corresponding specification.

- The main procedure of the algorithm proceeding the main loop is modified on lines 32 to 36 to accept the proactive generation of a unikernel image. This content is driven by the *proactiveImageSpecs* structure.

- The main loop incorporates an additional verification over the existence of a already-built image. This verification appears from lines 41 to 45 on the algorithm.

This extension leverages several benefits to the unikernel generation architecture. First, this extension combines a faster reconfiguration delay while still supporting the execution of highly constrained VMs. This dwarfs the delay to set up an effective enforcement following a policy update. Second, limiting the image generation reduces the resource consumption during the unikernel VM exploitation. This is a critical issue in limited resource infrastructures, as considered in edge-computing related use-cases.

The proposed extension is however subjected to two major shortcomings:

- The variating parameter provisioning alternative versions of the unikernel image is limited to a discrete and finite set of values. The proposed extension cannot cope with the contrary case as the *proactiveImageSet* structure would not be necessarily be finite, and could not be correctly iterated over. This is a strong condition over parameters that excludes configuration parameters from the pooling process such as continuous value range, or no size-limited strings. From a security policy perspective, this constraint impacts the scope of a policy that can be pooled. This shortcoming is partly circumvented by registering a new unikernel image in the pool at each build.

- The selection of the configuration parameters to be addressed by pooling has to be delegated to an entity with the knowledge of the relevant security parameters. This entity can be either a human security operator or a component in the management plane. It requires a holistic understanding of the secured unikernel resource from the management plane.

For these reasons, the resource pooling is not addressed by the remainder of this chapter. They are however accounted in the contribution of the next chapters.

4.4.4 Integration with the SDSec Architecture for Distributed Clouds

In chapter 3, we introduced a software-defined architecture for enforcing a policy-based protection over a distributed cloud. We focus here on the integration of this architecture with the unikernel generation framework. As a reminder, the considered SDSec architecture is based on the following components:

- The *PEPs* and the cloud resources are part of the security resource plane. They correspond to the assets requiring protection and the programmable security mechanisms leveraging it.

- The *PDPs* are part of the security management plane. They are in charge of the decision taking process at the tenant level for security mechanisms configuration. This process is framed by a security policy aligned with the tenant resources.

- The *security orchestrator* supports the security management at the scale of the cloud service. Unlike the PDPs, it does not directly configure the security mechanisms. However, it provisions them with the necessary elements to enable them to construct their tenant-level security policy.

- The *cloud orchestrator* is not related to the protections of resource. It is in charge of the life cycle management of cloud resources. Its behavior is aligned with regular cloud policy, but can interact with the security orchestrator.

Algorithm 2 Unikernel image generation with pooling features

```
1: function BuildSecureImageSpec(imageSpecs,securityRequirements)
2:     for m ∈ securityRequirements do
3:         Insert(imageSpecs, m)
4:         for c ∈ securityRequirements[m] do
5:             Insert(imageSpecs[m], c)
6:         end for
7:     end for
8:     return ImageSpec
9: end function
10: function BuildImage(imagePool, imageSpecs)
11:     u ← u_0
12:     for m ∈ imageSpecs do
13:         if IsInstallable(u, m) then
14:             u ← Install(u, m)
15:         else
16:             throw exception
17:         end if
18:         for c ∈ imageSpecs[m] do
19:             u ← Configure(u, m, c)
20:         end for
21:     end for
22:     imagePool[imageSpecs] ← u
23:     return u
24: end function
25: function IsAlreadyPooled(imagesPool,imageSpecs)
26:     if imagePool[imageSpecs] exists then
27:         return true
28:     else
29:         return false
30:     end if
31: end function
32: imagePool ← Already built images
33: proactiveImageSpecs ← Specification of image to be proactively build
34: for all imageSpecs ∈ proactiveImageSpecs do
35:     BuildImage(imagePool, imageSpecs)
36: end for
37: repeat
38:     imageSpecs ← unikernel image specification
39:     securityRequirements ← unikernel sec. requirements
40:     imageSpecs ← BuildSecureImageSpec(imageSpecs, securityRequirements)
41:     if IsAlreadyPooled(imagePool, imageSpecs) then
42:         u ← imagePool[imageSpecs]
43:     else
44:         u ← BuildImage(imagePool, imageSpecs)
45:     end if
46:     Instantiate(u)
47: until securityRequirements changes
```

The unikernel generation framework tackles the design of protected resources for a dedicated infrastructures. The configuration is configurable proactively, before the VM allocation, and reactively, during its allocation.

Several components are configured by the security management plane to meet security requirements:

- The unikernel application source code is modified before the unikernel image is built, to insert the security mechanisms. This requires the unikernel generation framework to expose interfaces to the security management plane, and therefore to be programmable.
- The cloud infrastructure in charge of VM allocation has to interact with the security management plane to provision the matching boot parameters. This infrastructure has to expose interfaces to be programmable.

Therefore, the security components inside unikernel VMs are assimilated to the PEP components. They are programmable through the unikernel generation platform and the infrastructure driving VM instantiations. The allocated unikernel VMs are the resources being protected by the framework. From a management viewpoint, the SDSec architecture and the generation framework converge as well:

- The security orchestrator has no direct equivalent in the generation framework, as no service wide security management nor collaboration with the cloud orchestrator are present in this framework.
- The security configuration handles the security decision taking process in the configuration of the secured resources and while accounting the knowledge to support it. This task corresponds to the PDP in the SDSec architecture.
- The remainder of configuration and regular cloud resource management shares the same objectives than the cloud orchestrator over the resource lifecycle. They can be assimilated to the same entity for the two solutions.

4.5 Performance Evaluation

In order to evaluate the proposed software-defined strategy exploiting unikernels, we designed and implemented a proof-of-concept prototype. It serves as a basis to quantify the performances of our approach through extensive series of experiments, and to compare them to traditional VM solutions. The considered security mechanisms support both the authentication and the access control for HTTP servers.

4.5.1 Prototype Implementation

We have implemented a proof-of-concept prototype based on an HTTP server over the MirageOS unikernel. As unikernels do not come with of-the-shelve applications but ought the developers to redesign them, we could not refer to a standard solution heavily deployed in production (e.g. Apache webserver in LAMP stack). Instead, we have considered the demonstrative HTTP server provided by the Mirage project itself [97], that has been complemented to insert the required security mechanisms. MirageOS is a unikernel solution based on the OCaml programming language. It features the compilation of VM images for Xen or KVM hypervisors or Unix executables. The OCaml language provides object-oriented programming and integrates a strong

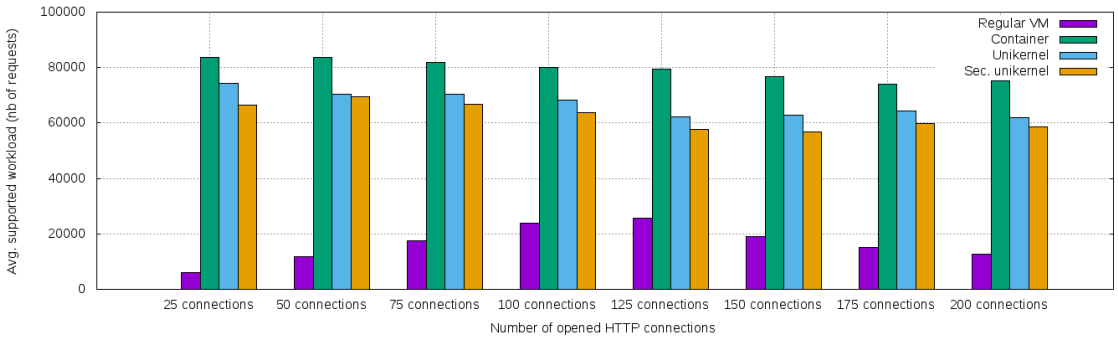


Figure 4.4: Average supported workload of securized unikernels compared to virtualization solutions

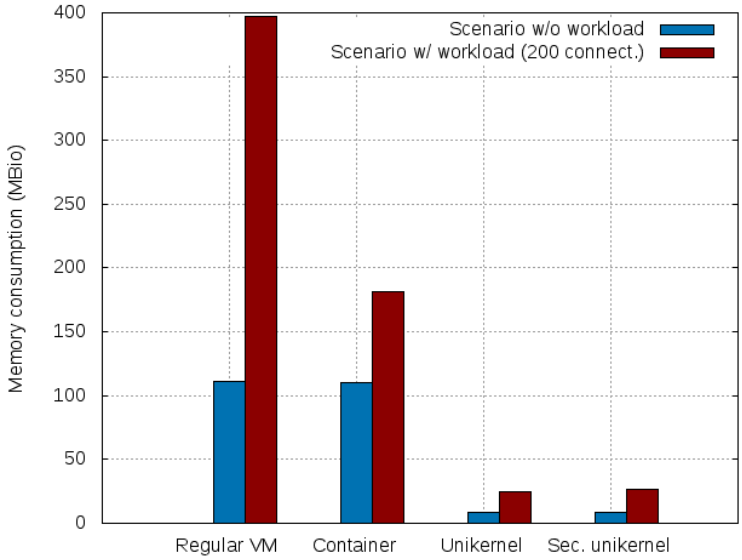


Figure 4.5: Memory consumption of securized unikernels

static and inferred typing model. This confers to the compiler the ability to statically detect most of typing error as well as some execution stream issues (e.g. incomplete pattern matching). Software resource are compartmentalized into namespaces called *modules*, and are loadable at glance. A garbage collector enables the resource reclamation of unaddressed software component. Although application, libraries and hardware resources management are directly managed in the OCaml runtime, the hardware resource bootstrapping is performed by a kernel feature library provided by third party projects: Xen platform is handled by mini-os project routines [164], while KVM is addressed by the Solo5 project [147].

The web server implementation is based on the CoHTTP HTTP processing library [33] provided by the Mirage project. The instantiated server listens for connections on TCP ports 80 (plain) and 443 (TLS). Incoming connections on port 80 are redirected to the second one through legacy HTTP redirection, while the ones on port 443 are TLS-ciphered with a sample certificate, and processed normally. The multi-threading permits to dispatch incoming client connections among several threads. The response edition to one client is consequently independent from the processing of another client. The configuration of the security mechanism is specified through

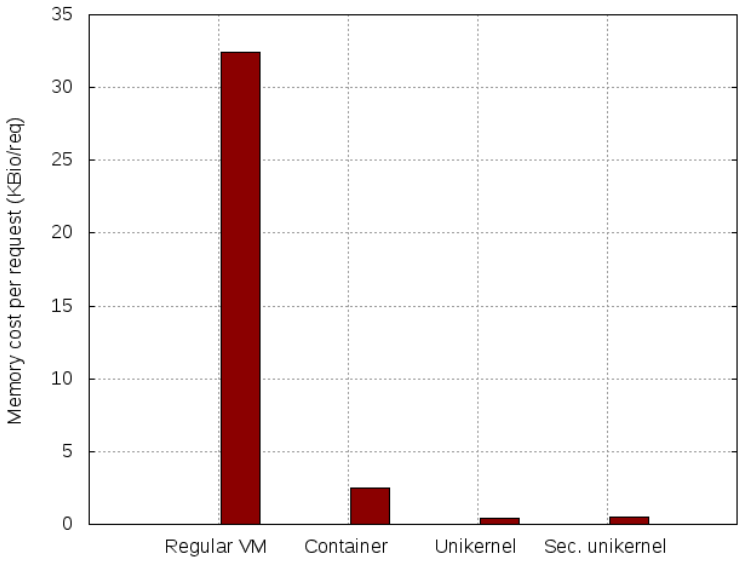


Figure 4.6: Memory cost per HTTP request of securized unikernels

its source code. Actually, there are two main configuration points: (i) a boolean specifying the mechanism behavior for anonymous or unrecognized credentials, (ii) a data structure for each enforced resources, enumerating themselves the accepted credential list. The security monitor we implemented as a module is integrated into the unikernel image. Its main procedure takes as input the HTTP client request, and returns a boolean according to the access granting defined in the configuration. To integrate it with the HTTP server, we had to insert one hook in the client request processing procedure. We have additionally added the support for the 403 HTTP error code response.

4.5.2 Qualitative and Quantitative Evaluations

From a qualitative viewpoint, this implementation has proven the feasibility to integrate security mechanisms in unikernel VMs, with a limited base modification from protected resources. Their performances together with hardware resource management and self-scalability capability through fast boot time identify them as considerable newcomers for cloud infrastructures, but also for VNF (Virtual Network Functions) design. We address their configurability before resource instantiation, sealing their configuration options values during their lifespan. In practice, this security module implements authentication and access control to resources: supplying the correct credentials corresponding to the accessed resource results in a 200-typed HTTP server response including the content of the related resource. Inversely, submitting the wrong credentials for a requested resource results in a 403-typed HTTP answer. Accessing a resource referenced in the security policy or not without supplying credentials results in a response specified by a dedicated configuration option.

We have performed the quantitative evaluation on the following testbed. As host virtualization system and image building environment, we have used a Intel i7-5500U CPU at 4x2.40GHz equipped with 16GB of RAM. It runs Ubuntu 16.04 LTS OS with Linux kernel 4.4.0-93, Oracle Virtualbox 5.0.40 for regular VM experimentation, Docker 1.12.6 as the container engine and ukvm 0.2.2 as unikernel monitor [160]. For each virtualization solution, we have allocated 1 vCPU, 512 MBio of RAM, and the latest versions of their software components (i.e. up-to-date

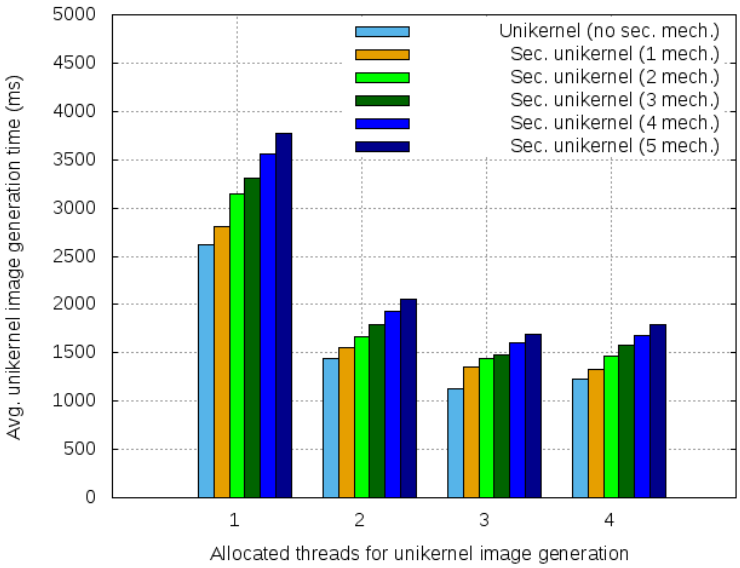


Figure 4.7: Unikernel image generation delay time

Ubuntu 16.04 LTS for Virtualbox VM and Docker image, and Mirage 3.0.4 for unikernel). Virtualbox VM and Docker containers provides Apache2 webserver with mod_ssl enabled. The Mirage unikernel implements CoHTTP-based webserver. Although being technically different, both webserver solutions are fully adapted to the virtualization architecture supporting them and are evaluated on common features. HTTP workload tests have been performed with WRK [49] while processes execution times are measured with `time` utility. HTTP servers are only requested through TLS-ciphered connections, with WRK or `curl`.

In a first series of experiments, we wanted to quantify the HTTP server performances with secured unikernels, and compare them to unikernels and other virtualization solutions. Figure 4.4 illustrates workload concerns by measuring successful HTTPS responses modulated by the number of concurrent connections: unikernels have been praised for their performances compared to regular OS. This assertion is confirmed with this workload analysis: although containers provide a more important workload support than unikernels, those have a substantially lighter memory footprint (Figure 4.5), dwarfing their memory cost per request (Figure 4.6). This experiment also states that the insertion of our security mechanisms slightly lowers the workload support (an average of 6.5% of reduction) with almost no impact on VM memory consumption (an overhead of 2 MBio with the 200-connections workload scenario, no overhead when idle).

In the second and third series of experiments, we were interested in evaluating the incidence of adding security mechanisms on the delay times required for generating unikernel images (Figure 4.7) and rebooting unikernel VMs (Figure 4.8). The generation process is analyzed by parameterizing the number of security mechanisms to be inserted and the allocated threads. We have focused on unikernel source configuration, compilation and packaging phases, at the expense of the dependencies installation, as it carries a bias for iterative time measurements and is not relevant for the recompilation case. The evaluated reboot process is considered accomplished when the HTTPS port is listened back. The large sampling of sequential reboots aims at evaluating the linearity of reboot delays related to the number of requested ones. In these experiments, we observed that security mechanism insertion induces an average overhead of 7.87% for image generation and preserves the linearity of unikernel VM reboot delays toward

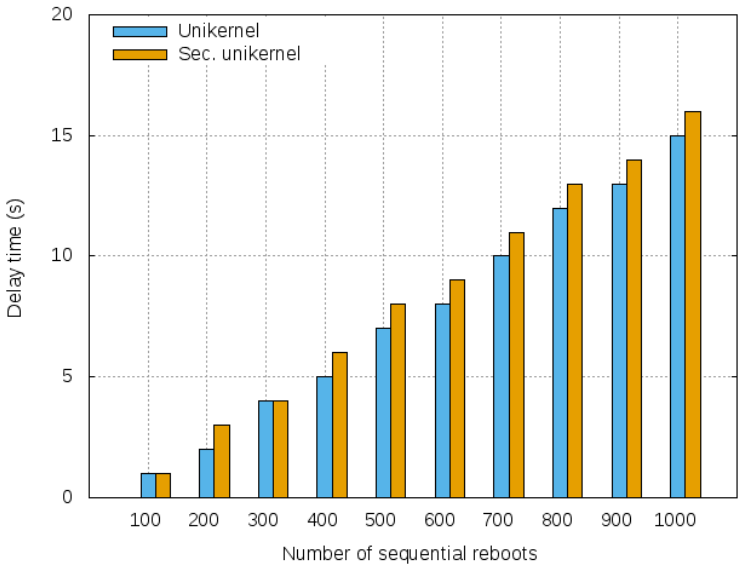


Figure 4.8: Unikernel VM reboot delay time

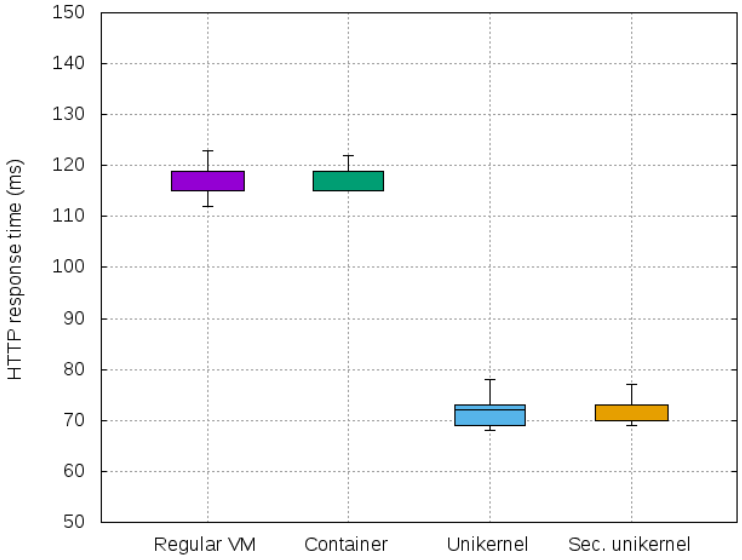


Figure 4.9: HTTP request delay time for an authorized access

the number of performed ones.

In a last series of experiments, we have compared the performances with respect to HTTP requests delay time over a unikernel with and without the built security mechanisms. The obtained results are detailed on Figure 4.9, together with the regular VM and container solutions without security mechanisms. The observed overhead induced by security mechanisms over the unikernel solution is limited to 0.2 ms on average during experiments. Secured unikernels sustain a 38% delay decrease over regular VMs. These different results show the benefits that unikernels bring to our software-defined security strategy.

4.6 Summary

We have proposed a unikernel generation approach for supporting software-defined security in cloud infrastructures. We have specified the underlying framework and formalized the on-the-fly generation of unikernel images that support this solution. We have exploited unikernels to build highly-constrained configurations limited to the strict necessary with a time-limited validity. Security mechanisms are directly integrated to the unikernel images at building time. A proof of concept prototype based on MirageOS was developed and the performance of such a software-based security strategy was evaluated through extensive series of experiments. We have also compared them to other regular virtualization solutions. Our results show that the costs induced by security mechanisms integration are relatively limited, and unikernels are well suited to minimize risk exposure. This unikernel generation framework constitues the second building block of our approach. In the next chapter, we will show how to extend a cloud orchestration language in the context of our work. The objective is to drive the generation of protected unikernels based on the presented framework and to enable an orchestration relying on different security levels, with the support of the SDSec architecture.

Chapter 5

Topology and Orchestration Specification for SDSec

Contents

5.1 Introduction

In the previous chapters, we have considered the software-defined security (SDSec) as an approach for supporting the programmability of security mechanisms that are used to protect resources offered by cloud infrastructures. In Chapter 3, we have analyzed the feasibility of such a security programmability layer for addressing multi-cloud and multi-tenant environments, through different realistic scenarios. The foundation of this layer relies on the SDSec logic to express and propagate security policies to the considered cloud resources, and on the autonomic paradigm to dynamically configure and adjust these mechanisms to distributed cloud constraints. We have also evaluated in Chapter 4 the benefits of using unikernel virtualization techniques to build and maintain specific cloud resources embedding security mechanisms. These lightweight virtual machines are built using a minimal set of libraries, enabling the reduction of the attack surface. We have defined an architecture for generating protected unikernels. However, it requires to be

orchestrated to integrate our SDSec architecture and contribute efficiently to the protection of cloud resources.

In this chapter, we propose to an extend the TOSCA[3] orchestration language for supporting our SDSec approach based on unikernels. The objective is to exploit this language, which supports the specification of cloud topologies and their orchestrations, in order to drive the integration and configuration of security mechanisms within cloud resources. This contributes to leverage a security-by-design cloud, from the specification of multiple levels of security requirements to the generation (and regeneration) of specific unikernel-based virtual machines to address them. The goal is be able to describe unikernel components and specify multi-level security requirements, in line with a security orchestrator and our unikernel generation framework. The protected unikernels corresponding to the different orchestrated security levels can be generated in a proactive manner, and are compatible with the elasticity and on-demand properties of cloud resources. Our main contributions in this chapter are (i) proposing and formalizing a software-defined security approach based on TOSCA for protecting cloud services, (ii) extending the TOSCA language to support our unikernel and multi-level security requirements, (iii) designing a framework capable of interpreting this extended language to generate and configure protected unikernels, and (iv) evaluating different SDSec strategies based on a proof-of-concept prototyping.

The remainder of this chapter is organized as follows. Section 5.2 presents existing work in the areas related to cloud security. Section 5.3 gives an overview of our TOSCA-oriented software-defined security approach. The extensions of the TOSCA language are described in Section 5.4, while Section 5.5 describes the underlying framework exploiting them to enable security-by-design clouds. Section 5.6 concludes the chapter and points out additional research perspectives.

5.2 Related Work

Our work on distributed cloud security concerns the enforcement of security requirements that impact both the orchestration and the building of cloud resources.

Important efforts have been spent to support security programmability and orchestration in cloud environments. The programmability of resources has been developed in the area of software-defined networking, with multiple applications dedicated to security management. For instance, [84] proposes and evaluates delegation strategies for enforcing security mechanisms at the level of SDN switches, and [110] defines several dynamic access control methods taking into account the current threat levels with respect to devices in SDN environments. This programmability contributes to a more flexible composition of security functions. A typical example can be given with the FRESCO framework [142] which supports modular security functions that can be composed into security chains to protect resources. [119] also considers a complete life cycle management of virtual network functions, driven by security constraints. Security programmability should not be limited to network-based mechanisms, but should of course also consider system-based and software-based enforcements in cloud environments. Authors of [52] envision an architecture for protecting virtualized resources with multiple security functions and propose several protection use-cases. Existing work in policy-based management have contributed to extend security languages for cloud infrastructures. For instance, [135] proposes policy specifications for supporting cloud security, while considering a medium-agnostic enforcement including network-based and system-based counter-measures. Another important

[3]Topology and Orchestration Specifications for Cloud Applications

aspect concerns the minimization of virtual machines to the strict necessary components and libraries, in order to both increase their performance and restrict the attack surface. We argue in favor of exploiting unikernels to minimize the attack surface and have showed in the previous chapter how such unikernel-based virtual machines can be generated in an on-the-fly manner. It is essential to take into account the generation of such protected virtual machines into orchestration languages, in order to support cloud security, right from the design phase. Such extensions therefore require to describe the software components that compose unikernel virtual machines. A large variety of description languages has already been proposed in the literature, coming from software engineering, but also from service design. Historically, software programming has contributed to several description standards [78] to address internal software interactions amongst routines. Extensions have also been specified to integrate security requirements, such as [85]. These descriptions are often too fine-grained and do not address exploitation considerations. The aspect-oriented programming has inspired work in [9] to specify and enforce access control policies in pervasive environment. However, the considered security functions remain closely tied to the access control. In the meantime, service design efforts provide another description scale. For instance, [93] provides service descriptions, by considering packages and their dependencies on Linux operating systems. The cloud orchestration languages, such as TOSCA, should take into account such descriptions. While they support the specification of the topology and orchestration of distributed cloud services, they only rely on off-the-shelves software descriptions [20]. This integration is an important lack to support security requirements from the design to the orchestration of services.

5.3 TOSCA-Oriented Software-defined Security Approach

We propose a software-defined security approach based on the TOSCA language [118], in order to protect cloud infrastructures using unikernels. This topology and orchestration language provides a support to describe distributed cloud services. It is extended to specify security requirements, according to different orchestrated levels. The extended language serves as an input to our security framework, which drives the generation and configuration of protected cloud resources, as depicted on Figure 5.1. These resources rely on unikernels, corresponding to lightweight virtual machine, and are characterized by a low attack surface. They only contain the strict necessary software components and libraries, and embed security mechanisms. The integration of security mechanisms at the design phase goes in favor of further security-by-design for distributed orchestrated cloud environments. The generation of unikernel virtual machines is performed in an on-the-fly manner, in order to cope with contextual changes related to threats and risks. Our SDSec approach relies on several challenging requirements that are detailed below.

A first major requirement concerns the adequacy to distributed cloud. Our goal is to protect services that are typically distributed over different clouds and involve multiple tenants. The services are specified as logical combinations (also called topologies) of cloud resources. The language also details their relationships, and the processes that manage them. In that context, we exploit the TOSCA language for security purposes. We consider a dedicated extension to specify the security policy of the described service. This abstraction is in phase with the programmability of security mechanisms over cloud infrastructures. These mechanisms are typically spread over tenants and infrastructures. This extension contributes to a more consistent specification of distributed and heterogeneous configurations. Moreover, we consider the whole life cycle of the cloud service in this security policy, going from the design of protected cloud resources to their orchestration. This includes the integration of security mechanisms at the building of resources,

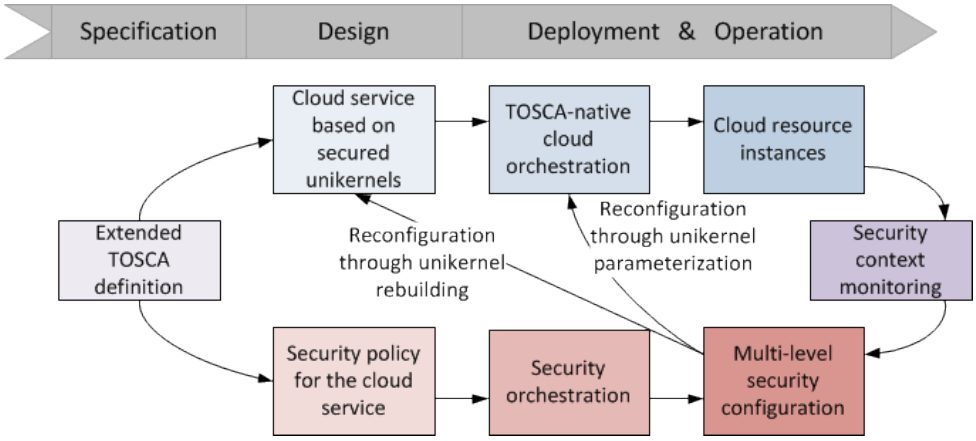


Figure 5.1: From the specification of TOSCA-based security requirements to the generation and operation of secured unikernel virtual machines

in accordance with a security-by-design approach. The security policy is taken into account by our security framework, which includes a cloud resource orchestrator and a security orchestrator.

A second important requirement concerns the security enforcement based on unikernel virtualization. The minimization of virtual resources permits a better control of the attack surface, through the usage of the strict necessary code base. The security enforcement relies on the TOSCA language to drive the generation of secured unikernel virtual machines. These machines embed the required security mechanisms, which depend on the technical constraints imposed by the cloud environments and its tenants. In order to enable such a fine-grained enforcement, we consider an extension of the TOSCA language to describe unikernel resources, in addition to the security requirements already mentioned previously. This permits to exploit the expressiveness of the language to describe the unikernel resources to be protected with their relationships amongst other resources while keeping a human-readable format. The nature of security requirements depends on the considered security function (e.g. access control vs. intrusion detection/prevention), and their scope can be specified at different levels of granularity, from a single unikernel to a whole distributed cloud service. The enforcement of the security policy may of course rely on different security functions, which may be realized by multiple security mechanisms depending on the environment constraints. Following a software-defined paradigm, the resources and their security mechanisms are decoupled from the security management.

A third requirement is about the adaptation to security-related contextual changes. The security policy should be continuously enforced along the protected service life cycle, including after a change in the security context of service resources. These changes include the ones related to the security policy and their propagation to protected resources, but also the others related to resource exploitation that may be triggered by cloud orchestration (e.g. scalability, elasticity, resource reconfigurations) or be due to external parties (e.g. resource customer). In the case of unikernel resources, this adaptation may imply a complete resource rebuilding. The TOSCA language enables through its orchestration facilities to anticipate the changes that may occur, including changes that may concern the service topology. In particular, several unikernel virtual machines corresponding to different security level requirements may be generated in a proactive manner.

Based on these requirements, we introduce extensions for the TOSCA language to specify the security requirements in our context, and propose a framework to enforce these requirements

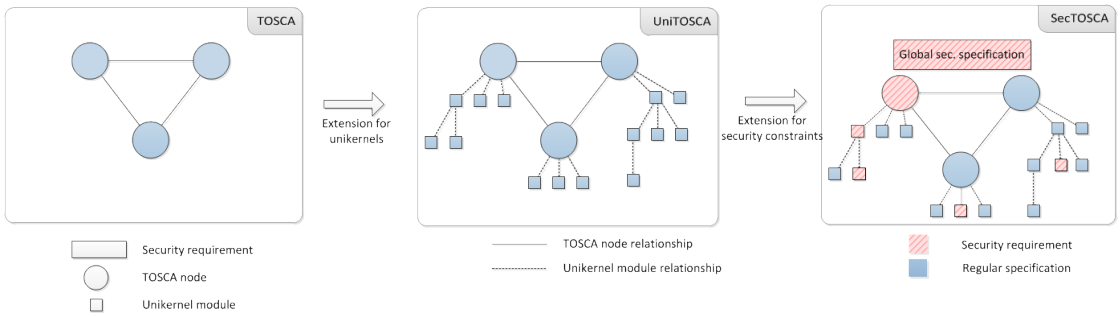


Figure 5.2: Extensions of the TOSCA language for describing unikernels (UniTOSCA) and specifying security requirements (SecTOSCA)

based on the generation of secured unikernel virtual machines.

5.4 Extensions of the TOSCA Language

In order to support our software-defined security solution, we have first extended the TOSCA orchestration language, which provides a baseline for describing distributed and orchestrated cloud services. These extensions are depicted on Figure 5.2. The first extension, called UniTOSCA, permits to refine the description level of TOSCA in the context of services implemented based on unikernels. It introduces a specific type for unikernels, and permits to describe them as a composition of software components. This description is then exploited to generate unikernels required by a cloud service. The second extension, called SecTOSCA, permits to specify security constraints in the TOSCA language. As previously mentioned, the scope of these constraints goes from a single unikernel to a whole cloud service. They are used to enforce security over resources using dedicated mechanisms (firewalls, intrusion detection systems, access control). This enforcement is performed in a dynamic manner to adapt to contextual changes and takes benefits from the orchestration facilities offered by TOSCA. In particular, it is possible to specify different security levels to cope with various contexts. After presenting the key concepts of the TOSCA language, we will detail successively the two UniTOSCA and SecTOSCA specifications, with illustrative examples.

5.4.1 The TOSCA Language

The TOSCA language stands for Topology and Orchestration Specification for Cloud Applications. It is a standardized specification for describing the deployment and orchestration of cloud services. It serves as an input for cloud orchestrators to determine the resources to be instantiated and the operations to configure and operate them, in order to provide a given service. As depicted on the left part of Figure 5.2, a cloud service is described by the TOSCA language as a topology of resources (also called *nodes*) that are interconnected amongst them through links (also called *relationships*). It is then possible to specify orchestration procedures over this topology, such as starting, shutting down a node or changing a relationship. Each element (node or relationship) takes benefits from inheritance and template mechanisms offered by the language. Following an object-oriented paradigm, each type defines a class of elements sharing a common set of properties and interfaces, while a template defines a type with a set of pre-defined values affected to properties. An instance can be obtained from a template and corresponds to an implementation of the resources in a given contextual environment. The lan-

```
topology_template:
    relationship_templates:
        load_balancing:
            type: tosca.relationships.RoutesTo
            interfaces:
                Standard:
                    configure: lb_configure.sh

    node_templates:
        my_server:
            type: tosca.nodes.Compute
            capabilities:
                host:
                    properties:
                        num_cpus: 2
                        mem_size: 2048 MB
                        disk_size: 10 GB

        web_server:
            type: tosca.nodes.WebServer
            capabilities:
                data_endpoint:
                    properties:
                        port_name: 8080
            requirements:
                - host: my_server

        front_portal:
            type: tosca.nodes.loadbalancer
            properties:
                algorithm: fifo
            requirement:
            - application:
                node: web_server
                relationship: load_balancing
            interfaces:
                Standard:
                    configure: lb_configure.sh
```

Figure 5.3: Example of a simple TOSCA specification

guage also permits to specify relationships in a implicit manner, using requirements (specifying what the node expects from other nodes on hosting infrastructures) and capabilities (specifying what the node may provide to other nodes on the infrastructures). Figure 5.3 is an example of a TOSCA specification inspired from [136]. In its current form, the TOSCA specification is non normative with respect to the orchestration policy. Interfaces are typically used to define

the operations performed on the nodes, following traditional workflow and process formalisms. In order to clarify the terminology introduced by TOSCA and provide the materiel to formaly define our processing on TOSCA specification, we can define a node type T_{node}, as given by Equation 5.1, where *prop* corresponds to the properties of the node, itf indicates its interfaces, *req* and *cap* stand for its requirements and capabilities.

$$T_{node} = \langle prop, itf, req, cap \rangle \tag{5.1}$$

It is then possible to specialize a given node type T_{node} into a node template T'_{node}, as given by Equation 5.2, where $prop'$, cap' and req' permits to refine the node type. The interfaces are not concerned by this specification and are kept unchanged.

$$T'_{node} : T_{node} \longmapsto \langle prop', itf, req', cap' \rangle \tag{5.2}$$

A node instance I_{node} is then generated from a node template T'_{node}, as given by Equation 5.3. A template can be seen as a subclass, while an instance corresponds to an object. This latter is a resource deployed and running on an hosting infrastructure given a provided execution context e.

$$I_{node} : \langle T'_{node}, e \rangle \longmapsto \langle prop'(e), itf, req'(e), cap'(e) \rangle \tag{5.3}$$

We consider the same formalism to represent the explicit relationships supported by the TOSCA language. Therefore, a relationship type, noted $T_{relationship}$, can be specialized into a relationship template, noted $T'_{relationship}$. This one can be deployed as a relationship instance, noted $I_{relationship}$. The TOSCA language offers several benefits with respect to our approach. In particular, it provides a support to describe distributed cloud services as topologies interpretable by cloud orchestrators, the interfaces allowing to specify orchestration operations to operate them. In addition, the notion of types, templates and instances is in phase with the abstraction level required by our software-defined strategy, enabling the decoupling between the control and implementation planes. Finally, it is easily extensible to cover our specific requirements with respect to unikernels.

5.4.2 Describing Unikernels

We extend the TOSCA language in order to describe unikernel virtual machines. This extension is represented in the middle of the Figure 5.2. The purpose is both to increase the granularity of the language, and to drive the building of unikernels before their instantiation. For that, we introduce an additional element, called unikernel component m, in order to describe routines and compose them to elaborate unikernels. The relationships amongst these components correspond to the dependencies that may exist amongst routines. These components are characterized by attributes and values that are configurable. However, taken separately, each of them cannot lead individually to a resource instance. A minimal set of routines is required to be composed in order to generate such an instance. In phase with our approach exposed in chapter 4, we take benefit from the simplified system architecture of unikernels to compose and build resources. This description of unikernel resources enables the orchestrator to take in charge the building and parameterization of these resources. The knowledge on unikernel components provided by the UniTOSCA extension facilitates the adaptation to contextual constraints by the orchestrator, while keeping a relatively simplified modeling of unikernel resources. The consistency of images generated from the descriptions relies on the dependency relationships amongst components, the satisfaction of these dependencies is checked before generating unikernel images that are used to elaborate the services. We consider a description of unikernel resources in phase with

Resources	**Node template**	**Unikernel component**
Scope	Instantiable resource	Routines composing an instantiable resource
Implementation	Virtual machine images or applications	Source code of the routine
Interface	Used to operate a resource	Used to build a resource
Configuration	Partial at runtime	Fixed at runtime, required rebuilding
Relationships	Distribution amongst hosting infrastructures	Dependencies amongst unikernel components

Table 5.1: Comparison of node templates (TOSCA) and unikernel components (UniTOSCA)

the description of the other TOSCA resources. An example of such a specification is given in Figure 5.4, where we detail a unikernel resource, called `my_unikernel`, composed of three unikernel components. These components detail each routine on which the resource is built, considering a finer granularity than regular TOSCA resources.

We can represent formally a unikernel component, in a similar manner than a node, but at a different granularity, as given by Equation 5.4, with $prop_m$ standing for the component properties, itf_m representing its interfaces, req_m indicating its requirements, and cap_m specifying its capabilities. The specified components serve as a basis for building TOSCA node type, as given by Equation 5.5 where $\langle m_1, ..., m_k \rangle$ indicates a consistent subset of unikernel components. These types are then turned into node templates. The building of unikernel images will be further detailed in the next section.

$$m = \langle prop_m, itf_m, req_m, cap_m \rangle \tag{5.4}$$

$$image_{uni} : \langle m_1, ..., m_k \rangle \longmapsto T_{node} \tag{5.5}$$

The choice of inferring node types (instead of node templates) from a set of unikernel components is motivated by a pragmatic approach consisting in specializing the unikernel resources through a template. The unikernel components selected to build the resource permit to infer the main properties of the corresponding node type. Our approach distinguishes the configuration associated to the unikernel image specifically built from the parametrization at instanciation time (e.g. boot arguments). It is interesting to compare the notion of node templates and unikernel components, as detailed in Table 5.1. First, they do not address the same granularity. The unikernel components permit to build node types, and then to infer node templates. The node templates refer to instantiable resources used during the service deployment and operation phases. These resources can be composed through orchestration operations. Complementarily, unikernel components are only exploited during the service building, in order to elaborate a resource from a set of routines. In addition, node templates define a predefined sets of properties and interfaces, while unikernel components inject their own ones to the nodes they are contributing to during their building. This fine granularity enables us to drive our software-defined security solution based on unikernels.

```
unikernel_modules:
    my_unikernel:
        cohttp_lwt_server:
            capabilities:
                http_processing: tosca.capabilities.Endpoint
            requirements:
                - available_nic: unikernel.virtualenv.networking
                    .nic
        log:
            capabilities:
                console_logging: unikernel.virtualenv.console
        dispatch:
            properties:
                https_port:
                    type: integer
                http_port:
                    type: integer
            capabilities:
                webserver_front: tosca.capabilities.Endpoint
            requirements:
                - http_processing: tosca.capabilities.Endpoint
                - console_logging: unikernel.virtualenv.console

topology_template:

    relationship_templates:
        load_balancing:
            type: tosca.relationships.RoutesTo
            interfaces:
                Standard:
                    configure: lb_configure.sh

    node_templates:
        my_unikernel_vm:
            type: my_unikernel_type
            properties:
                name: vm1

        front_portal:
            type: tosca.nodes.loadbalancer
                properties:
                    algorithm: fifo
                interfaces:
                    Standard:
                        configure: lb_configure.sh
```

Figure 5.4: Example of a UniTOSCA specification exploited to describe a unikernel resource as a set of routines

5.4.3 Specifying Security Requirements

Based on this language leveraged by unikernel components, we propose to specify the security requirements related to the considered cloud services. We therefore introduce an extension of the TOSCA language, called SecTOSCA, which is represented on the right part of Figure 5.2. This extension serves as a support to define the security policy related to our SDSec approach. It contributes to the decoupling of the security management logic from the security enforcers and the resources to be protected, in line with a policy-based management strategy. In that context, we consider a security orchestrator, complementary to the cloud resource orchestrator, taking in charge the security policy and the configuration of security functions. We consider a security function (access control, intrusion detection, encryption mechanisms) to be a feature aiming at enforcing a set of security requirements (access control lists, firewall rules) on the cloud resources. The security requirements can be specified at different scales, from a single unikernel to a whole cloud service. The TOSCA language already includes a non-normative policy specification. Some efforts have already shown the benefits of exploiting this policy in a specific use case, in order to specify access control rules on resources in [118]. We argue in favor of exploiting TOSCA to specify a security policy capable of covering the deployment and operation phases, but also the building of unikernel resources. The goal is to enable a security enforcement at an early stage through the generation of specific unikernel components and resources.

In addition, we take benefits from the orchestration facilities of the TOSCA language in order to specify several security levels, with respect to a given context. Our SecTOSCA extension enables an adaptation to contextual changes in two different manners. First, security mechanisms expose interfaces, enabling the security orchestrator to adjust their configuration parameters. This parameterization is performed on the instance of a node I_{node}, where the security level can be dynamically changed by the security orchestrator, as given by Equation 5.6.

$$configure_{sec} : \langle I_{node}, level_{sec} \rangle \longmapsto I_{node} \tag{5.6}$$

Second, the resources themselves can be rebuilt at runtime to cope with security constraints. In particular, this concerns unikernel resources embedding security mechanisms, which can be dynamically re-generated to cope with security constraints. To that purpose, we extend the $image_{uni}$ (defined in Equation 5.5) method into $building_{sec}$ by introducing an additional parameter, so that the security level can be specified, as given by Equation 5.7. In that case, the result is not an instance, but a node type T_{node}, which is then specialized and instantiated as an instance I_{node}.

$$building_{sec} : \langle \langle m_1, ..., m_k \rangle, level_{sec} \rangle \longmapsto T_{node} \tag{5.7}$$

The SecTOSCA specification details the different security levels that can be required for the cloud service. The generation of unikernel virtual machines with different security levels can be performed in a proactive manner. The security orchestrator can therefore efficiently order to the resource orchestrator the deployment of a new instance of a given unikernel, from a pool of already generated unikernels.

5.4.4 An Illustrative Case

In order to illustrate our approach, we give an example of a SecTOSCA specification in Figure 5.5. We consider the case of a service exposing static resources over an HTTP server. These resources correspond to dedicated unikernel virtual machines, while a load balancer is in charge

```
unikernel_modules:
  my_unikernel:
    cohttp_lwt_server:
      capabilities:
        http_processing: tosca.capabilities.Endpoint
      requirements:
        - available_nic: unikernel.virtualenv.networking.nic
    log:
      capabilities:
        console_logging: unikernel.virtualenv.console
    dispatch:
      properties:
        https_port:
          type: integer
        http_port:
          type: integer
      capabilities:
          webserver_front: tosca.capabilities.Endpoint
      requirements:
          - http_processing: tosca.capabilities.Endpoint
          - console_logging: unikernel.virtualenv.console

topology_template:
  node_templates:
    my_unikernel_vm:
      type: my_unikernel_type
      properties:
         name: vm1
         vm_security_level:
           multi_level_security:
             default_security_level: medium
             critical_security_level: high
         nvi_pop: openstack_infra1
         tenant_domain: tenant1
         members: {{user1,high},{user2,medium}, ... , {user5, low}}

    front_portal:
      type: tosca.type.loadbalancer
        properties:
          lb_ddos_mitigation_level:
            multi_level_security:
              default_security_level: regular_mitigation
              critical_security_level: paranoid_mitigation
          nvi_pop: openstack_infra1
          tenant_domain: tenant1
      interfaces:
        Standard:
          configure: lb_configure.sh

security_group:
  VM_SP1
    type: groups.unikernel
    description: defining security group for tenant1
    target: {vm1}
```

Figure 5.5: Example of a SecTOSCA specification

of balancing incoming connections to each unikernel instance. The load balancer has a built-in DDoS[4] mitigation mechanism, which can be activated on demand. The unikernel virtual machines are based on three unikernel components: `cohttp_lwt_server`, `log` and `dispatch`. To secure the service, we want to (i) control the DDoS mitigation mechanism offered by the load balancer and (ii) enforce an access control on unikernel virtual machines. In Figure 5.5, we can observe a `unikernel_modules` section which describes the composition of the unikernel image `my_unikernel` with no reference to security mechanisms. The `topology_template` section enumerates both the load balancer (`front_portal`) and the unikernel virtual machine (`my_unikernel_vm`). The security requirements are specified in two manners:

1. The configuration of the DDoS mitigation mechanism provided by the load balancer is characterized by the `lb_ddos_security_level` property part of the `front_portal` node. The security levels may depend on different contextual parameters, such as the presence of specific threats for local monitoring results. In this example, two different security levels are specified and correspond to regular and paranoid mitigations (`regular_mitigation` and `paranoid_mitigation` values).

2. The access control on unikernel virtual machines is specified by additional properties corresponding to `vm_security_level`,`nvi_pop`,`tenant_domain` and `members`. The specification of access control requirements follows the same formalism as [119]. Again, the `vm_security_level` property enables multi security levels so that the access control can be changed with respect to current threats. The last section `security_group` enumerates the information specifically related to the access control.

In the following of this chapter, a SecTOSCA specification will serve as a basis to define our security policy. We will then successively infer from it an enriched UniTOSCA specification, and a TOSCA specification. When we refer to Figure 5.2 presenting the different extensions, it looks like we are taking the reverse path to obtain a TOSCA specification. The purpose is to guarantee the compatibility of our solution with TOSCA-native architectures. The enriched UniTOSCA specification will include the security mechanisms to be embedded into the unikernel resources, while the TOSCA specification will refer to pre-compiled protected unikernel images.

The corresponding UniTOSCA specification is illustrated by Figure 5.6. It diverges from its SecTOSCA counterpart mainly by the removal of all security-related information. In accordance with the formalism from [119], `security_group` section, `tenant_domain`, `members` and `nvi_pop` properties are removed. The node properties subjected to security levels are also affected with the values of the `default_security_level`. As a counterpart, this specification integrate the inclusion of the `accesscontrolmodule` security mechanism, in charge of enforcing a security policy.

Finally, the TOSCA specification is generated from the UniTOSCA by compiling requirements, capabilities and properties information from the software components in the `unikernel_modules` section to node template. The properties are also affected with default values while implementation artifacts are specified. Figure 5.7 draws an example of the produced TOSCA specification.

5.5 Underlying Security Framework

In this section, we detail how the proposed extended specification supports the design and management of protected cloud services. For that purpose, we introduce a security framework in

[4]Distributed Denial-of-Service

```
unikernel_modules:
  my_unikernel:
    cohttp_lwt_server:
      capabilities:
          http_processing: tosca.capabilities.Endpoint
      requirements:
          - available_nic: unikernel.virtualenv.networking.nic
    log:
      capabilities:
          console_logging: unikernel.virtualenv.console
    dispatch:
      properties:
        https_port:
          type: integer
        http_port:
          type: integer
      capabilities:
        webserver_front: tosca.capabilities.Endpoint
      requirements:
        - http_processing: tosca.capabilities.Endpoint
        - console_logging: unikernel.virtualenv.console
    accesscontrolmodule :
      attributes:
 - security_monitor_url: "http://192.168.0.42"
 - security_domaine: "tenant_1"
      capabilities:
 access_control_security: security.accesscontrol.http
      requirement:
 webserver_front: tosca.capabilities.Endpoint

topology_template:
  node_templates:
    my_unikernel_vm:
      type: my_unikernel_type
      properties:
        name: vm1

    front_portal:
      type: tosca.type.loadbalancer
      properties:
        algorithm: fifo
      interfaces:
        Standard:
          configure: lb_configure.sh
```

Figure 5.6: Example of a UniTOSCA specification generated from a SecTOSCA specification

```
node_types:
  my_unikernel_type:
    properties:
      https_port:
        type: integer
      http_port:
        type: integer
    capabilities:
      access_control_security: security.accesscontrol.http
    requirements:
      - available_nic : unikernel.virtualenv.networking.nic
      - console_logging: unikernel.virtualenv.console

topology_template:
  node_templates:
    my_unikernel_vm:
      type: my_unikernel_type
      properties:
        name: vm1

    front_portal:
      type: tosca.type.loadbalancer
      properties:
        algorithm: fifo
      interfaces:
        Standard:
          configure: lb_configure.sh
```

Figure 5.7: Example of a TOSCA specification issued from a SecTOSCA specification

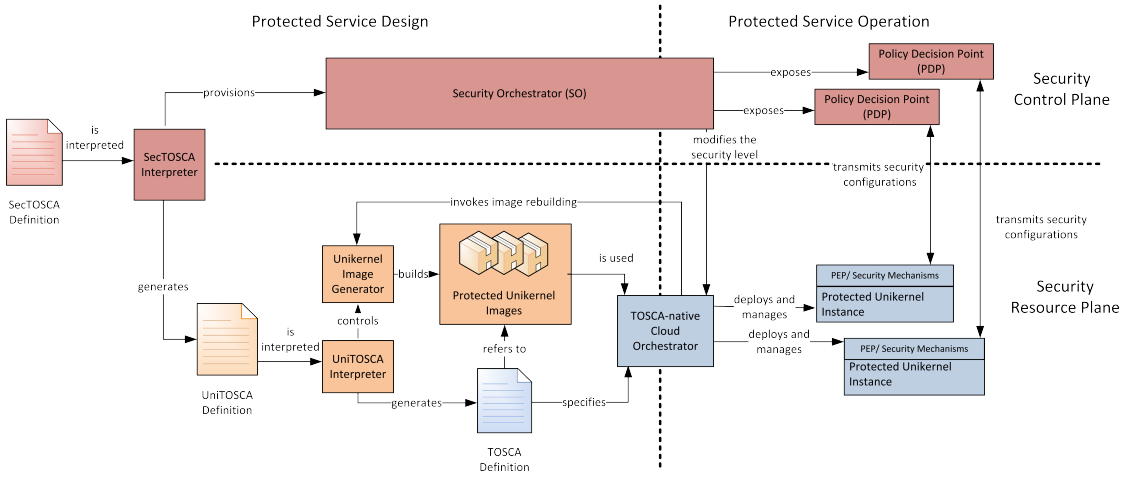


Figure 5.8: Overview of the TOSCA-oriented SDSec framework for protecting cloud services

line with the concepts of SDSec, and analyze the different steps related to the operation of this solution based on unikernels. Coupling the design of protected resources with their management is a challenging issue, whose complexity is increased by the distribution and heterogeneity of resources.

In order to address it, we organize the security framework according to three different tasks. The considered framework is depicted in Figure 5.8. First, it takes in charge the building of protected resources implementing the cloud service, as represented by yellow blocks on the Figure. These resources have to be compatible with security programmability. Considering such a resource-centric strategy enables a fine-grained security enforcement. The UniTOSCA specification supports the design of protected unikernel images, embedding SDSec-capable security mechanisms. Second, it permits the management of security mechanisms with respect to security requirements specified by the SecTOSCA specification. Decoupling this management from protected resources, as represented by the two planes (security management plane and security resource plane) facilitates the support of distributed and heterogeneous environments. This task corresponds to the red blocks in the Figure. Third, the framework supports the adaptation to contextual changes. The purpose is to maintain security enforcement when changes occur over resources and their environments. The TOSCA-based orchestration, represented by blue blocks on the Figure, addresses the whole life cycle of resources and can notify any changes that may occur on resources.

5.5.1 Main Components

The proposed framework includes several components depicted in Figure 5.8. In addition to the three tasks previously mentioned, we can observe two different axes: one horizontal axis referring to security programmability and distinguishing the security management plane from the security resource plane, and another vertical axis distinguishing the design/building of protected resources from their deployment and operation. It takes as input a SecTOSCA specification, serving as starting point to build and orchestrate protected resources embedding security mechanisms. We detail below the role of the main components:

- **SecTOSCA interpreter.** The role of this interpreter is to analyze a SecTOSCA specification and provide security requirements to the security orchestrator. It also produces

a UniTOSCA specification detailing the unikernel resources to be generated with the embedded security mechanisms supporting security enforcement.

- **Security orchestrator.** This component is responsible for translating security requirements into a consistent orchestration of security mechanisms that are distributed over resources to be protected. It interacts with policy decision points (PDP) capable of taking into account specific tenant and host requirements. As an example related to access control, the security orchestrator is fed with groups of entities allowed to access each others. The security orchestrator constitutes access control list aligned with those groups. These PDPs are then in charge of parameterizing Policy Enforcement Points (PEP) corresponding to the security mechanisms embedded on protected resources.

- **UniTOSCA interpreter.** This interpreter analyzes a UniTOSCA specification and is capable to infer a TOSCA-native specification. It drives the generator of unikernels which builds protected unikernel resources from the description of unikernel components. The TOSCA specification refers to the unikernel images that are produced by the unikernel generator.

- **Generator of unikernel images.** This component is in charge of building unikernel images based on the description of unikernel components in Chapter 4. This description includes the components required to build the service, but also the ones required to protect it (e.g. embedded security mechanisms). It may also be invoked by the cloud orchestrator to address changes that may occur during the operation phase.

- **Cloud orchestrator.** It controls the life cycle of cloud resources in accordance with the TOSCA specification. These resources include more particularly protected unikernel instances that are deployed and managed in the infrastructure. The proposed architecture is compatible with any TOSCA-compatible cloud orchestrator.

As previously mentioned, we start from a SecTOSCA specification, which is successively translated into a UniTOSCA specification, serving to build protected resources, and then a TOSCA specification serving their orchestration.

5.5.2 Interpreting SecTOSCA Specifications

We detail the operation of the SecTOSCA interpreter, responsible for extracting the security requirements from the SecTOSCA specification and enriching the TOSCA topology in order to integrate the security mechanisms enforcing these requirements. We can distinguish two major tasks: (i) the enrichment of the TOSCA topology to support the enforcement of security functions, that can be seen as a policy refinement step enabling the integration of security mechanisms to the topology; and (ii) the provisioning of the security orchestrator with security rules to parametrize these mechanisms during their operation. We can remark that our solution supports also the enforcement of security rules directly on the resources. Further discussions with that respect will be given in the next section dedicated to performance evaluation.

Based on the extensions of the TOSCA language introduced in Section 5.4, the SecTOSCA interpreter is in charge of determining whether security functions can be enforced on a given topology representing a cloud service. For that purpose, it relies on the properties, capabilities and requirements of resources composing this topology. This includes both the TOSCA nodes and their TOSCA relationships. In a more formalized manner, it interprets a SecTOSCA specification, noted $D_{secTOSCA}$, and generates a UniTOSCA specification, noted $D_{uniTOSCA}$ as well

as a policy P_{SO} for the security orchestrator, as given by Equation 5.8.

$$translate : D_{secTOSCA} \longmapsto \langle D_{uniTOSCA}, P_{SO} \rangle \tag{5.8}$$

As access control example, P_{SO} can represent the groups of entities allowed to access each others. This refinement is only possible if the set of security functions, noted $S(D_{secTOSCA})$, described in the secTOSCA specification is enforceable on the secTOSCA topology, noted $L(D_{secTOSCA})$, for a given execution environment e. This supposes that these security functions are supported by the different types of resources of the topology, as described by Equation 5.9.

$$\begin{aligned} &\forall sf \in S(D_{secTOSCA}), \\ &\quad isEnforceable(sf, L(D_{secTOSCA}), e) \equiv \\ &\quad \forall T \in L(D_{secTOSCA}), isSupported(sf, T, e) \end{aligned} \tag{5.9}$$

The type T can stand for both a node type T_{node} or a relationship type $T_{relationship}$. The fact that a given type supports a given security function does not necessarily mean that the type requires to integrate specific security mechanisms. The resulting $D_{uniTOSCA}$ is used to drive the generation of unikernels.

5.5.3 Building and Orchestrating Unikernel Resources

We describe now the role of the uniTOSCA interpreter, in charge of driving the generator of protected unikernel images and of providing a TOSCA-native specification to the cloud orchestrator. The uniTOSCA specification describes the different modules required to build the unikernel images. This also includes modules implementing security mechanisms. The uniTOSCA interpreter therefore takes a uniTOSCA specification, noted $D_{uniTOSCA}$, and produces a TOSCA-native specification, noted D_{TOSCA}, together with the generation policy, noted P_{UG} for building protected unikernel images, as given by Equation 5.10.

$$translate : D_{uniTOSCA} \longmapsto \langle D_{TOSCA}, P_{UG} \rangle \tag{5.10}$$

The generation of a unikernel image relies on a set of k modules $m_1, m_2 ... m_k$, as previously given in Equation 5.5. This image permits to define a TOSCA type T_{node}, which is referred by the TOSCA-native specification D_{TOSCA}. The composition of modules (or unikernel components) to build a TOSCA type is only possible if these modules are consistent amongst them and with the execution environment. This means that all the requirements of modules, including security mechanisms, are satisfied by the capabilities provided by other modules or by the execution environment. In a more formalized manner, we consider the $\oplus$ operator representing the composition of modules and the $\rhd$ operator indicating that the requirements of a module are satisfied by the capabilities of another module. Let consider two modules $m_1 = \langle prop_{m_1}, itf_{m_1}, req_{m_1}, cap_{m_1} \rangle$ and $m_2 = \langle prop_{m_2}, itf_{m_2}, req_{m_2}, cap_{m_2} \rangle$, we can therefore define the $\oplus$ and $\rhd$ operators respectively by Equations 5.11 and 5.12.

$$\begin{aligned} m_1 \oplus m_2 = \langle &prop_{m_1} \cup prop_{m_2}, itf_{m_1} \cup itf_{m_2}, \\ &req_{m_1} \cup req_{m_2}, cap_{m_1} \cup cap_{m_2} \rangle \end{aligned} \tag{5.11}$$

$$m_1 \rhd m_2 \equiv req_{m_1} \subseteq cap_{m_2} \tag{5.12}$$

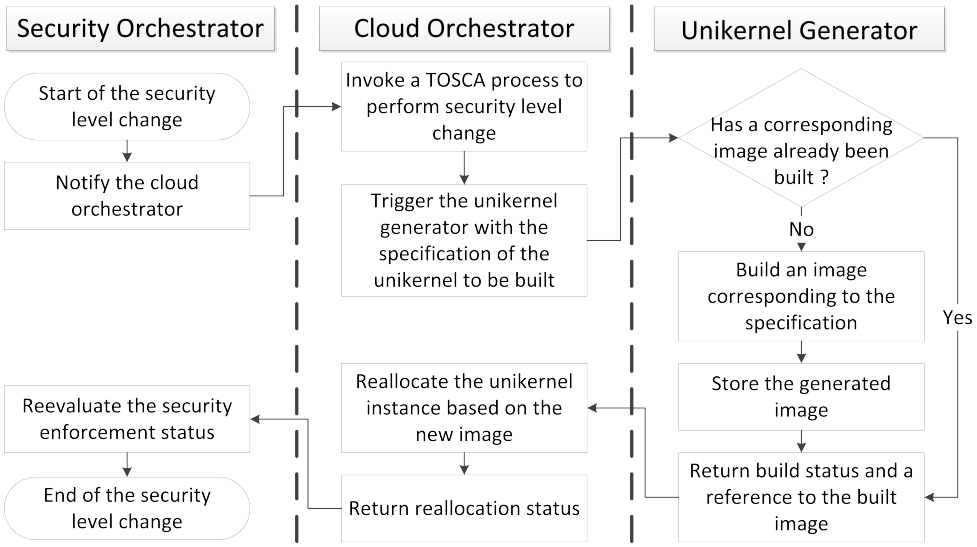


Figure 5.9: Interaction diagram related to a security level change

Let cap_{env} be a set of capabilities provided by the execution environment. The set of modules $m_1...m_k$ are considered as cap_{env}-consistent when all required capabilities are satisfied by other components or by this environment, as given by Equation 5.13.

$$\begin{aligned} &isConsistent_{cap_{env}}(m_1...m_k) \equiv \\ &(\bigoplus_{i=1}^{k} m_i) \rhd (\bigoplus_{i=1}^{k} m_i) \oplus \langle \oslash, cap_{env}, \oslash, \oslash \rangle \end{aligned} \tag{5.13}$$

This cap_{env}-consistency means that the set of modules can be composed and substituted by a TOSCA node type, noted T_{node} whose capability requirements correspond to at most the cap_{env} capabilities, as given by Equation 5.14.

$$\begin{aligned} T_{node} = &\langle prop_{m_1 \oplus .... \oplus m_n}, itf_{m_1 \oplus .... \oplus m_n}, \\ &req_{m_1 \oplus .... \oplus m_n} \setminus cap_{m_1 \oplus .... \oplus m_n}, \\ &cap_{m_1 \oplus .... \oplus m_n} \setminus req_{m_1 \oplus .... \oplus m_n} \rangle \end{aligned} \tag{5.14}$$

This resulting type corresponds to the building of the unikernel images integrating security mechanisms, as previously introduced with the $image_{uni}$ operation. These node types, referred by the TOSCA specification, can then be deployed and orchestrated by the cloud orchestrator.

5.5.4 Adapting to Contextual Changes

Another important aspect concerns the adaptation to contextual changes. The cloud service is subject to changes, such as the allocation/deallocation of resources or their reconfiguration during the operation phase. Security threats and their potentiality may also dynamically evolve over time. From its policy P_{SO}, the security orchestrator is responsible for maintaining a security level, modifying a security level when necessary, and determining changes to be operated on security mechanisms. It is the only component of the architecture having a view on both the service topology and its security configuration through the policy decision points.

In order to maintain or change the security level of the topology implementing a cloud service, the security orchestrator may proceed in two different ways:

- It may adjust the security rules exposed to the policy decision points (PDP), in order to modify the security configuration of resources. This corresponds to the $configure_{sec}$ operation previously defined. The scope of this option is relatively limited in a unikernel context, where we try to minimize the configurability of resources.
- It may regenerate the unikernel-based resources in most cases. This corresponds to scenarios where most of rules may be statically implemented over the resources, integrating an internal PDP. This regeneration enables to modify the parametrization of the resources, but also to insert or remove security mechanisms from unikernel resources. This corresponds to the $building_{sec}$ operation previously defined.

The regeneration of unikernel images is triggered by the security orchestrator through the cloud orchestrator, as depicted on Figure 5.9. Unikernel images corresponding to different security levels may be generated in a proactive manner. Further information regarding the generation of unikernels can be found in Chapter 4, where we detail a generator framework.

5.6 Summary

We have proposed in this chapter a software-defined security approach based on the TOSCA language, in order to support the protection of cloud resources using unikernel techniques. The TOSCA language enables the specification of cloud services and their orchestration. It is extended to drive the integration and configuration of security mechanisms within cloud resources, at the design and operation phases. We rely on unikernel techniques to elaborate cloud resources using a minimal set of libraries and to reduce the attack surface. In that context, we have introduced two extensions of the TOSCA language. The first extension, called UniTOSCA, permits to refine the description level to specify the building of unikernel-based resources. The second extension, called SecTOSCA, permits to define multi-level security requirements. We have illustrated the conversion process of SecTOSCA into UniTOSCA and later TOSCA in Section 5.4.4. We have then designed a framework capable of interpreting this extended language and of generating and configuring protected unikernel virtual machines. In particular, we have described the different components and modeled their interactions with respect to the building of cloud resources embedding security mechanisms and their adaptation to contextual changes. This adaptation can be performed through the reconfiguration of security mechanisms, but also through the regeneration of protected unikernel virtual machines. Unikernel images corresponding to the different security levels can be proactively generated by the security framework. In the next chapter, we will present prototyping work related to our approach, as well as experimental results for quantifying its benefits and limits.

Chapter 6

Prototyping and Evaluation

Contents

6.1 Introduction

In the previous chapters, we have presented different contributions (architecture, unikernel generation, policy specification) related to our approach. This chapter describes prototyping and evaluation work regarding the components of the software-defined security framework. The MirageOS platform provides the unikernel resources to be protected. The security functions to be enforced are focused on access control, by designing authentication and authorization security mechanisms. A generator of protected unikernels is in charge of designing and inserting security mechanisms inside the resources requiring protection. The security orchestrator is provided by the Moon project [106], and is in charge of managing security mechanisms, in accordance with the policy.

The designed framework is in phase with a distributed cloud environment. It supports the generation of the virtualized resources over multiple infrastructures. The multi-tenancy is directly endorsed by the Moon security orchestrator. The implementation prototypes serve as building blocks to our experimentations, in order to evaluate the benefits and limits of our approach. Experiments are performed with a testbed environment featuring both single-node and multiple-node configurations.

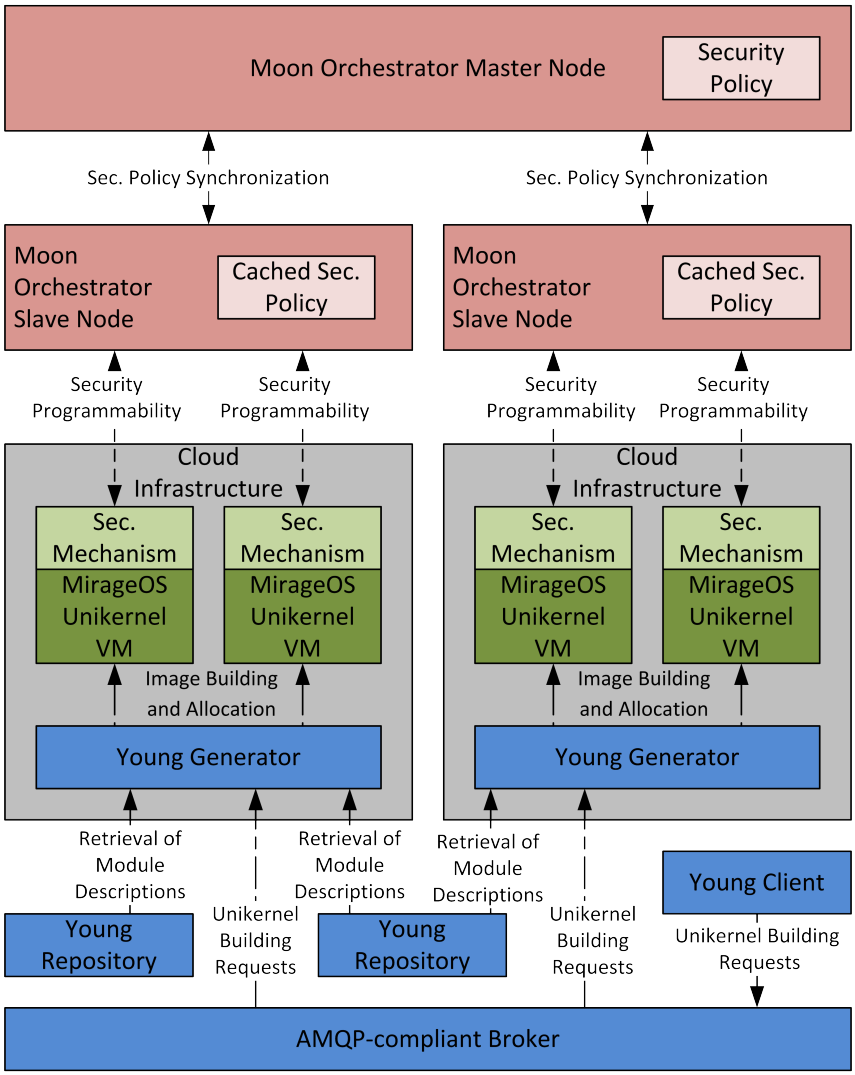


Figure 6.1: Implementation prototypes and their environment at a glance

The remaining of this chapter is organized as follows: Section 6.2 describes the technical implementations of prototypes, including a unikernel generator and security mechanisms, in link with the Moon framework. Experimental results are then detailed and discussed in Section 6.3. Section 6.4 gives a summary of this work.

6.2 Implementation Prototypes

The implementation work consists in the elaboration of prototypes (unikernel generator and security mechanisms) and the integration with a security orchestrator. The Young unikernel generator has been elaborated to generate and allocate constrained unikernel images. The Moon framework provides a support for policy-based security orchestration on allocated unikernel images. Two different security mechanisms for protecting unikernels have been developed and integrated with the Moon framework. Figure 6.1 describes how these prototyped components are interacting to ensure the protection of cloud resources in one tenant, over multiple infrastructures.

6.2.1 Young Unikernel Generator

The YOung UNikernel Generator (Young) drives the production of protected unikernel images in a distributed environment. It receives specifications describing images to be produced, retrieves the necessary technical informations to build them, and store them in a registry. It handles distributed environments, by receiving image specifications and collaborating with other instances over a distributed broker. Such a collaboration permits to signal about images being, or having already been built, and to alert regarding the technical inability to support a specification. Although it only supports MirageOS in its current form, it can be extended to support other unikernel platforms with limited modifications. The prototype is of around 2.9 KLoC. This framework is based on four components.

- First, one or several repositories is/are in charge of hosting the technical description of modules that can be integrated into the unikernel images. Our implementation prototype of these components is developed in Python, using the flask framework. The content of the repository is statically specified in the application, and is accessed through HTTP requests.

- Second, a client emits requests for the unikernel images to be generated. Our prototype takes a unikernel image specification as input at start up, and emits a corresponding request to all the available generators on the network. This client is an application written in Java, with messaging and logging capabilities similar to the generators.

- Third, a generator can be allocated multiple times, in each infrastructure, and can be configured with a specific set of repositories. This design choice is motivated by the multi-cloud support: each generator instance can cope with the technical description of the sole module supported by the infrastructure, preventing the building of unikernel images on an infrastructure unable to operate it. This application is written with the Java programming language.

- Finally, the AMQP-compliant broker is in charge of supporting the interactions between the client and the generator instances in a multi-cloud context. Our prototype exploits the RabbitMQ [130] message broker.

The design of this prototype is driven by both the framework for generating protected unikernels, and the cloud environment. The generator is able to get instructions from a client, to determine the dependencies and source codes of unikernels, to build unikernels, and to store them.

The different components collaborate all along the operation of the framework. The client emits over the message broker the request for the unikernel image to be build to the listening generators, but does not wait for feedback from them. The generators listen on the message broker for incoming unikernel building requests. When a generator is not able to cope with the building of an image, it notifies other generators of its inability to handle the request. When the image is already prepared, it notifies all the other generators, so they do not have to proceed with the building of an image. The client simply obtains the already generated image. When the generator starts and finishes the image generation, it also notifies other generators to prevent them from starting a similar job during the ongoing image building process. The generator can also receive requests for importing and exporting produced images from a persistent storage.

The requests that are received by the generator are composed of three fields. The `request-name` field identifies the unikernel to be processed. The `action` field selects the processing to apply (e.g. "`build`","`building`" and "`already-prepared-image`"). The `arg` field can supply parameters for indicating actions requiring additional arguments. Figure 6.2 details an

example of a request for building a unikernel image labeled `protectedHttpd`. It integrates four unikernel modules: `Dispatch`, `AccessControlModule`, `AccessControlModuleDispatchHook` and `AccessControlModuleConfig`.

```
{
  "action":"build",
  "request-name":"protectedHttpd",
  "arg":"[
    {
      \"module-name\":\"Dispatch\",
      \"module-configuration\" : []
    },{
      \"module-name\":\"AccessControlModule\",
      \"module-configuration\" : []
    },{\"module-name\":\"AccessControlModuleDispatchHook\",
      \"module-configuration\" : []
    },{\"module-name\":\"AccessControlModuleConfig\",
      \"module-configuration\" : []
    }
  ]"
}
```

Figure 6.2: Example of a request to build a unikernel image

In the repository, the technical description of available modules is organized as follows. Each repository is a set of technical descriptions of modules. Each of them is characterize by (i) a name to identify it, (ii) a method to obtain its implementation, (iii) the remote location of this element, (iv) a method to import it locally, (v) the location where to insert it and (vi) several available configuration options. The latter is a list of a technical configurations reflecting how the module can be configured. Each configuration option contains (i) a name to identify it relatively to a module, (ii) a method indicating how the configuration is proceeded technically, (iii) an argument to identify technical resources involved in the configuration process, (iv) the number of arguments expected from the specification, and (v) the default options to be set if no option argument are set. All the methods are not necessarily supported by all the generators, considering multiple unikernel platforms. Figure 6.3 gives an example of a unikernel module description.

The image generation processing is composed of several steps:

1. The generator receives a valid unikernel build request.

2. If an already built image is found, it is notified to other generator instances and the job is aborted.

3. If an similar on-going job is found, it is notified to other generator instances and the job is aborted.

4. A backoff delay elapses to enable other generator instances to terminate the current job, if other corresponding images or jobs are found.

```
[{
        "name": "AccessControlModule",
        "origin-strategy": "http",
        "origin-location": "http://lightning.lan:8888/sec/
           accesscontroldynamic.ml",
        "destination-strategy": "download",
        "destination-location": "./accesscontroldynamic.ml",
        "configuration-points": [{
                "configuration-name": "security_monitor_url",
                "strategy": "instanceoption",
                "content": "--moon_url=",
                "arguments-nmb": 1,
                "default": "https://moon_url/"
        }, {
                "configuration-name": "security_domaine",
                "strategy": "instanceoption",
                "content": "--moon_domain=",
                "arguments-nmb": 1,
                "default": "default_moon_domain"
        }, {
                "configuration-name": "pull_security_mode",
                "strategy": "instanceoption",
                "content": "--pull_security_mode=",
                "arguments-nmb": 1,
                "default": "true"
        }]
}, {
        "name": "AccessControlModuleDispatchHook",
        "origin-strategy": "http",
        "origin-location": "http://lightning.lan:8888/sec/
           dispatch.ml",
        "destination-strategy": "download",
        "destination-location": "./dispatch.ml",
        "configuration-points": []
}, {
        "name": "AccessControlModuleConfig",
        "origin-strategy": "http",
        "origin-location": "http://lightning.lan:8888/sec/config
           .ml",
        "destination-strategy": "download",
        "destination-location": "./config.ml",
        "configuration-points": []
}]
```

Figure 6.3: Example of a unikernel module description

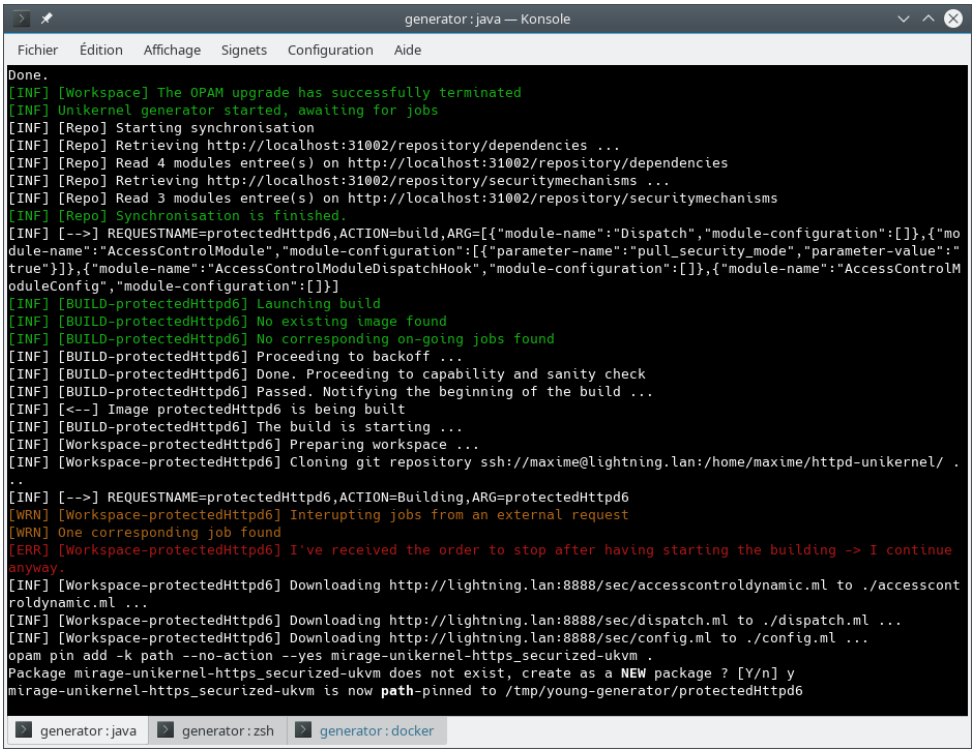


Figure 6.4: A screenshot of the unikernel generator starting a build job

5. The capability of the generator to handle the request is checked. In case of a negative result, a notification is emitted and the job is aborted.
6. The build job is notified to other generators.
7. The build job is processed.
   (a) A workspace is initialized.
   (b) The build request is unwrapped to retrieve module technical implementations and configure them accordingly.
   (c) The image is built and linked.
   (d) The build is closed.
8. The notification of the build success is notified to other generators.

Any build process can be interrupted by external notifications, before the backoff delay is elapsed. These steps are implemented to support multiple implementations. Figure 6.4 shows different steps during the generation of a unikernel image.

6.2.2 Moon Framework

The design and the implementation of the Moon framework is not part of the work of this thesis. The components are taken off-the-shelf as an appliance developed by the research projects *trusted cloud* and *security management* from Orange Labs. The source code is available at [106].

Moon integrates the OpenStack tool-suite to cope with the keystone authorization and authentication appliance. The keystone decision process is delegated to a Moon instance and its

policy engine. The framework is internally defined around two different planes. The *usage plane* supports user-centric policies while *control plane* hosts the security policy affecting a whole cloud. As stated in [119], the decoupling between these two planes permits each user to elaborate their own policies with respect to the global one, enabling the support for multi-tenant cloud. Moreover, Moon supports instances on multiple infrastructures thanks to a slave management feature, as described in [27]. It can be allocated as a master instance or a slave instance. A master node is in charge of managing the data of the security policy and propagating policy updates to slave nodes. A slave node processes the incoming decision requests from keystone, fetches only required elements from the master, and acts as cache for policies in case of connectivity loss with the master. This feature makes Moon compatible with the security policy management in multi-cloud environments.

In the perspective of the SDSec framework of this thesis, this feature makes Moon to provide the *Security Orchestrator* entity and contribute to the *Policy Decision Point* policy. The master node contains the materials for building and hosting cloud-wide security policies, similarly to the *Global Security Policy (GSP)*. Slaves nodes cope with incoming decision requests and have a dedicated security policy in cache and aligned with their enforcement perimeter, similarly to the *Tenant Level Security Policy (TLSP)*. Currently, the Moon framework is affected by several shortcomings affecting its compliance with the architecture. First, the implemented security engine is limited to access control, thus limiting addressable security functions to authorization and authentication. Moreover, Moon slave instances are only able to respond to incoming requests and not to proactively configure protected components to conform to a security statement, as expected from the *security statement protocol*, described in section 3.4.4. Finally, in case of security policy updates, the master node pushes the modifications to slave nodes, contravening to the *security policy discovery protocol*.

6.2.3 HTTP Authentication and Authorization for MirageOS Unikernels

We have also developed an HTTP authentication and authorization mechanism for MirageOS unikernels. The prototyping relies on (i) a *policy decision point (PDP)* taking local security decisions in phase with the *security orchestrator (SO)*, (ii) a *policy enforcement point (PEP)* enforcing these decisions on a unikernel resource and (iii) a generator of constrained unikernel images containing the corresponding PEP. The PDP takes as input the user (subject) requesting an access to a resource, the resource being accessed (object), and the action describing the modalities of this access. In accordance with the security orchestrator policy, the PDP will grant or deny the access to the resource. The PEP is implemented on the unikernel resource as a hook into the access routine, in order to prevent any forbidden access to the resource.

Technically, the considered resource is a secured HTTP server over a MirageOS unikernel [91], which was extended to implement an access control mechanism including authorization and authentication. The MirageOS platform is based on the strongly-typed OCaml programming language. It does not propose ready-to-use applications but provides an SDK to design them. We therefore considered the web server from the mirage-skeleton repository [97], which relies on the CoHTTP library [33]. This module is integrated to the generated unikernel images. Its access verification feature is invoked through a hook inserted in the web server, in the resource retrieval routine: while a granted access does not impact the resource retrieval, a negative decision results in a 403-type HTTP code response to the client. The instantiation of these unikernels is supervised by a uKVM [160] monitor running over a KVM hypervisor. This permits to generate a specific VM monitor for each unikernel, whose unused features can be proactively disabled.

Three different approaches have been implemented for the PDP: (i) an internal PDP, (ii) a

pushing PDP and (iii) a pulling PDP. The first approach integrates the decision process into the unikernel. The policy knowledge together with the PDP is integrated to the image during its unikernel generation, and no interaction with any external agent is required during the lifetime of the unikernel virtual machines. The second approach considers an external PDP agent interacting with the PEP according to a push communication model. The third approach is based on an external PDP agent interacting with the PEP according to a pull communication model. Technically speaking, the configuration of the security mechanism with the internal configuration is performed before the image building process, through the manual provisioning of the `authorized_buffer` and `forbidden_buffer` variables in the source code of the security mechanism. Figure 6.5 showcases an example of the internal configuration of this module. To enable the push communication model, the unikernel image needs to be launched with the `pull_security_mode` boot argument set to false. In this state, the unikernel opens up to the port `38001` to receive configuration directives. As the Moon security orchestrator does not support yet this model, we have sketched up a plain text protocol. The pull model relies on a permanent connection between the unikernel and the Moon interface to get the corresponding access rights in an demand manner. The Moon orchestrator provides the corresponding access decision. These results are both located in the content of the response and its header.

```
1  let buffer_authorized:(authorization list ref) = ref
2          [
3            ({resource = "%2Fmyfile.html"}, [
4              {login = "mylogin"; password = "mypassword"};
5              ])
6          ]
7
8        let buffer_forbidden:(authorization list ref) = ref
9          [
10           ({resource = "%2myfile2.html"}, [
11             {login = "mylogin2"; password = "mypassword2"}
12             ])
13         ]
```

Figure 6.5: Configuration example of the HTTP authentication and authorization mechanism

6.2.4 Application Firewalling for Mirage OS Unikernels

As an extension of the previously presented mechanism, we have developed an application firewall for Mirage OS web server. This security mechanism enforces an access control on incoming HTTP requests to accept or refuse their processing. The core data model remains the same than the programmable HTTP authentication: An identification element is stored inside a cookie in the client browser. It is involved in the process of access control decision taking. The action is delegated to the HTTP method used by the browser to access a resource. The supported configuration models remain strictly the same, with similar operational modalities of configuration. Therefore, this module is again designed to cope with the Moon orchestrator. Figure 6.6 describes a configuration example for this mechanism.

```
1  (*Policy specification - Data structure to convert to a heap to
       implement cache or pushing*)
2      let buffer_authorized:(authorization list ref) = ref
3        [
4          ({resource = "%2Findex.html"}, [
5            {subject = "mylogin"; action = "GET"};
6            ])
7        ]
8
9      let buffer_forbidden:(authorization list ref) = ref
10       [
11         ({resource = "%2Fmyfile2.html"}, [
12           {subject = "mylogin2"; action = "GET"}
13           ])
14       ]
```

Figure 6.6: Configuration example of the application firewalling mechanism

6.3 Evaluation Scenarios

In this section, we evaluate the performances of the proposed solution with respect to protected unikernels that are generated. We exploit the security mechanisms previously mentioned to evaluate our framework through extensive series of experimentations. A particular interest has been given, within this evaluation, to the PDPs and PEPs, which constitute key components to this integration, through the interactions between the security control plane and the security resource plane.

6.3.1 Experimental testbed

The experiments have been performed on the following testbed. The host system features an Intel Xeon E5-1620 CPU at 8x3.6 GHz with 8 GB of RAM. It executes an up-to-date version of the Ubuntu 16.04 LTS distribution with the linux kernel 4.4.0-112. Unikernel images are built with MirageOS 3.0.8 development kit. The virtual machines are instantiated with the uKVM monitor 0.2.2-1. They are equipped with 1 vCPU and 512 MB of RAM, and the latest version of available modules from their distribution. The PDP interacts with the security orchestrator based on the `moon_bouchon` interface (version 2018-01-30) from the Moon project. This interface permits to emulate the behavior of the Moon orchestrator and the interactions with the PDP to support access control. The connectivity amongst unikernels is based on a virtual network built with OpenVSwitch 2.5.2 [115]. The performance evaluation has been done using the ApacheBench [2] framework and the `time` standard tool.

6.3.2 Performance of the three approaches

In a first series of experiments, we wanted to evaluate the performance of the three approaches individually, and in particular quantify the overhead induced by the outsourcing of the PDP (pull or push approaches) from the protected unikernel virtual machines. In our TOSCA-based security framework, the PDP outsourcing is a decision which is taken by the SecTOSCA interpreter, for a

given security mechanism and according to a given service topology to be protected. The security requirements are transmitted by the SecTOSCA interpreter to the security orchestrator through the security policy. In the case of an internal PDP, the security requirements can be directly integrated to the unikernel images, through dedicated modules, enforcing a decision process internal to the protected unikernels. In that context, we have analyzed different parameters related to both the building and the operation of protected resources.

Considered Approach	**Image Size (MBio)**
Internal PDP	7.5
Pull PDP	8.1
Push PDP	8.1

Table 6.1: Comparison of protected unikernel image sizes

We first quantified the size of generated unikernel images implementing the three approaches previously presented (internal, pull-based and push-based). Table 6.1 gives a comparison of the size of these images. We expected the size of the image corresponding to an internal approach to be higher than the two other external approaches. In fact, it appears that the size of the internal approach is of 7.5 MBio, with the size of the external approaches reaches 8.1 MBio. This overhead of 600 KBio for the external approaches is due to the additional modules required to support the interactions between the PDP and the PEP components of our solution.

We also evaluated the time required for generating protected unikernels from the source code, with these different approaches. The obtained results are detailed on Figure 6.7 for the internal approach (*PDP-Int*), the push-based external approach (*Push-PDP*), and the pull-based external approach (*Pull-PDP*). We observed a generation time of around 4.1 seconds with the first approach, while it reaches 4.45 and 4.47 seconds with respectively the push-based and pull-based approaches. The overhead induced by the external approaches is again observed here, and can be correlated with the sizes of protected unikernel images.

We were also interested in quantifying the resource consumption of a protected unikernel virtual machine based on these different images. A particular focus has been given to the network performance presented on Figure 6.8 and to the memory consumption given on Figure 6.9. In both cases, we are varying the number of active connections from 0 to 1000 connections during experiments. We can observe on the first figure that the push-based and pull-based approaches introduce an overhead of respectively 41.3% and 1.2% in average, in comparison to the internal approach. The number of requests sent to the protected unikernel virtual machines is supposed to induce the same number of responses from them. The differences at high load are due to the congestion of the machine. A similar phenomenon is observable on the second figure, where the memory consumption related to the push-based and pull-based approaches generate an overhead of 32.7% and 1.2%, in comparison to the internal approach. For instance with 500 active connections, we obtain a resource consumption of 38 KB with the internal scenario, against 39 KB and 51 KB with respectively the push-based and pull-based scenarios. The overhead percentages amongst the different approaches are relatively stable, while varying the number of active connections.

Finally, we compared the average time required for processing HTTP requests, including the authentication and authorization processing, with the three different approaches. The results are described on Figure 6.10, where we considered again the same range for the number of active connections. The different approaches produce experimental results that are quite similar,

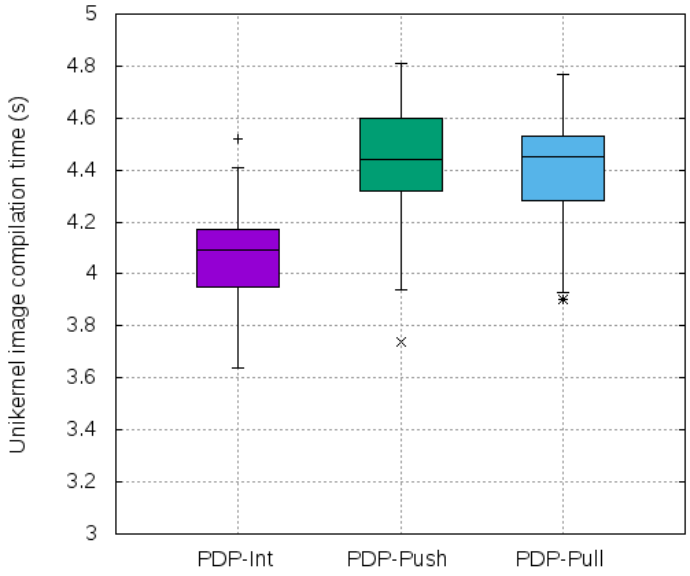


Figure 6.7: Generation time of protected unikernel images

with an average authenticated HTTP processing time of 11 ms. Several peaks can be observed with the pull-based approach, in particular with a number of connections between 600 and 1000 connections. This can be due to the external PDP which becomes a bottleneck with respect to authenticated requests, in such a pull-based scenario.

6.3.3 Performance with a pool of protected unikernels

In a second series of experiments, we wanted to evaluate the performance of our solution with a pool of protected unikernels. We consider a pool that may be composed of both unikernels implementing the pull-based approach and unikernels implementing the push-based approach. In our security framework, the deployment of these unikernels is performed by the cloud orchestrator, while the selection of unikernel types (pull-based or push-based) is under the charge of the SecTOSCA interpreter. We performed the same experiments than previously to quantify the memory consumption, the networking performance, and the authenticated HTTP request processing time with such a pool of 100 protected unikernels. During experiments, we varied the ratio of unikernel virtual machines implementing each of the two approaches, and considered a workload from 1 to 1000 incoming concurrent connections.

We can observe on Figure 6.11 the results obtained with this pool with respect to the memory consumption. As we expected, the memory consumption increases when the incoming workload is growing. In particular, we notice that a higher proportion of push-based unikernel virtual machines can minimize the memory consumption by 22.0% on average during experiments. These results can be explained by the requirement for unikernel virtual machines based on the pull-based approach to include further features in their networking stack (e.g. DNS resolver, HTTP client library) compared to the push-based approach. The same observation can be done for the networking performance, as shown on Figure 6.12. A higher ratio of push-based unikernel virtual machines improves the performances. However, the impact might be less significant, when considering an enhanced pull-based approach integrating caching facilities.

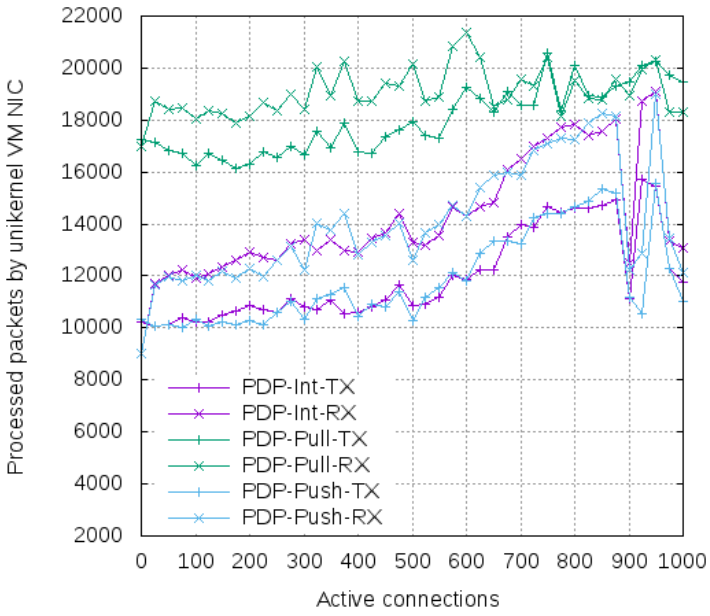


Figure 6.8: Network performance with the different approaches

We also evaluated the performance on authenticated HTTP processing with the different workload scenarios. The experimental results are detailed on Figure 6.13, where we plotted the cumulated processing time. These results clearly show that increasing the number of push-based unikernels decreases such a processing time for all the considered scenarios. Most of the time, the processing time is growing with the number of concurrent active connections. Except for the scenarios with a ratio of more than 90% of push-based unikernels, the one connection workload induces a significantly longer processing time than any other workload scenarios. These results may be due to the DNS resolution time, which takes a more important part in the overall request processing.

6.3.4 Security policy propagation and enforcement

In a last series of experiments, we were interested in evaluating the time required for the propagation and enforcement of a security policy with the different approaches. This also concerns the updates of the security policy. In our security framework, such a policy update typically occurs during the operation phase and is triggered by the security orchestrator. Chapter 4 has already shown the costs induced by the regeneration of unikernel images. We focus in this series of experiments on the reconfiguration of security mechanisms integrated into the protected unikernel virtual machines. This reconfiguration requires the regeneration of the unikernel images in the case of the internal approach. The scenario is based on unikernel virtual machines, whose the security policy is updated through the `moon_bouchon` interface provided by the Moon security orchestrator. The security policy corresponds to an access control list. We considered the worst case scenario, where this list may be subject to changes at any resource access request. The size of the list is varying from 0 to 40,000 rules during the experiments. We quantified the delay time between the update of the security policy specification and its enforcement on the cloud resource. The effective enforcement was evaluated from the perspective of a client, through the sending of authenticated HTTP requests to access resources, and the evaluation of observed access rights.

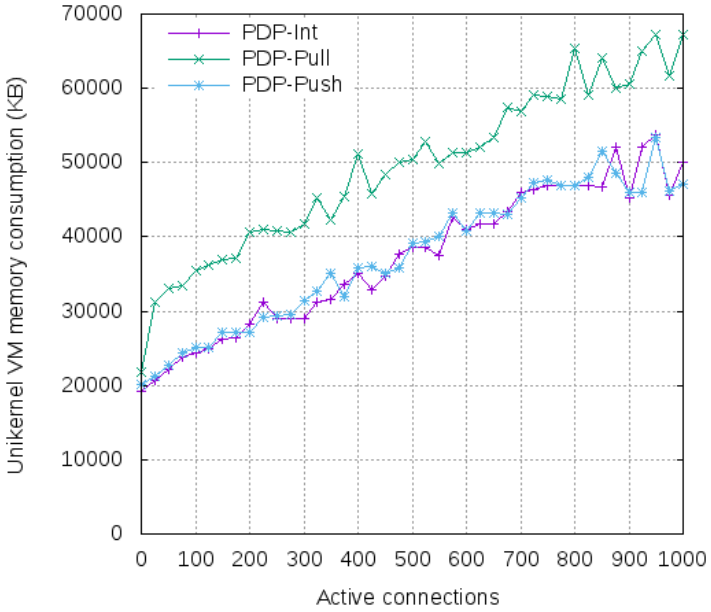


Figure 6.9: Memory consumption with the different approaches

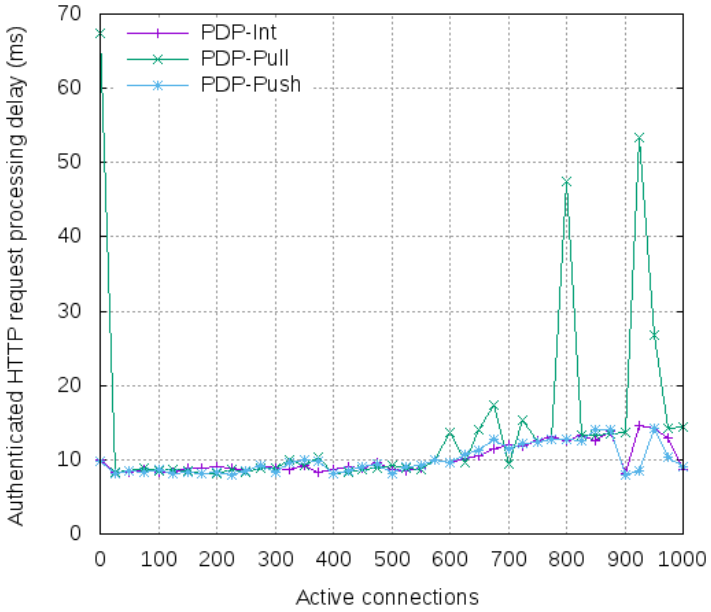


Figure 6.10: Authenticated HTTP processing time with the different approaches

The results are detailed on Figure 6.14 showing the delay time obtained with the pull-based and push-based approaches. We can observe a behavior close to linearity with respect to the size of the access control list. We expected that the delay times obtained with the push-based approach will be better than the ones induced by the pull-based approach, which was the case. The push-based approach leads to a shorter delay time, in comparison to the pull-based approach showing an overhead of 48,6% on average. The delay times obtained with the internal approach are sensitively higher than the two other approaches, and are more distributed. The overhead

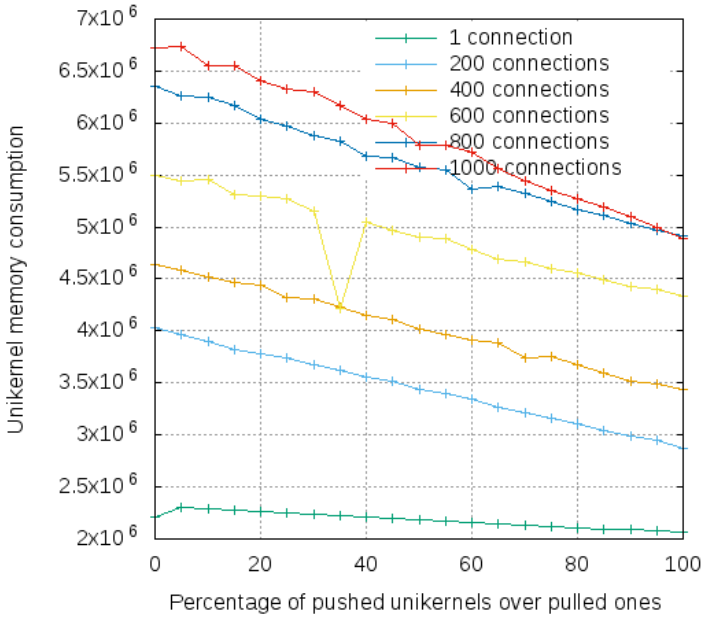


Figure 6.11: Memory consumption with a pool of protected unikernels

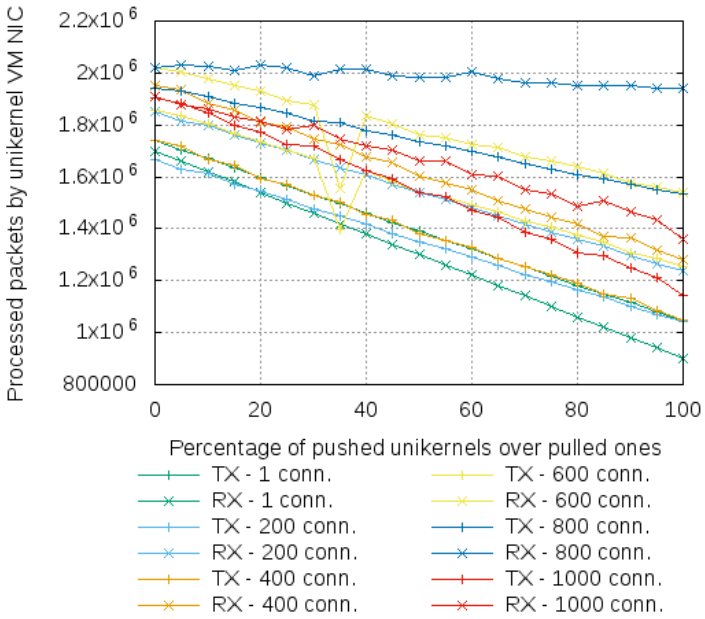


Figure 6.12: Network performance with a pool of protected unikernels

induced by the internal approach is 5.82 times higher due to the compilation time. The update of the security policy requires to regenerate the unikernel images implementing the internal approach. Proactive strategies permit to anticipate changes in the security policy and to re-generate unikernel images at an early stage. In the mean time, these experiments permitted to quantify the scalability of the three different approaches with respect to the size of the access control list. While it generates an additional network traffic, the pull-based approach operates successfully for the different experimental scenarios from 0 to 50,000 security rules. The push-based

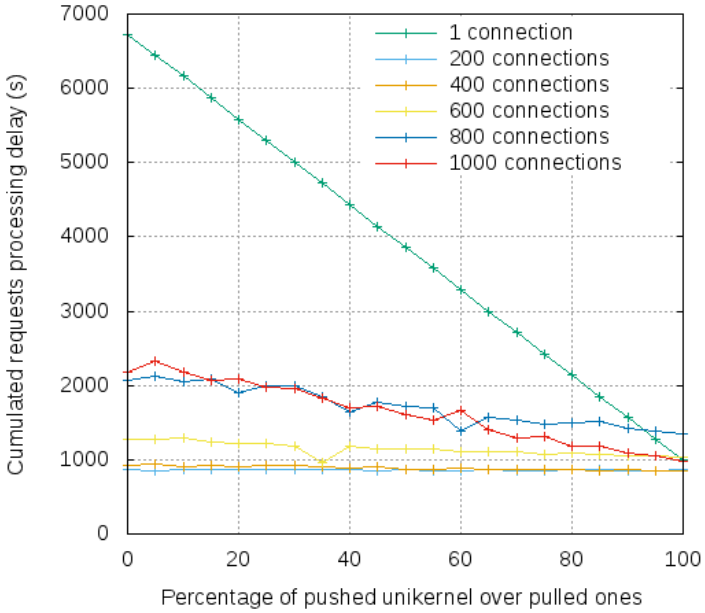


Figure 6.13: Cumulative HTTP processing time with authentication based on a pool of protected unikernels

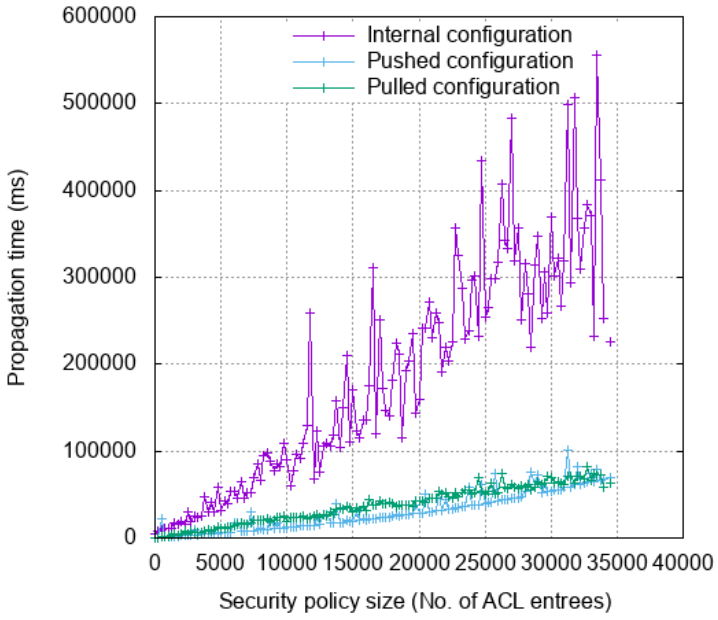


Figure 6.14: Time performance for the propagation and enforcement of a security policy based on the different approaches

approach was capable of supporting up to 33,500 rules with a single unikernel, due to memory depletion, while the internal approach supported up to 21,750 rules with a single unikernel, due to unikernel compilation. This corresponds to a high number of security rules with respect a single unikernel scenario. These experiments have shown the compared performances of three different approaches to implement our TOSCA-oriented security framework.

6.4 Summary

In this chapter, we have detailed the prototyping and evaluation of our security software-defined approach based on the generation of protected unikernels. In particular, we have developed the Young unikernel generator, as well as two unikernel security mechanisms (HTTP authentication and authorization, and application firewall). The proposed solution is compatible with the Moon security orchestrator. In that context, we have detailed the different steps corresponding to the generation of protected resources. Our prototyping has served as a support for evaluating the performances of our solution based on extensive series of experiments. In particular, we have quantified the benefits and limits of three approaches of configuration for the protected resources. We have also considered the scenario of a pool of protected unikernel virtual machines, with various performance metrics, including memory consumption, network performance and request processing time. Finally, we have quantified the delay time from the specification of a change in the security policy to its enforcement on unikernel resources.

Chapter 7

Conclusions

Contents

This chapter concludes the thesis manuscript. We give a summary of the main contributions related to our software-defined security approach for distributed clouds. We then point out several research perspectives related to these works.

7.1 Summary of Contributions

The large-scale deployment of distributed clouds has clearly increased the complexity of security management for supporting cloud services. In particular, it introduces new constraints to be taken into account, with respect to the multi-cloud and multi-tenancy properties of these infrastructures. This concerns the requirements of coping with the technical specificities and the dynamism of cloud resources that have to be protected, while ensuring the completeness of the security perimeter. The diversity of protection mechanisms also supposes a flexible and adaptive security management. This thesis addresses the security of distributed clouds, by proposing a unified and homogeneous security management plane able to protect cloud services in that context. It relies on the generation of protected unikernel resources, on the programmability of security mechanisms using the software-defined paradigm, and on the specification of security policies based on the TOSCA cloud orchestration language. The thesis approach, as well as the related contributions, are highlighted on Figure 7.1.

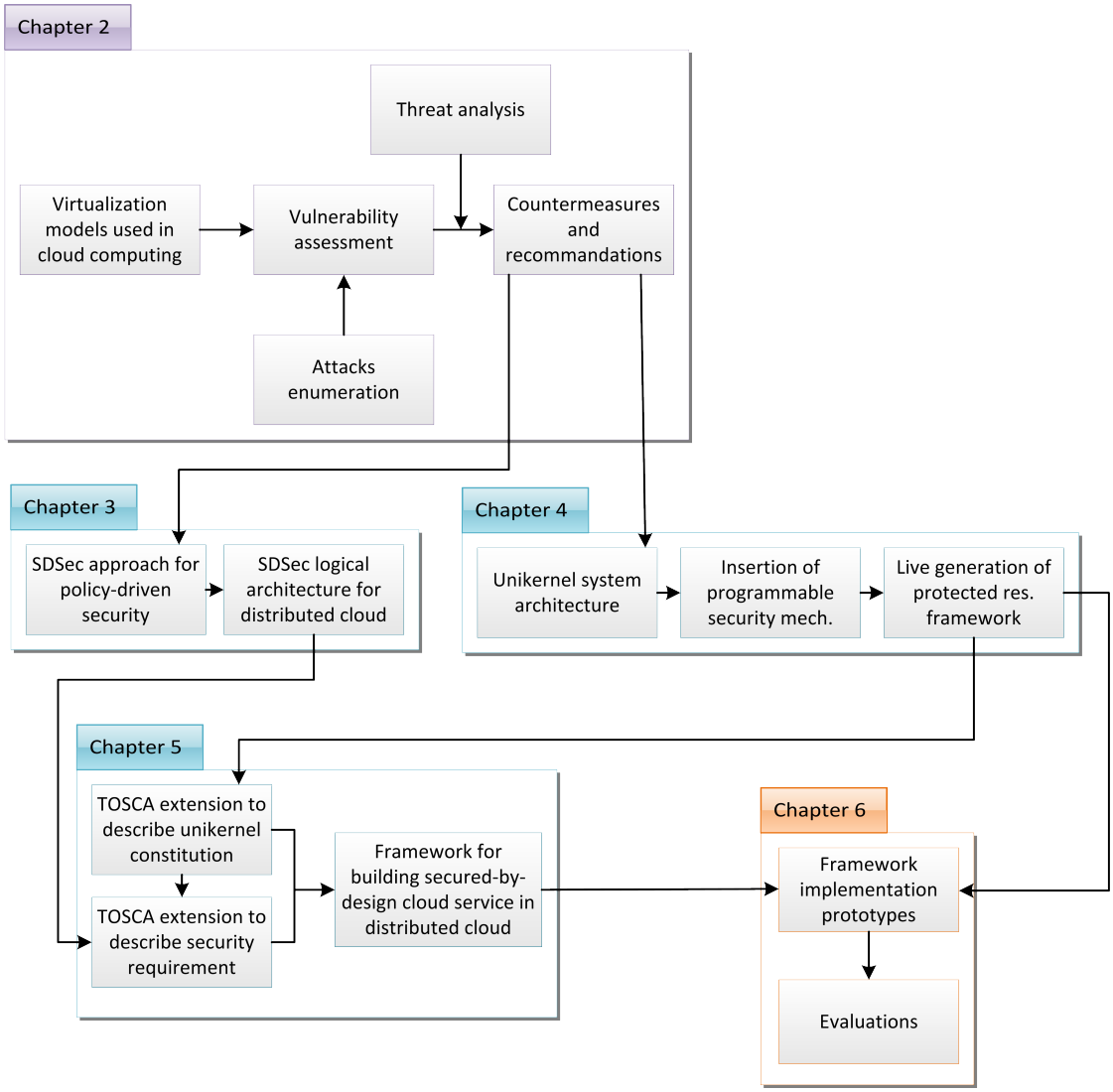


Figure 7.1: Synthesis of the thesis approach and related contributions

7.1.1 Analyzing Virtualization Models for Cloud Security

We have first conducted a comparative analysis of virtualization models, where we described their properties with respect to cloud environments to be protected. The considered models include virtualization based on type-I hypervisors, virtualization based on type-II hypervisors, OS-level virtualization and unikernel-based virtualization. We have identified several threats affecting these virtualization models, and analyzed their criticality with regard to each virtualization model. We have also evaluated their impacts by determining security attacks leveraged by them. We have determined several countermeasures and recommendations serving as raw material to frame our security management approach. In particular, we have shown that the unikernel-based virtualization provides interesting security properties with respect to cloud resources. It permits to restrict the attack surface, while generating light-weight virtual machines. The security requirements should be taken into account at an early stage, as soon as the building of unikernel images, in order to support the full integration of security mechanisms. We have also mentioned the benefits of security programmability for protecting resources in distributed cloud

infrastructures. This programmability can drive both the simple configuration, or the whole rebuilding of the cloud resources based on unikernels.

7.1.2 Designing a Software-defined Security Architecture

In order to support security management in distributed clouds, we have designed a software-defined security strategy for configuring security mechanisms in a multi-cloud and multi-tenant context. It relies on an architecture composed of different abstraction levels. The first one supports security management at the scale of a distributed cloud, with the involvement of a security orchestrator. The second one supports a security decision process at the scale of a tenant. It addresses the specificities related to the local context of each tenant. The third one is composed of cloud resources to be protected, that expose interfaces to be configured accordingly. We have detailed the different components of this architecture, as well as their interactions. We have evaluated the adequacy of this architecture with a set of realistic scenarios. While being aligned with the recommendations issued from the comparison of virtualization models, this architecture brings the gap between the generation of protected unikernel resources, and the specification of policies using the TOSCA cloud orchestration language.

7.1.3 Generating Protected Unikernel Resources on The Fly

We have then proposed a framework for generating protected unikernel resources. The purpose is to build specific unikernel images that include security mechanisms and cope with security requirements. These security mechanisms are programmable, matching with the proposed software-defined security architecture. The building of unikernel images relies on both the configuration of internal components and the integration of programmable security mechanisms. The generation of resources can be performed in an on-the-fly manner, through the deployment of image instances (virtual machines) that have a limited lifespan. Contextual changes can lead to reconfiguring the instances, or may induce the building of new unikernel images. Some instances may also be pooled in a proactive manner, in phase with the rapid elasticity of cloud infrastructures. We have described the modeling of unikernel images, as well as algorithms supporting the generation process. The benefits of this unikernel-based generation include the sanitization of images at building time, the reduction of the attack surface of instances based on these images, and more generally the code portability in line with multi-cloud and multi-tenant deployments. The image generation process has been prototyped and evaluated through extensive sets of experiments, in the context of statically-configured security mechanisms for authentication and authorization over unikernel webservers using MirageOS.

7.1.4 Extending the TOSCA Cloud Orchestration Language

We have extended a cloud orchestration language to drive the generation and configuration of cloud resources using the software-defined security architecture. This language specifies a cloud service, as a topology of resources that can be spread over several cloud infrastructures. We have considered two extensions for this language. A first one, called UniTOSCA, permits to describe the internal modules of unikernel images, ans supports the generation of protected unikernel resources. The second one, called SecTOSCA, addresses the specification of security constraints, according to different security levels. These specifications are used to build unikernel resources and to configure security mechanisms related to cloud services. The orchestration language goes from the design of cloud resources to their deployment and operation, contributing to the concept of secured-by-design cloud resources.

7.1.5 Prototyping and Evaluating the Solution

We have complemented the prototyping and evaluation of our solution, through different implementations. In particular, we have considered the case of (i) a dynamically-configured authorization and authentication mechanism for MirageOS webservers, (ii) an applicative firewall for MirageOS webservers, as well as (iii) a dedicated unikernel image generator for distributed cloud environments. This prototyping has served as a support to evaluate the proposed solution in a quantitative and qualitative manner. We have analyzed the adequacy with the security orchestrator part of the Moon framework developed at Orange Labs. We have also considered different configuration strategies to parameterize security mechanisms, and have evaluated their impact on the performances of unikernel resources (compilation time, memory consumption, networking) that are protected. We have also investigated the benefits and limits of pooling unikernel virtual machines, that can be deployed on cloud infrastructures pro-actively. We have finally quantified the enforcement delay of security policies over unikernel resources, according to different configuration strategies.

7.2 Discussions

This work has focused on proposing a software-defined security approach for distributed clouds, that combines a security management architecture with an on-the-fly generation of protected unikernel resources, and an extended orchestration language. We have also identified several limitations with respect to this approach in its current form. First, the generation of protected resources is restricted to unikernel resources. The properties of these resources facilitate the enforcement of security policies and permit to reduce the attack surface. It also redefine the configuration approach through the rebuilding of resources. We have not considered in the scope of our work scenarios that combine unikernel resources, with other virtualization models (such as type-I virtualization), while our multi-tenant multi-cloud architecture is sufficiently generic to support these cases. Second, the interactions amongst PEPs and PDPs could be more complex. We have considered that a given PEP is under the authority of a single PDP. Scenarios where PEPs (and the corresponding security mechanisms) could be configured by several PDPs might also be interesting to be investigated. This introduces potential redundancy and propriety mechanisms, with respect to enforcement perimeters. Finally, we considered in our work that cloud resources composing the cloud service to be protected, are available all along the operation of this service. These resources (including PDPs and PEPs) may be subject to faults and failures that should be taken into account by the security orchestrator (in addition to the cloud orchestrator).

7.3 Research Perspectives

The contributions developed during this thesis, related to our software-defined security approach for distributed clouds, has opened up new research perspectives.

7.3.1 Exploiting Infrastructure-As-Code for Security Programmability

Our solution relies on the generation and instantiation of unikernel images to enforce security policies. This choice is motivated by their simplified architecture contributing to a low attack surface, and being in phase with an on-the-fly generation of resources to be protected. However, this leaves aside the other resources composing cloud infrastructures nowadays. Our approach

is extensible to other categories of resources. In particular, the infrastructure-as-code [8] trend might contribute to cover additional resources, and to support the integration of a large variety of security mechanisms. In addition, the short life-cycle of OS-level virtualization resources make them also interesting candidates with respect to on-the-fly generation. However, this requires to be able to control the codebase of containers to be protected.

7.3.2 Supporting the Security of IoT Devices

The Internet-of-Things (IoT) [166] permits to interconnect physical devices (vehicles, home appliances, sensors) to the network, in order to build new services. It may also exploit and interact with resources coming from cloud infrastructures. The physical nature of resources, as well as the criticality of collected data, exposes them to security attacks. The work developed in this thesis could be extended to cover the protection of IoT devices. The properties of unikernels seem to be in phase with the limited resources (energy, cpu, bandwith) of IoT devices. However, the costs of programmability and reallocation processes of unikernel resources have to be evaluated over these constrained environments. It is also important to quantify the effects of low power and lossy networks on the security enforcement. Security management algorithms should be adapted to address these limitations.

7.3.3 Checking the Consistency of Security Policies

Software-defined security permits to decouple the control of security mechanisms from the cloud resources that integrate these mechanisms. The TOSCA cloud orchestration language has been extended to specify security constraints according to various levels of security. These levels correspond to different security contexts, and may impact on both the building of unikernel resources and their configuration at runtime. In addition, these resources are typically distributed over several cloud infrastructures. Checking methods and techniques should be exploited to guarantee the consistencies of specified security policies. In particular, this concerns the verification of orchestration policies and their instantiations in different contexts.

7.3.4 Contributing to Cloud Resilience

The design of this security management plane has been motivated by the protection of cloud resources from malicious activities. The solution proposed in this thesis could be extended to cover other resilience requirements (e.g. fault tolerance), using dedicated mechanisms complementary to security mechanisms. Our approach permits to exploit both the building and configuration of resources to address these resilience requirements. For instance, some unikernel resources could be proactively elaborated and instantiated to address some specific faults and failures. In the same manner, resilience constraints could be introduced, when specifying TOSCA cloud orchestration policies. In that context, we could take benefits from the multi-cloud property to increase the resilience of cloud services.

7.4 List of Publications

International Peer-Reviewed Conferences.

- M. Compastié, R. Badonnel, O. Festor, R. He, and M. Kassi-Lahlou. *A Software-Defined Security Strategy for Supporting Autonomic Security Enforcement in Distributed Cloud.* In Proceedings of the IEEE International Conference on Cloud Computing Technology and

Science (IEEE CloudCom), PhD Symposium, 464-467, 2016. https://doi.org/10.1109/CloudCom.2016.0079.

- M. Compastié, R. Badonnel, O. Festor, R. He, and M. Kassi-Lahlou. *Towards a Software-Defined Security Framework for Supporting Distributed Cloud.* In Proceedings of the 11th IFIP International Conference on Autonomous Infrastructure, Management and Security (IFIP AIMS), 47-61, 2017. https://doi.org/10.1007/978-3-319-60774-0_4.

- M. Compastié, R. Badonnel, O. Festor, R. He, and M. Kassi-Lahlou. *Unikernel-based Approach for Software-Defined Security in Cloud Infrastructures.* In Proceeding of the IEEE/IFIP Network Operations and Management Symposium (IEEE/IFIP NOMS), 1-7, 2018. https://doi.org/10.1109/NOMS.2018.8406155.

Demonstration.

- M. Compastié, R. Badonnel, O. Festor, and R. He. *Demo: On-The-Fly Generation of Unikernels for Software-Defined Security in Cloud Infrastructures.* In Proceeding of the 2018 IEEE/IFIP Network Operations and Management Symposium (NOMS), 1-2, 2018. https://doi.org/10.1109/NOMS.2018.8406131.

Industrial Patents.

- M. Compastié and R. He. *Procédé et Système pour Créer une Image d'une Application*, Orange Patent filed with the INPI. No. 201553FR0. 21st March 2018.

In-Submission Work to International Peer-Reviewed Journals.

- M. Compastié, R. Badonnel, O. Festor, and R. He. *Security Issues in System Virtualization And Solutions: A Survey.*

- M. Compastié, R. Badonnel, O. Festor, and R. He. *A TOSCA-Oriented Software-Defined Security Approach for Unikernel-Based Protected Clouds.*

Chapter 8

Appendix

Contents

8.1 Linux Features for OS-level Virtualization Support

OS-level virtualization is based upon host OS features. To explicit the isolation ability supported by the container engines, we describe several examples of Linux OS mechanisms contributing to it.

Chroot. The `chroot()` system call [31] enables the modification of the root filesystem directory of the running process. It is also invokable through the `chroot` base utility.

Its feature contributes to the container filesystem isolation, but is not sufficient to guarantee it, as demonstrated in [151].

In term of implementation, the `chroot()` system call is implemented in [41], while `libcontainer` is instead tied to the `pivot_root()` system call.

Union Mount. Reunifying several media through only one mount operation and mount point defines the union mount concept. It enables to restrain the access to each implicated media, and provides an homogeneous abstraction layer through the mount point.

Type of isolation	**Related Linux features**
Process execution settings	`uts, usr, mnt, cgroup` namespaces
In-Memory process execution environment	`pid, ipc` namespaces, Cgroup
Networking	Linux Bridge, Virtual Ethernet, `net` namespace, Cgroup
I/O	Chroot, Union Mount, `mnt` Namespace, Cgroup

Table 8.1: Linux features supporting OS-level virtualization

One use-case of the operation are the live operating system, designed to be burned on a read-only medium but supporting persistent data storage with an additional writable media: in these conditions, when booting, the operating system proceed to a union mount on root filesystem with the read-only media and the writable media: the first provides the system files whereas the second one is used to store any written file, including user data.

Union filesystem implementations include UnionFS, AuFS (advanced multi-layered unification filesystem), and the OverlayFS. The latter has integrated Linux kernel mainline since version 3.18.

Docker is compatible with AuFS. More concretely with this filesystem, when instantiating a container, it proceeds to the union-mount of `/var/lib/ docker/containers/<container-id>` and `/var/lib/docker/aufs/diff/` as the container root filesystem, according to [39].

Linux-Bridge and Virtual Ethernet. Linux-Bridge [46] is the implementation of the 802.1d standard. It takes over the layer-2 trame forwarding, required for network isolation.

To fulfill this objective, it must be deployed with by a virtual NIC technology enabling a network virtualization such as *virtual-ethernet (veth)*, *MAC Virtual Lan (macvlan)* or *virtual lan (vlan)*.

In the implementation area, container engine relies on various subset of these technologies. LXC [41] allows veth, macvlan or vlan usage, whereas libcontainer (and so, Docker [108]) implements Linux-Bridge and virtual ethernet for its Linux backend.

Control Group (Cgroup). Resource access regulation in Linux OS is performed by the *control group* subsystem (referred as *cgroup*).

In the second version of this subsystem, the ruling is modeled as a hierarchy whose node can be related to resource-specific access ruling controllers [59]. Each one of these relationships defines a resource access control rule for the node and its children. Finally, a node can be purely abstracted or referred to a process.

The first version of cgroup is still heavily used, and is based on a multi-hierarchy ruling model, whose each root node corresponds to a controller (labeled as *subsystem*) [96].

lmctfy and libcontainer are still employing the first version of cgroup, but are planning their migrations.

Namespaces. Allowing Linux processes to address a subset of kernel resources is achieved through the Linux namespaces subsystem. It contributes to the isolation capability of Linux-compatible container engines.

More precisely, there are seven different namespaces, each of them alluding to a resources type: the `mnt` namespace isolates a group of mount point from other process, the `net` one acts on network interfaces, the `pid` one is responsible for process tree isolation , the inter-process communication (*Posix Message Queue*) isolation is handled through the `ipc` one, the `uts` one manages hostname, the `user` one isolates unix user list, and, if the second version of cgroup is loaded, the `cgroup` namespace permits several concurrent cgroup configuration [59].

The namespace mechanism is manipulable with three system calls. `clone()` creates a child process with the same execution context as it ancestor, and accepts additional arguments to create new namespace. `unshare()` takes part in separating the execution environment of the different process. Finally, `setns()` changes the namespace of the current process.

8.2 Glibc System Calls Invokation Implementation

```
1  #include <sysdep.h>
2
3  /* Please consult the file sysdeps/unix/sysv/linux/x86-64/sysdep.h
       for
4     more information about the value -4095 used below.  */
5
6  /* Usage: long syscall (syscall_number, arg1, arg2, arg3, arg4, arg5,
        arg6)
7     We need to do some arg shifting, the syscall_number will be in
8     rax.  */
9
10
11  .text
12 ENTRY (syscall)
13  movq %rdi, %rax  /* Syscall number -> rax.  */
14  movq %rsi, %rdi  /* shift arg1 - arg5.  */
15  movq %rdx, %rsi
16  movq %rcx, %rdx
17  movq %r8, %r10
18  movq %r9, %r8
19  movq 8(%rsp),%r9 /* arg6 is on the stack.  */
20  syscall   /* Do the system call.  */
21  cmpq $-4095, %rax /* Check %rax for error.  */
22  jae SYSCALL_ERROR_LABEL /* Jump to error handler if error.  */
23  ret   /* Return to caller.  */
24
25 PSEUDO_END (syscall)
```

Figure 8.1: Routine from the source code from the GLib responsible for syscall invokation

Figure 8.1 exposes the routines in source code of Glibc 2.23-2 responsible for syscall invokation from unprivileged mode, on x86 64bits system architecture. The syscall function (1) configures CPU registers with syscall number and parameter, (2) relinquishes control to the kernel running in privileged mode with the syscall instruction and then (3) proceed to error management.

Chapitre 9

Résumé détaillé en français du mémoire

Contents

9.1 Introduction

9.1.1 Contexte des travaux

La croissance de l'Internet a été rendue possible par la mise en service de centres de données. Ils fournissent les ressources de calculs (applications logiciels, matériels virtualisés) qui peuvent être partagées et associées pour constituer des services informatiques complexes. Leur utilisation mesurée et efficace permet de diminuer les coûts de mise en place et d'exploitation des infrastructures les supportant. Plus particulièrement, l'informatique nuagique (ou *Cloud Computing* en anglais) a émergé comme modèle permettant l'usage de ces ressources sur Internet. Dans cette optique, un fournisseur de service d'informatique nuagique (CSP) peut mettre à disposition plusieurs types de ressources de calculs (à différents degrés de gérabilité) à des consommateurs pour leurs usages propres. Selon l'institut NIST, l'informatique nuagique peut être mise à disposition en accord avec trois modèles de services différents : *le logiciel en tant que service*, *la plateforme en tant que service* et *l'infrastructure en tant que service*. Ce dernier modèle repose abondamment sur les techniques de virtualisation système pour assurer la configuration et la mise en disponibilité de ressources matérielles aux consommateurs de façon sécurisée.

Ainsi, la virtualisation est une composante importante de l'informatique nuagique permettant de partager et d'isoler les ressources logicielles et matérielles mises à disposition des consommateurs. Dans ce contexte, la virtualisation système permet de distinguer l'exécution de logiciels des ressources matérielles mises à profit. Cette approche est porteuse de bénéfices en termes de sécurité et de résilience. En effet, le contrôle opéré par l'hyperviseur sur les applications opérées en environnements virtualisés peut être exploité pour en contraindre le comportement et pour détecter toute activité susceptible de lui être malveillante. Cependant, la mise en œuvre d'une telle stratégie requiert de connaître et de disposer de moyens propres aux spécificités de l'application à protéger.

Les ressources en environnement d'informatique nuagique peuvent être mises à l'usage d'utilisateurs spécifiques ayant des besoins propres dans les traitements appliqués aux ressources. Dans ce contexte, chaque utilisateur peut être appelé "*tenant*" et un service d'informatique nuagique en acceptant plusieurs est dit "*multi-tenant*". De façon similaire, un service est dit "*multi-cloud*" s'il repose sur l'usage de multiples infrastructures collaborant entre elles. La mise en œuvre d'un service multi-cloud et multi-tenant constitue le cloud distribué, et implique une gestion plus complexe. À titre d'exemple, un ensemble de machines virtuelles supportant un service d'informatique nuagique peut être réparti vers plusieurs propriétaires, et être hébergé par plusieurs centres de données alors que chaque machine virtuelle héberge ses propres programmes et dépendances vis-à-vis des centres de données où elle doit être déployée.

Selon une perspective orientée sécurité, si la gestion opérée sur les ressources vise à satisfaire des objectifs de protection, toute erreur peut compromettre la sécurité des données d'un tenant ou d'un service tout entier. La propriété de large accessibilité réseaux de l'informatique nuagique aggrave la portée de ce constat, en augmentant l'exposition de services d'informatique nuagique et en en faisant des cibles de choix pour des acteurs malveillants. Un premier axe pour mettre en œuvre une gestion de la sécurité s'intéresse au domaine de l'informatique autonome (*autonomic computing*). Celui-ci a offert de nouveaux moyens pour assurer la gestion et l'orchestration en traitant les enjeux d'auto-configuration, d'auto-réparation, d'auto-optimisation et d'auto-protection de systèmes complexes. Il fournit une approche soutenant l'automatisation des environnements d'informatique nuagique, réduisant de facto le coût de leur entretien et les manipulations qui sont sources d'erreurs. Un second axe concerne le paradigme de la programmabilité, qui cherche à séparer les ressources exploitées de leurs gestions en deux plans distincts. À titre d'illustration, la programmabilité des réseaux isole les équipements réseaux, constituant le plan de données, de leur gestion, constituant le plan de gestion. L'association de ces deux axes ouvre une nouvelle perspective pour automatiser et orchestrer la sécurité des clouds distribués. Dans ce contexte, l'objectif principal de cette thèse est de concevoir un plan de gestion de la sécurité unifié et homogène pour la protection de clouds distribués en faisant le lien entre les techniques d'orchestration et de virtualisation.

9.1.2 Identification des problématiques

La constitution d'un tel plan de gestion de la sécurité sous-entend l'identification des critères permettant de déterminer si une ressource est dans un état accepté et se comporte tel qu'attendu, ou si elle requiert une opération de gestion pour atteindre un tel état. La politique de sécurité modélise cette distinction d'état, et est mise en œuvre par une architecture de sécurité et des algorithmes. Une telle architecture permet de s'assurer qu'une application se trouve dans un état acceptable et se conforme à la politique de sécurité. Elle détecte les ressources qui ne sont pas en accord avec les exigences de la politique et les contraint à se mettre en conformité. Dans le contexte de cette thèse, nous nous focalisons sur la gestion de configurations de mécanismes

qui sont programmables.

Par définition, le cloud distribué doit prendre en compte les propriétés de multi-tenancy et de multi-cloud. D'un point de vue sécurité, la multi-tenancy sous entend la possibilité pour chaque utilisateur de définir sa propre politique de sécurité, dont le périmètre d'actions se limite à leurs ressources seules. Cela requiert du plan de gestion de la sécurité qu'il puisse distinguer les différents tenants. La propriété de multi-cloud peut limiter la mise en œuvre de la politique de sécurité. En effet, en déployant des ressources sur certaines infrastructures, certaines d'entre elles ne peuvent se contraindre à la politique de sécurité car l'infrastructure ne fournit pas les moyens techniques de satisfaire cette dernière. Par conséquent, le plan de gestion devrait pouvoir prendre en compte les spécificités de l'infrastructure lors de l'orchestration de la sécurité des ressources.

Le cloud distribué peut aussi induire des exigences sur le plan de gestion de la sécurité. Les infrastructures clouds exploitent une large variété de ressources allouées. Le plan de gestion de la sécurité requiert une architecture capable de manipuler tout type de ressources allouées. En effet, chaque ressource allouée est assujettie à ses propres contraintes techniques : ceci impacte la façon dont le plan de gestion doit mettre en œuvre son action protectrice. Le plan de gestion devrait rester indépendant de ces spécificités techniques dans son processus de prise de décision, alors que leurs mises en œuvres techniques devraient les prendre en compte. La contrainte de haute disponibilité cadre la gestion de l'exploitation de ces ressources. Ces ressources peuvent être exposées à des attaques à tout moment. Il est donc nécessaire de s'assurer de la mise en œuvre de la sécurité tout au long de leurs cycles de vie. Finalement, la conception d'un tel plan de gestion de la sécurité doit prendre en compte la variabilité des exigences de sécurité et des mécanismes associés. Cela sous entend l'extensibilité du langage de spécification de politiques de sécurité, mais aussi les moyens d'intégrer les mécanismes de protection.

Dans cette optique, cette thèse propose la conception d'un plan de gestion homogène de la sécurité pour les clouds distribués selon les problématiques suivantes : (i) Comment peut-on concevoir un plan dédié à la gestion de la sécurité pour mettre en œuvre une protection automatique et homogène de la protection du cloud distribué ? (ii) Comment peut-on s'assurer de la compatibilité entre les ressources du cloud distribué à protéger et la protection offerte par ce plan de gestion ? (iii) De quelle façon les besoins de sécurité des ressources clouds devraient-ils être spécifiés pour la mise en action d'une orchestration de sécurité à partir du plan de gestion ?

9.1.3 Approche proposée

Cette thèse propose une approche centrée sur la programmabilité de la sécurité dans le cloud distribué. Nous utilisons la programmabilité des mécanismes de sécurité protégeant les ressources du cloud pour concevoir un plan de gestion homogène de la sécurité. Celui-ci sert de support pour l'orchestration de sécurité. Il est complété par un plan de sécurité niveau tenant supervisant les opérations de gestion adaptées à chaque tenant, mais supportant de multiples infrastructures. Cette solution permet de diminuer la complexité due à la distribution des ressources. Les mécanismes de sécurité sont intégrés à l'intérieur même des ressources nécessitant une protection, durant leurs phases de conception. Cette solution permet une intégration holistique des mécanismes de sécurité dans les ressources cloud et d'exposer des interfaces au plan de gestion. De plus, sécuriser des ressources avant leurs allocations contribue à l'assurance que la protection est mise en œuvre tout au long du cycle de vie de la ressource. Dans le cadre de cette thèse, nous étudions le cas de clouds distribués et composés de ressources unikernels. Nous tirons profit des propriétés de ces ressources pour réduire la surface d'attaque et faciliter l'intégration de mécanismes de sécurité au moment de la construction. Les ressources peuvent être reconfigurées ou reconstruites pour intégrer les changements dans le contexte de la sécurité. En outre,

nous étendons le langage d'orchestration TOSCA pour conduire la construction de ces ressources unikernels nécessitant une protection et fournir leur configuration selon différents niveaux de sécurité.

9.2 Contributions

La contribution de cette thèse s'organise en quatre parties.

9.2.1 État de l'art des modèles de virtualisation pour le cloud et analyse de leur sécurité

Tel que précédemment précisé, la virtualisation système est un des constituants principaux des environnements d'informatique nuagique. En effet, il permet de dissocier les ressources de calculs de l'infrastructure les supportant. Les virtualisations basées sur les hyperviseurs de type I et de type II sont, depuis des décennies, les modèles de références pour mettre en œuvre la virtualisation système. Cependant, l'émergence de nouveaux modèles remet en cause ce statu quo. Dans cette première contribution, nous présentons ces modèles et dressons une analyse comparative de leur sécurité. Plus particulièrement, nous identifions les vulnérabilités de ces modèles, au regard des menaces existantes, et pondérons leurs criticités en tenant compte des menaces les exploitant. Enfin, nous identifions les contres-mesures et recommandations qui sont applicables pour soutenir la sécurité des ressources virtualisées dans le contexte d'infrastructures d'informatique nuagique.

Modèles de virtualisation système

La virtualisation système définit une architecture permettant l'exécution simultanée et l'isolation de multiples systèmes sur un même environnement matériel. Elle réside en l'insertion d'une couche logicielle encapsulant un système pour en contrôler l'accès aux ressources matérielles. Ce contrôle est permissif et transparent pour les instructions non sensibles des ressources matérielles. En revanche, les instructions sensibles sont interceptées, et leurs effets simulés auprès de ce système. Cette couche d'encapsulation peut être disposée selon deux approches différentes. Dans un premier cas, elle s'exécute en exploitant directement les ressources matérielles, ce qui définit le modèle de virtualisation d'*hyperviseurs de type-I*. Dans l'autre cas, la couche d'encapsulation s'exécute comme application d'un système hôte, et lui délègue sa gestion des ressources matérielles. Elle correspond au modèle de virtualisation d'*hyperviseurs de type-II*. Les contraintes opérationnelles liées à la mise en application de ces deux modèles et la recherche du gain de performances ont incité l'émergence récente de nouveaux modèles de virtualisation : *la virtualisation au niveau du système d'exploitation (S.E.)* est un modèle circonscrivant les environnements virtualisés à des ensembles d'applications pour permettre à un système hôte d'y exercer plus efficacement un contrôle dessus. *La virtualisation basée sur l'unikernels* simplifie l'architecture des systèmes en environnements virtualisés en les limitant au support d'une seule application gérant elle-même les ressources de cet environnement (unikernel). Nous construisons une architecture de référence pour effectuer une analyse comparative de la sécurité de ces quatre modèles de virtualisation. Elle est constituée des niveaux *machine virtuelle*, *système d'exploitation hôte*, *hyperviseur* et *matériel*. La virtualisation système apporte les fonctionnalités de sécurité d'*isolation* inter-machines virtuelles et entre machines virtuelles et système hôte, d'*introspection* de machines virtuelles et de *prise d'instantanés*.

Analyses de sécurité

Nous analysons les différentes vulnérabilités affectant chacun des composants de l'architecture de référence. Nous dressons une taxonomie autour des différents composants impactés : *application en machine virtuelle* (*gestion de la mémoire* et *interface des logiciels*), *S.E. invité en machine virtuelle* (*gestion du logiciel,supervision du noyau du S.E.*), *hyperviseur* (*diaphonie entre les machines virtuelles*, *diaphonie entre les machines virtuelles et l'hôte*, *console de gestion*) et *environnement d'exécution de l'hyperviseur* (*S.E. hôte*, *matériel*). La pondération des différentes vulnérabilités au regard des modèles de virtualisation montre que le modèle de virtualisation basé sur l'unikernel est le moins sujet aux vulnérabilités identifiées dans l'état de l'art. Nous définissons ensuite un modèle de menaces considérant la malveillance de l'utilisateur d'une machine virtuelle et de l'administrateur de l'hyperviseur. Nous appliquons la méthodologie d'analyse de menaces *STRIDE* pour identifier et classifier les attaques affectant les composants selon les menaces d'impersonification, d'altération, de répudiation, d'exfiltration d'informations, de déni de service et d'élévation de privilège. Puis, nous les classifions selon si (i) elles ne sont pas liées à la compromission de composants de l'architecture, (ii) si elles la provoquent, ou (iii) si elles la nécessitent pour être mises en œuvre. Les différentes attaques identifiées sont justifiées et circonstanciées par une analyse les concernant.

Contre-mesures

À partir du travail d'analyse de sécurité, nous proposons des contre-mesures et des recommandations visant la sécurisation de l'architecture de référence pour l'informatique nuagique. Les recommandations identifiées ne remettent pas en cause les bénéfices en terme de sécurité de la virtualisation, et ont un impact limité sur l'exploitabilité des ressources. La première recommandation concerne l'intégration de mécanismes de sécurité dès la conception de l'hyperviseur et de la machine virtuelle. Dans ce contexte, le processus de construction des image unikernels permet l'insertion de tels mécanismes apportant une haute couverture sécuritaire de la ressource virtualisée. La deuxième recommandation vise la minimisation de la surface d'attaque des composants de l'architecture de référence. De part sa simplicité, la virtualisation basée sur l'unikernel est la plus à même de contribuer à sa mise en place. La dernière recommandation propose l'établissement de la programmabilité des mécanismes de sécurité pour en adapter constamment la configuration face aux nouvelles menaces et attaques. Cette programmabilité peut être exploitée par une activité d'orchestration, exploitant les résultats de surveillance de ressources. Elle peut aussi être étendue à la génération des images unikernels sécurisée. Ces trois différentes recommandations cadrent l'élaboration de notre approche exploitant la programmabilité de la sécurité pour la protection du cloud distribué.

9.2.2 Architecture pour la programmabilité de la sécurité dans le cloud distribué

La gestion des ressources en environnements cloud est complexifiée par les propriétés de multi-tenancy et de multi-cloud du cloud distribué. En effet, elles doivent être prises en compte dans la spécification et la mise en rigueur de politiques de sécurité s'adressant à des ressources réparties entre plusieurs infrastructures et tenants. Cette complexité est renforcée par les changements dans le contexte de différentes ressources à protéger induite par les environnements d'informatique nuagique. Les mécanismes automatiques permettent de déporter certaines tâches de gestion aux infrastructures, contribuant ainsi à l'automatisation des services clouds. Le paradigme de la programmabilité fournit aussi une perspective intéressante pour configurer des mécanismes de

sécurité et les aligner sur des exigences de sécurité destinées à différentes infrastructures. Dans cette optique, nous proposons une architecture reposant sur la programmabilité de la sécurité pour instaurer une gestion centralisée et par politique pour la sécurité des environnements clouds distribués. Les objectifs visés par cette architecture sont ainsi *la cohérence de la gestion de la sécurité*, *l'indépendance de la gestion vis-à-vis des contextes techniques de chaque ressource*, *le support de multiples fonctionnalités de sécurité* et *la diminution du coût d'exploitation des services sécurisés*.

Architecture de programmabilité de la sécurité

Pour répondre à ces différents objectifs, nous proposons l'approche SDSec respectant trois principes :

1. La gestion de la sécurité est répartie selon deux plans distincts : (i) le plan de contrôle de la sécurité est en charge de l'interprétation de la politique de sécurité et du processus de prise de décision, et (ii) le plan de ressources correspond à l'ensemble des ressources nécessitant une protection, avec les mécanismes de sécurité pouvant la mettre en œuvre. Ces mécanismes de sécurité sont typiquement dénués de pouvoir décisionnel.

2. La logique mise en œuvre par une fonctionnalité de sécurité est laissée pour le compte du plan de contrôle, correspondant aux fonctionnalités d'un orchestrateur. Les mécanismes de sécurité fournissent l'outillage nécessaire à la mise en œuvre de cette logique. Une fonctionnalité de sécurité se représente donc comme une logique du plan de contrôle, et un ensemble de mécanismes à insérer dans le plan de ressources.

3. Les interactions entre ces deux plans se limitent aux interfaces de configuration des mécanismes de sécurité. Elles ne concernent que les informations nécessaires pour configurer ces mécanismes en accord avec la politique de sécurité.

Nous mettons en œuvre l'approche SDSec dans une architecture supportant la spécification de deux types de politiques de sécurité : (i) la *politique de sécurité globale (GSP)* définissant formellement les objectifs de sécurité et la *politique de sécurité du tenant (TLSP)*, découlant de la première, définissant des règles auxquelles les ressources d'un tenant doivent se conformer. Celles-ci sont hébergées et interprétées respectivement par l'*orchestrateur de sécurité (SO)* et le *point de décision (PDP)*, prenant part au plan de contrôle de la sécurité. Les *points de mise en œuvre (PEPs)* correspondent aux mécanismes insérés dans le plan de ressources, qui interagissent avec les PDPs et sont chargés d'appliquer une configuration sur les ressources clouds pour les conformer à la TLSP. Nous définissons les protocoles permettant aux composants de se découvrir entre eux, et d'interagir pour permettre une mise en œuvre pro-active et réactive de la GSP. Le support de multiples tenants est assuré par l'usage de multiples PDPs propres à la gestion de chacun des tenants. Le support de multiples infrastructures est assuré par la capacité des PDPs d'interagir avec des PEPs instanciés sur des infrastructures différentes.

Évaluation de l'architecture

La viabilité de l'approche est vérifiée à travers cinq scénarios d'usages sélectionnés dans l'environnement industriel de cette thèse. Pour chaque scénario, nous explorons le comportement de l'architecture, et vérifions qu'elle assure la sécurité des ressources via le plan de gestion. Les scénarios explorés incluent (i) l'allocation de ressources, (ii) la mise à jour de politiques de sécurité, (iii) l'attaque de ressources, (iv) la tentative de connexion sous contrôle d'accès et (v)

la suppression de ressources du cloud. Nous complétons cette vérification analytique par une étude technique relative à l'implémentation de l'architecture : nous (i) identifions les contraintes auxquels les implémentations des composants doivent se conformer, (ii) vérifions l'existence de solutions techniques déjà disponibles et (iii) relevons les verrous restant à lever avant de permettre une implémentation exhaustive de l'architecture. Cette dernière catégorie inclut l'intégration et l'adaptation des mécanismes de sécurité aux ressources du cloud nécessitant une protection. Nous nous intéressons à ces deux verrous dans les autres contributions de la thèse.

9.2.3 Génération à la volée d'images unikernels protégées

La mise en œuvre de la politique de sécurité sur les ressources est assujettie à la disponibilité de mécanismes de sécurité qui leur sont adaptés. En effet, le contraire peut induire l'impossibilité de prendre en compte (i) toutes les ressources dans le périmètre à protéger et (ii) toutes leurs spécificités techniques et l'intégralité de leurs cycles de vie. Nous considérons l'usage d'unikernels pour constituer des ressources clouds intégrant des mécanismes de sécurité programmables. En effet, un système unikernel n'intègre qu'une seule application avec ses seules dépendances, permettant donc une couverture plus large du système par un mécanisme de sécurité. De plus, l'unikernel fournit un cadre de travail pour construire des images de systèmes et y effectuer une gestion ex-situ, facilitant l'adaptation des mécanismes aux systèmes à protéger. Enfin, leur court cycle de vie permet une reconfiguration rapide et en profondeur des images instanciées par leur dé-allocation, reconstruction et ré-instanciation. Nous proposons un cadre de travail pour construire des images unikernels en accord avec des exigences de sécurité.

Framework pour la génération d'images unikernels sécurisées

Nous concevons un cadre de travail permettant de générer des images unikernels contraintes pour intégrer des mécanismes de sécurité. Celui-ci étend l'architecture de référence de construction d'image unikernels : les besoins en mécanismes de sécurité sont interprétés à partir du code source de l'image unikernel et de ses dépendances, et sont stockés sur un dépôt qui leur est dédiés. Un plan de gestion séparé est en charge de (i) contrôler la construction des images, (ii) de gérer le cycle de vie des instances et (iii) de piloter la configuration des mécanismes de sécurité pour qu'ils se conforment à des exigences sécuritaires. Les mécanismes ainsi insérés dans l'image unikernel bénéficient ainsi de ses simplicités d'architecture pour viser la configuration de la pile d'exécution et des routines de gestion de ressources matériel en complément de l'application elle-même. Ils sont aussi intégrés dès la conception des ressources à protéger, contribuant à une protection de bout-en-bout de leurs cycles de vie. Nous fournissons une modélisation des images unikernels les représentant comme un ensemble de composants logiciels configurables. Elle nous permet définir l'algorithme d'interprétation des exigences de sécurité, d'insertion des mécanismes et de construction d'images pour chaque évolution du contexte de sécurité. Pour améliorer la réactivité de cette construction, nous proposons une extension de l'algorithme pour supporter le stockage et l'usage d'images pré-construites. Les bénéfices de cette approche incluent (i) une surface d'attaques réduite grâce aux propriétés de l'unikernel, (ii) une vérification de la consistance de l'image protégée dès sa construction et (iii) une portabilité des ressources à protéger accrue. Nous vérifions aussi que cette architecture de génération s'insère à celle de programmabilité de la sécurité pour le cloud distribué.

Évaluation des performances

Nous utilisons un prototype de serveur HTTP implémenté sur la plateforme unikernel *MirageOS* sur l'hyperviseur uKVM, et nous concevons un prototype de mécanisme de sécurité capable de s'y insérer. Ce mécanisme de sécurité implémente l'authentification et l'autorisation des requêtes HTTP. Il est programmable par configuration dans son code source. Nous menons une évaluation quantitative du prototype en comparant (i) son support de charge, (ii) sa consommation mémoire, (iii) son coût mémoire par requête et (iv) le temps de traitement d'une requête. Cette étude compare un unikernel protégé face à une version non protégée et à des solutions basées sur les hyperviseurs de type I, de type II et la virtualisation niveau S.E. De plus, nous évaluons le temps de génération de l'image protégée selon le nombre de mécanismes de sécurité et leur temps de redémarrage séquentiel. Les résultats de ces évaluations soutiennent que le surcoût lié à l'insertion de mécanismes de sécurité est très acceptable tout en offrant une protection très couvrante pour les ressources unikernels.

9.2.4 Spécification de l'orchestration pour la programmabilité de la sécurité

Définir une politique de sécurité à l'échelle d'un cloud distribué signifie savoir appréhender l'hétérogénéité des modèles de sécurité et des mécanismes envers les différents tenants et infrastructures. Chacun d'entre eux doit être pris en compte dans le contexte d'une collaboration inter-tenant et inter-cloud. Dans ce contexte, le langage TOSCA permet de spécifier la constitution et l'orchestration de services cloud devant être déployés sur de multiples infrastructures et de multiples tenants. Les services sont représentés selon une topologie où chaque nœud correspond à un composant instanciable et où les arêtes modélisent une relation entre eux. Les processus d'orchestration spécifiés viennent modifier l'état des nœuds et de leurs relations. Nous proposons deux extensions ai modèle TOSCA : la première, *UniTOSCA*, vient détailler la constitution des ressources unikernels à construire et à orchestrer, et sert de base à l'intégration des mécanismes de sécurité. La seconde, *SecTOSCA*, permet de positionner des exigences de sécurité sur le service selon plusieurs niveaux de sécurité. En outre, nous fournissons un cadre de travail les mettant en application pour constituer des services de cloud distribués protégés. Ce travail se conforme aux besoins de (i) se conformer au contexte du cloud distribué, (ii) d'employer la virtualisation basée sur l'unikernel pour mettre en œuvre la sécurité, et (iii) de s'adapter aux évolutions du contexte de sécurité du service.

Extensions du langage TOSCA

Nous fournissons une modélisation du langage TOSCA en permettant la description des nœuds et de relations les associant. Celle-ci prend en compte le paradigme "orienté objet" présenté par TOSCA en distinguant (i) les types de nœuds et de relations, qui spécifie les fonctionnalités qu'ils fournissent et requièrent, (ii) leurs modèles, qui renseigne les propriétés pour les rendre instanciables, et (iii) leurs instances elles-mêmes. Nous étendons TOSCA en UniTOSCA pour décrire les images unikernels comme un ensemble de composants tout en identifiant les fonctionnalités et nouvelles dépendances qu'apporte chacun d'entre eux. Dans cette spécification, un ensemble cohérent de ces composants s'interprète comme un type du langage TOSCA. Chaque composant peut voir des propriétés de configuration renseignées. Cette approche nous permet d'exploiter une spécification UniTOSCA avec notre cadre de travail de génération d'images unikernels protégées. Enfin, nous proposons l'extension SecTOSCA, qui étend UniTOSCA, pour spécifier les exigences de sécurité à différentes échelles du service, de sa globalité à un composant précis. Les exigences identifiées peuvent faire référence à de multiples fonctionnalités de sécurité, telles que

le contrôle d'accès, la détection d'intrusion ou le chiffrement. SecTOSCA permet de spécifier de multiple valeurs sur les propriétés de sécurité pour permettre de définir différents niveaux de sécurité, qui pourront être mise ne œuvre durant l'exploitation du service. Cette extension permet ainsi de spécifier une orchestration de sécurité lors de la construction et le déploiement d'un service, mais aussi durant son exploitation.

Architecture de conception et d'exploitation de services d'informatique nuagique sécurisés

Nous définissons également un cadre pour construire, déployer et exploiter un service cloud selon notre approche SDSec en exploitant nos différentes extensions de TOSCA. Il repose sur une architecture couvrant la phase de conception et celle d'exploitation d'un service cloud à protéger. Durant la première phase, l'*interpréteur SecTOSCA* consomme une spécification pour configurer l'*orchestrateur de sécurité* et produire une spécification UniTOSCA intégrant les mécanismes de sécurité adéquat. L'*interpréteur UniTOSCA* la récupère pour générer une spécification TOSCA standard et les images unikernels qu'elle exploite. Durant la phase d'exploitation, la spécification TOSCA est exploitée par un *orchestrateur cloud* pendant que l'orchestrateur de sécurité effectue un suivi des ressources unikernels instanciées pour vérifier que leurs configurations se conforment à sa politique. En cas de divergence, il peut reconfigurer les ressources incriminées en exploitant les interfaces de configuration des mécanismes de sécurité embarqués, ou reconstruisant et ré-instanciant intégralement la ressource. En cas d'évolution du contexte de sécurité, il peut contacter l'orchestrateur cloud pour procéder à un changement de niveau de sécurité, induisant une reconfiguration globale du service.

Implémentation et évaluation

Nous proposons un prototype d'implémentation visant (i) la construction d'un générateur d'images unikernels pour le cloud distribué (cadriciel Young), (ii) l'usage du cadriciel Moon comme orchestrateur de sécurité, (iii) la conception et le développement de mécanismes de sécurité, insérables dans des unikernels, (iii) d'autorisation/authentification de requêtes HTTP et (iv) de pare-feu applicatif HTTP. Ces deux mécanismes implémentent trois modèles de programmation : (i) par reconstruction complète de la ressource (interne), (ii) par récupération de configuration d'une entité et (iii) par réception de configurations depuis une entité. Nous conduisons une évaluation quantitative sur les performances des ressources protégées par un modèle de sécurité, sur un ensemble de ressources associant plusieurs modèles et sur leurs propriétés de sécurité. Les résultats montrent que le modèle de récupération de configurations impacte les performances de la ressource protégée, mais est le modèle permettant la propagation plus rapide de politiques de sécurité tout en en supportant des plus complexes.

9.3 Conclusion

9.3.1 Analyse critique

Ce travail s'est focalisé sur la proposition d'une approche basée sur la programmabilité de la sécurité pour le cloud distribué, qui combine une architecture de gestion de la sécurité, avec une génération à la volée d'images unikernels protégées, et avec une extension d'un langage d'orchestration. Le travail présenté est néanmoins limité en plusieurs aspects. Premièrement, seules des ressources unikernels peuvent être générées en version protégée. Leurs propriétés

facilitent la mise en œuvre de politiques de sécurité et permettent de réduire la surface d'attaque. Elles redéfinissent l'étape de reconfiguration au travers la reconstruction complète de ressources. Nous n'avons pas pris en compte des scénarios associant la virtualisation basée sur l'unikernel avec d'autres modèles de virtualisation, alors que le contexte de cloud distribué s'avère suffisamment généraliste pour englober de tels scénarios. Deuxièmement, les interactions entre PEPs et PDPs pourraient être plus complexes. En effet, nous avons seulement considéré le cas où tout PEP est sous l'autorité d'un seul PDP. L'étude de scénarios assignant plusieurs PDPs auprès d'un seul PEP (et des mécanismes de sécurité associé) relèverait un certain intérêt. Elle pourrait introduire des mécanismes de redondance et des propriétés vis-à-vis des périmètres protection. Enfin, nous considérons que les ressources en charge de la protection du service sont disponibles durant toute l'exploitation du service. Ces ressources (dont les PDPs et PEPs font partie) peuvent être affectées par des erreurs et des défaillances qui devraient être prises en compte par l'orchestrateur de sécurité (en sus de l'orchestrateur cloud).

9.3.2 Perspectives de recherche

Les contributions apportées par cette thèse ouvrent la porte à plusieurs perspectives de recherche, identifiées ci-dessous :

Utilisation de l'infrastructure en tant que code (IaC) pour la programmabilité de la sécurité. Notre solution se base sur la construction et l'instanciation d'images unikernels pour mettre en œuvre des politiques de sécurité. Nous avons effectué ce choix en raison de leur architecture simplifiée limitant leurs surfaces d'attaques et permettant la génération à la volée d'images unikernels protégées. Cependant, cela laisse de côté tous les autres types de ressources constituant les infrastructures cloud de nos jours. Notre approche est extensible aux autres catégories de ressources. En particulier, l'approche de l'*infrastructure en tant que code* pourrait contribuer à prendre en compte ces autres ressources et à supporter l'intégration d'une large variété de mécanismes de sécurité. En complément, le court cycle de vie des conteneurs usant de la virtualisation au niveau du S.E. permet de les considérer comme des potentiels candidats vis-à-vis de la génération à la volée de ressources. Cependant, cela requiert un accès à la base de code des conteneurs à protéger.

Support de la sécurité des objets connectés. L'Internet des objets connectés permet d'interconnecter des objets physiques aux réseaux de données pour élaborer de nouveaux services. Il peut aussi exploiter et interagir avec des ressources localisées dans des infrastructures clouds. Les propriétés physiques des ressources et la criticité des données collectées les exposent à des attaques. Les travaux avancés dans cette thèse peuvent être étendus pour englober la protection des objets connectés. Les propriétés des ressources unikernels s'alignent avec les contraintes de limitation en ressources des objets connectés. En revanche, le coût de la programmabilité et des ressources et de leurs réallocations reste encore à être évalué dans ce type d'environnements contraints. L'évaluation des effets d'une alimentation électrique limitée et d'une connectivité fluctuante est aussi un enjeu important à traiter. Les algorithmes de gestion de la sécurité doivent être adaptés pour prendre en compte ces limitations.

Vérification de la cohérence de politiques de sécurité. La programmabilité de la sécurité permet de séparer la gestion des mécanismes de sécurité des ressources qui les intègrent. Le langage d'orchestration TOSCA a été étendu pour spécifier des contraintes de sécurité selon

différents niveaux de sécurité. Ces niveaux correspondent à différents niveaux de contexte de sécurité influençant autant la construction d'images unikernels que leur configuration à l'instanciation. De plus, ces ressources sont généralement réparties sur plusieurs infrastructures clouds. Des méthodes et techniques de vérification devraient être exploitées pour s'assurer de la cohérence des politiques de sécurité spécifiées. En particulier, cela concerne la vérification de politiques d'orchestration et de leurs mises en œuvre dans différents contextes.

Contribution à la résilience de l'informatique nuagique. La conception de ce plan de gestion a été motivé par la protection de ressources d'informatique nuagique contre des activités malveillantes. La solution proposée dans cette thèse peut être étendue pour traiter d'autres exigences de résilience (ex: la tolérance aux fautes), en utilisant des mécanismes dédiés à cet usage. Notre approche nous permet d'utiliser autant la construction que la configuration des ressources pour intégrer les exigences de résilience. À titre d'illustration, des ressources unikernels pourraient être pro-activement conçues et instanciées pour intégrer la gestion d'erreurs et de défaillances particulières. De la même façon, les contraintes de résilience peuvent être intégrées lors de la spécification des politiques d'orchestration cloud TOSCA. Dans ce contexte, nous pourrions tirer profit de la propriété multi-cloud pour améliorer la résilience des services.

Bibliography

[1] *0install: Overview*. URL: http://0install.net/.

[2] *ab - Apache HTTP Server Benchmarking Tool - Apache HTTP Server Version 2.4*. URL: http://httpd.apache.org/docs/2.4/en/programs/ab.html (visited on 02/27/2018).

[3] Pietro Abate et al. "A Modular Package Manager Architecture". In: *Information and Software Technology* 55.2 (2013), pp. 459–474. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2012.09.002.

[4] Cloud Security Alliance. "Top Threats to Cloud Computing v1". In: *White Paper* (2010). URL: https://www.cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf.

[5] *Apt - Debian Wiki*. URL: https://wiki.debian.org/Apt.

[6] Claudio Agostino Ardagna et al. "An XACML-based Privacy-centered Access Control System". In: *Proceedings of the First ACM Workshop on Information Security Governance*. WISG '09. New York, NY, USA: ACM, 2009, pp. 49–58. ISBN: 978-1-60558-787-5. DOI: 10.1145/1655168.1655178. URL: http://doi.acm.org/10.1145/1655168.1655178 (visited on 07/18/2018).

[7] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX." In: *OSDI*. Vol. 16. 2016, pp. 689–703. ISBN: 978 -1- 931971-33 -1. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-arnautov.pdf.

[8] M. Artac et al. "DevOps: Introducing Infrastructure-as-Code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. May 2017, pp. 497–498. DOI: 10.1109/ICSE-C.2017.162.

[9] Samiha Ayed et al. "Achieving dynamicity in security policies enforcement using aspects". en. In: *International Journal of Information Security* 17.1 (Feb. 2018), pp. 83–103. ISSN: 1615-5270. DOI: 10.1007/s10207-016-0357-6.

[10] Andrei Bacs et al. "Slick: An Intrusion Detection System for Virtualized Storage Devices". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. New York, NY, USA: ACM, 2016, pp. 2033–2040. ISBN: 978-1-4503-3739-7. DOI: 10.1145/2851613.2851795.

[11] Thomas Ball et al. "VeriCon: Towards Verifying Controller Programs in Software-defined Networks". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. New York, NY, USA: ACM, 2014, pp. 282–293. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594317. URL: http://doi.acm.org/10.1145/2594291.2594317.

[12] Paul Barham et al. "Xen and the Art of Virtualization". In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462.

[13] M. Barrère, R. Badonnel, and O. Festor. "A SAT-based Autonomous Strategy for Security Vulnerability Management". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. May 2014, pp. 1–9. DOI: 10.1109/NOMS.2014.6838309.

[14] Antonio Barresi et al. "CAIN: Silently Breaking ASLR in the Cloud." In: *WOOT*. 2015. URL: https://www.usenix.org/system/files/conference/woot15/woot15-paper-barresi.pdf.

[15] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *ACM Trans. Comput. Syst.* 33.3 (Aug. 2015), 8:1–8:26. ISSN: 0734-2071. DOI: 10.1145/2799647.

[16] M. Bazm et al. "Side-channels Beyond the Cloud Edge: New Isolation Threats and Solutions". In: *2017 1st Cyber Security in Networking Conference (CSNet)*. 2017 1st Cyber Security in Networking Conference (CSNet). Oct. 2017, pp. 1–8. DOI: 10.1109/CSNET.2017.8241986.

[17] Fabrice Bellard. "QEMU, A Fast and Portable Dynamic Translator." In: *USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 41–46. URL: https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf.

[18] Alysson Bessani et al. "The TClouds Platform: Concept, Architecture and Instantiations". In: *Proceedings of the 2Nd International Workshop on Dependability Issues in Cloud Computing*. DISCCO '13. New York, NY, USA: ACM, 2013, 1:1–1:6. ISBN: 978-1-4503-2248-5. DOI: 10.1145/2506155.2506156.

[19] Bill Parducci, Hal Lockhart, and Erik Rissanen. *eXtensible Access Control Markup Language (XACML) Version 3.0*. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.doc. OASIS Standard, Jan. 22, 2013. URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[20] Tobias Binz et al. "TOSCA: Portable Automated Deployment and Management of Cloud Applications". In: *Advanced Web Services*. Ed. by Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel. New York, NY: Springer New York, 2014, pp. 527–549. ISBN: 978-1-4614-7535-4. DOI: 10.1007/978-1-4614-7535-4_22. URL: http://www-iaas.informatik.uni-stuttgart.de/RUS-data/INBOOK-2014-01%20TOSCA%20Portable%20Automated%20Deployment%20and%20Management%20of%20Cloud%20Applications.pdf.

[21] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to Process Management*. 3rd ed. O'Reilly Media, Inc., 2005. 929 pp. ISBN: 978-0-596-00565-8.

[22] A. Bratterud et al. "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services". In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. Nov. 2015, pp. 250–257. DOI: 10.1109/CloudCom.2015.89.

[23] Alfred Bratterud, Andreas Happe, and Robert Anderson Keith Duncan. "Enhancing Cloud Security and Privacy: The Unikernel Solution". In: *Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, 19 February 2017-23 February 2017, Athens, Greece*. Curran Associates, 2017. URL: http://aura.abdn.ac.uk/bitstream/handle/2164/8524/AAB02.pdf?sequence=1 (visited on 08/04/2017).

[24] Justin Cappos et al. "A Look in the Mirror: Attacks on Package Managers". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. New York, NY, USA: ACM, 2008, pp. 565–574. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455841.

[25] M. Carbone, D. Zamboni, and W. Lee. "Taming Virtualization". In: *IEEE Security Privacy* 6.1 (Jan. 2008), pp. 65–67. ISSN: 1540-7993. DOI: 10.1109/MSP.2008.24.

[26] M. Carpenter, T. Liston, and E. Skoudis. "Hiding Virtualization from Attackers and Malware". In: *IEEE Security Privacy* 5.3 (May 2007), pp. 62–65. ISSN: 1540-7993. DOI: 10.1109/MSP.2007.63.

[27] *Centralized Policy Engine to Enable multiple OpenStack deployments for Telco/NFV*. OpenStack. URL: https://www.openstack.org/summit/vancouver-2018/summit-schedule/events/21536/centralized-policy-engine-to-enable-multiple-openstack-deployments-for-telconfv (visited on 09/08/2018).

[28] Stephen Checkoway and Hovav Shacham. "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 253–264. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451145. (Visited on 01/22/2018).

[29] David R. Cheriton and Kenneth J. Duda. "A Caching Model of Operating System Kernel Functionality". In: *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*. OSDI '94. Berkeley, CA, USA: USENIX Association, 1994. URL: http://dl.acm.org/citation.cfm?id=1267638.1267652.

[30] Mihai Christodorescu et al. "Cloud Security Is Not (Just) Virtualization Security: a Short Paper". In: *Proceedings of the 2009 ACM workshop on Cloud computing security - CCSW '09*. the 2009 ACM workshop. Chicago, Illinois, USA: ACM Press, 2009, p. 97. ISBN: 978-1-60558-784-4. DOI: 10.1145/1655008.1655022. (Visited on 09/01/2018).

[31] *chroot(2) - Linux man page*. URL: https://linux.die.net/man/2/chroot (visited on 10/17/2018).

[32] C. J. Chung et al. "NICE: Network Intrusion Detection and Countermeasure Selection in Virtual Network Systems". In: *IEEE Transactions on Dependable and Secure Computing* 10.4 (July 2013), pp. 198–211. ISSN: 1545-5971. DOI: 10.1109/TDSC.2013.8.

[33] *Cohttp: Very lightweight HTTP Server using Lwt or Async*. MirageOS, Sept. 2017. URL: https://github.com/mirage/ocaml-cohttp (visited on 09/04/2017).

[34] Patrick Colp et al. "Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. New York, NY, USA: ACM, 2011, pp. 189–202. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043575.

[35] John Criswell, Nathan Dautenhahn, and Vikram Adve. "Virtual Ghost: Protecting Applications from Hostile Operating Systems". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 81–96. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541986. (Visited on 06/13/2018).

[36] Nathan Dautenhahn et al. "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation". In: *SIGARCH Comput. Archit. News* 43.1 (Mar. 2015), pp. 191–206. ISSN: 0163-5964. DOI: 10.1145/2786763.2694386.

[37] Roberto Di Cosmo, Stefano Zacchiroli, and Paulo Trezentos. "Package Upgrades in FOSS Distributions: Details and Challenges". In: *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*. HotSWUp '08. New York, NY, USA: ACM, 2008, 7:1–7:5. ISBN: 978-1-60558-304-4. DOI: 10.1145/1490283.1490292.

[38] *Docker - Build, Ship, and Run Any App, Anywhere*. URL: https://www.docker.com/ (visited on 10/17/2018).

[39] *Docker and AUFS in practice*. URL: https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/ (visited on 06/20/2017).

[40] G. Dreo et al. "ICEMAN: An architecture for secure federated inter-cloud identity management". In: *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013). May 2013, pp. 1207–1210.

[41] R. Dua, A. R. Raja, and D. Kakadia. "Virtualization vs Containerization to Support PaaS". In: *2014 IEEE International Conference on Cloud Engineering*. Mar. 2014, pp. 610–614. DOI: 10.1109/IC2E.2014.41.

[42] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. "Exokernel: An Operating System Architecture for Application-level Resource Management". In: *SIGOPS Oper. Syst. Rev.* 29.5 (Dec. 1995), pp. 251–266. ISSN: 0163-5980. DOI: 10.1145/224057.224076.

[43] *EU GDPR Information Portal*. EU GDPR Portal. URL: http://eugdpr.org/eugdpr.org-1.html (visited on 06/29/2018).

[44] Seyed Kaveh Fayazbakhsh et al. "FlowTags: Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions". In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking - HotSDN '13*. the second ACM SIGCOMM workshop. Hong Kong, China: ACM Press, 2013, p. 19. ISBN: 978-1-4503-2178-5. DOI: 10.1145/2491185.2491203.

[45] W. Felter et al. "An Updated Performance Comparison of Virtual Machines and Linux Containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[46] The Linux Foundation. *Bridge*. URL: https://wiki.linuxfoundation.org/networking/bridge.

[47] Tal Garfinkel, Mendel Rosenblum, et al. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." In: *Ndss*. Vol. 3. 2003, pp. 191–206. URL: http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/13.pdf.

[48] Jason Geffner. *VENOM Vulnerability*. URL: http://venom.crowdstrike.com/.

[49] Will Glozer. *wrk: Modern HTTP Benchmarking Tool*. original-date: 2012-03-20T11:12:28Z. Sept. 5, 2017. URL: https://github.com/wg/wrk (visited on 09/05/2017).

[50] Ashvin Goel et al. "The Taser Intrusion Recovery System". In: *SIGOPS Oper. Syst. Rev.* 39.5 (Oct. 2005), pp. 163–176. ISSN: 0163-5980. DOI: 10.1145/1095809.1095826.

[51] R. P. Goldberg. "Survey of Virtual Machine Research". In: *Computer* 7.6 (June 1974), pp. 34–45. ISSN: 0018-9162. DOI: 10.1109/MC.1974.6323581. URL: http://web.eecs.umich.edu/~prabal/teaching/eecs582-w11/readings/Gol74.pdf.

[52] Guofei Gu et al. "Building a Security OS With Software Defined Infrastructure". In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys '17. New York, NY, USA: ACM, 2017, 4:1–4:8. ISBN: 978-1-4503-5197-3. DOI: 10.1145/3124680.3124720. (Visited on 10/25/2017).

[53] W. Gu et al. "Characterization of Linux Kernel Behavior under Errors". In: *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.(DSN)*. 2003 International Conference on Dependable Systems and Networks. Vol. 00. San Francisco, California, June 22, 2003, p. 459. ISBN: 0-7695-1952-0. DOI: 10.1109/DSN.2003.1209956. URL: doi.ieeecomputersociety.org/10.1109/DSN.2003.1209956.

[54] *gVisor: Container Runtime Sandbox*. July 2018. URL: https://github.com/google/gvisor (visited on 07/08/2018).

[55] Safaà Hachana, Nora Cuppens-Boulahia, and Frédéric Cuppens. "Mining a high level access control policy in a network with multiple firewalls". In: *Journal of Information Security and Applications*. Security, Privacy and Trust in Future Networks and Mobile Computing 20 (Feb. 1, 2015), pp. 61–73. ISSN: 2214-2126. DOI: 10.1016/j.jisa.2014.10.010.

[56] Akram Hakiri et al. "Software-Defined Networking: Challenges and research opportunities for Future Internet". In: *Computer Networks* 75 (Dec. 24, 2014), pp. 453–471. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2014.10.015. URL: http://www.sciencedirect.com/science/article/pii/S1389128614003703 (visited on 10/19/2018).

[57] Reed Hastings and Bob Joyce. "Purify: Fast Detection of Memory Leaks and Access Errors". In: *Proceedings of the Winter 1992 USENIX Conference*. 1991, pp. 125–138. DOI: 10.1.1.184.8081.

[58] Ruan He et al. "SDAC: A New Software-Defined Access Control Paradigm for Cloud-Based Systems". In: *Information and Communications Security*. Ed. by Sihan Qing et al. Cham: Springer International Publishing, 2018, pp. 570–581. ISBN: 978-3-319-89500-0. DOI: 10.1007/978-3-319-89500-0_49.

[59] Tejun Heo. *Control Group v2*. Oct. 2015. URL: https://www.kernel.org/doc/Documentation/cgroup-v2.txt.

[60] Jens Heyens, Kai Greshake, and Eric Petryka. "MongoDB Databases at Risk". In: *Center for IT-Security, Privacy, and Accountability* (Jan. 2015).

[61] G. Hurel et al. "Towards Cloud-based Compositions of Security Functions for Mobile Devices". In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM). May 2015, pp. 578–584. DOI: 10.1109/INM.2015.7140340.

[62] J. Park J. Jeong H. Kim. *Software-Defined Networking Based Security Services using Interface to Network Security Functions*. Oct. 2015.

[63] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. "Analyzing Integrity Protection in the SELinux Example Policy". In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 5–5. URL: http://dl.acm.org/citation.cfm?id=1251353.1251358 (visited on 06/22/2018).

[64] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. "Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. New York, NY, USA: ACM, 2007, pp. 128–138. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315262.

[65] Johns Martin. "Code-injection Vulnerabilities in Web Applications — Exemplified at Cross-site Scripting". In: *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik* 53.5 (2011), pp. 256–160. DOI: 10.1524/itit.2011.0651. URL: https://www.degruyter.com/view/j/itit.2011.53.issue-5/itit.2011.0651/itit.2011.0651.xml (visited on 10/17/2018).

[66] H. Kang, M. Le, and S. Tao. "Container and Microservice Driven Design for Cloud Infrastructure DevOps". In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. 2016 IEEE International Conference on Cloud Engineering (IC2E). Apr. 2016, pp. 202–211. DOI: 10.1109/IC2E.2016.26.

[67] *Kata Containers - The Speed of Containers, the Security of VMs*. URL: https://katacontainers.io/ (visited on 07/08/2018).

[68] J. O. Kephart and D. M. Chess. "The Vision of Autonomic Computing". In: *Computer* 36.1 (Jan. 2003), pp. 41–50. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1160055.

[69] S. T. King and P. M. Chen. "SubVirt: implementing malware with virtual machines". In: *2006 IEEE Symposium on Security and Privacy (S P'06)*. May 2006, 14 pp.–327. DOI: 10.1109/SP.2006.38.

[70] Avi Kivity et al. "KVM: the Linux Virtual Machine Monitor". In: *Proceedings of the Linux symposium*. Linux Symposium. Vol. 1. Ottawa, Ontario Canada, June 2007, pp. 225–230.

[71] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. New York, NY, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. (Visited on 06/07/2018).

[72] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology — CRYPTO' 99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings*. Ed. by Michael Wiener. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397. ISBN: 978-3-540-48405-9. DOI: 10.1007/3-540-48405-1_25. URL: http://dx.doi.org/10.1007/3-540-48405-1_25.

[73] Kirill Kolyshkin. "Virtualization in Linux". In: *White paper, OpenVZ* 3 (2006), p. 39.

[74] Joongjin Kook et al. "Optimization of out of Memory Killer for Embedded Linux Environments". In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. New York, NY, USA: ACM, 2011, pp. 633–634. ISBN: 978-1-4503-0113-8. DOI: 10.1145/1982185.1982324.

[75] Tommy Koorevaar. "Dynamic Enforcement of Security Policies in Multi-Tenant Cloud Networks". Master's Thesis. École Polytechnique de Montréal, 2012. URL: https://publications.polymtl.ca/1056/.

[76] Kostya Kortchinsky. "Cloudburst". In: *Black Hat USA* (2009). URL: http://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-PAPER.pdf.

[77] James Larkby-Lahet et al. "Xomb: an Exokernel for Modern 64-bit, Multicore Hardware". In: *WSO-VII Workshop de Sistemas Operacionais*. 2010, pp. 1991–1998. URL: http://www.lisha.ufsc.br/wso/wso2010/papers/st02_02.pdf.

[78] K. K. Lau and Z. Wang. "Software Component Models". In: *IEEE Transactions on Software Engineering* 33.10 (Oct. 2007), pp. 709–724. ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70726.

[79] C. Li, A. Raghunathan, and N. K. Jha. "Secure Virtual Machine Execution under an Untrusted Management OS". In: *2010 IEEE 3rd International Conference on Cloud Computing*. July 2010, pp. 172–179. DOI: 10.1109/CLOUD.2010.29.

[80] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. "Implementing an Untrusted Operating System on Trusted Hardware". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. New York, NY, USA: ACM, 2003, pp. 178–192. ISBN: 978-1-58113-757-6. DOI: 10.1145/945445.945463. (Visited on 06/13/2018).

[81] J. Liedtke. "On Micro-kernel Construction". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. New York, NY, USA: ACM, 1995, pp. 237–250. ISBN: 978-0-89791-715-5. DOI: 10.1145/224056.224075. (Visited on 09/11/2017).

[82] C. H. Lin, C. H. Chen, and C. S. Laih. "A Study and Implementation of Vulnerability Assessment and Misconfiguration Detection". In: *2008 IEEE Asia-Pacific Services Computing Conference*. Dec. 2008, pp. 1252–1257. DOI: 10.1109/APSCC.2008.212.

[83] *Linux Containers - LXC - Introduction*. URL: https://linuxcontainers.org/fr/lxc/introduction/ (visited on 10/17/2018).

[84] Jiaqiang Liu et al. "Leveraging Software-defined Networking for Security Policy Enforcement". In: *Information Sciences* 327.Supplement C (Jan. 2016), pp. 288–299. ISSN: 0020-0255. DOI: 10.1016/j.ins.2015.08.019. (Visited on 11/08/2017).

[85] Torsten Lodderstedt, David Basin, and Jürgen Doser. "SecureUML: A UML-Based Modeling Language for Model-Driven Security". In: *UML 2002 — The Unified Modeling Language*. Ed. by Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 426–441. ISBN: 978-3-540-45800-5. DOI: 10.1007/3-540-45800-X_33.

[86] Markus Lorch et al. "First Experiences Using XACML for Access Control in Distributed Systems". In: *Proceedings of the 2003 ACM Workshop on XML Security*. XMLSEC '03. New York, NY, USA: ACM, 2003, pp. 25–37. ISBN: 978-1-58113-777-4. DOI: 10.1145/968559.968563.

[87] B. M. Luettmann and A. C. Bender. "Man-in-the-middle attacks on auto-updating software". In: *Bell Labs Technical Journal* 12.3 (2007), pp. 131–138. ISSN: 1538-7305. DOI: 10.1002/bltj.20255.

[88] Song Luo and M. Ben Salem. "Orchestration of software-defined security services". In: *2016 IEEE International Conference on Communications Workshops (ICC)*. 2016 IEEE International Conference on Communications Workshops (ICC). May 2016, pp. 436–441. DOI: 10.1109/ICCW.2016.7503826.

[89] Anil Madhavapeddy et al. "Jitsu: Just-In-Time Summoning of Unikernels." In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15). Oakland, CA, USA, May 4, 2015, pp. 559–573. ISBN: 978-1-931971-21-8. URL: https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-madhavapeddy.pdf.

[90] Anil Madhavapeddy and David J. Scott. "Unikernels: Rise of the Virtual Library Operating System". In: *Queue* 11.11 (Dec. 2013), 30:30–30:44. ISSN: 1542-7730. DOI: 10.1145/2557963.2566628.

[91] Anil Madhavapeddy et al. "Unikernels: Library Operating Systems for the Cloud". In: *SIGPLAN Not.* 48.4 (Mar. 2013), pp. 461–472. ISSN: 0362-1340. DOI: 10.1145/2499368.2451167. URL: http://doi.acm.org/10.1145/2499368.2451167.

[92] Eve Maler et al. "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML)". In: *OASIS, September* (2003).

[93] F. Mancinelli et al. "Managing the Complexity of Large Free and Open Source Package-Based Software Distributions". In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). Sept. 2006, pp. 199–208. DOI: 10.1109/ASE.2006.49.

[94] Petr Matousek. *VENOM, Don't Get Bitten*. May 13, 2015. URL: https://access.redhat.com/blogs/product-security/posts/1976633.

[95] Peter Mell and Tim Grance. "The NIST Definition of Cloud Computing". In: (2011). URL: http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf.

[96] Paul Menage. *CGROUPS*. URL: https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/plain/Documentation/cgroup-v1/cgroups.txt?h=linux-4.9.y&id=refs/tags/v4.9.6.

[97] *Mirage Skeleton: Examples of simple MirageOS Applications - Static website TLS*. MirageOS, Sept. 2017. URL: https://github.com/mirage/mirage-skeleton/tree/master/applications/static_website_tls (visited on 09/04/2017).

[98] MITRE. *CVE-2007-1744*. 2007. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1744.

[99] MITRE. *CVE-2012-0217*. 2011. URL: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0217.

[100] MITRE. *CVE-2013-0273*. Dec. 2013. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0273.

[101] MITRE. *CVE-2014-0230*. 2013. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0230.

[102] MITRE. *CVE-2015-3456*. 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456.

[103] MITRE. *CVE-2016-3172*. 2016. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9962.

[104] MITRE. *CVE-2016-9919*. 2016. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9919.

[105] MITRE. *CVE-2017-3791*. Dec. 2016. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-3791.

[106] *Moon - Security Management Module*. URL: https://git.opnfv.org/moon/ (visited on 09/08/2018).

[107] Mathias Morbitzer et al. "SEVered: Subverting AMD's Virtual Machine Encryption". In: *Proceedings of the 11th European Workshop on Systems Security*. EuroSec'18. New York, NY, USA: ACM, 2018, 1:1–1:6. ISBN: 978-1-4503-5652-7. DOI: 10.1145/3193111.3193112. (Visited on 05/28/2018).

[108] Adrian Mouat. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, Inc., 2015.

[109] K. Nance, M. Bishop, and B. Hay. "Virtual Machine Introspection: Observation or Interference?" In: *IEEE Security Privacy* 6.5 (Sept. 2008), pp. 32–37. ISSN: 1540-7993. DOI: 10.1109/MSP.2008.134.

[110] Ankur Kumar Nayak et al. "Resonance: Dynamic Access Control for Enterprise Networks". In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. WREN '09. New York, NY, USA: ACM, 2009, pp. 11–18. ISBN: 978-1-60558-443-0. DOI: 10.1145/1592681.1592684. URL: http://doi.acm.org/10.1145/1592681.1592684 (visited on 07/02/2017).

[111] Luke Nelson et al. "Hyperkernel: Push-Button Verification of an OS Kernel". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. New York, NY, USA: ACM, 2017, pp. 252–269. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132748. (Visited on 01/18/2018).

[112] Robert H. B. Netzer and Barton P. Miller. "What Are Race Conditions?: Some Issues and Formalizations". In: *ACM Lett. Program. Lang. Syst.* 1.1 (Mar. 1992), pp. 74–88. ISSN: 1057-4514. DOI: 10.1145/130616.130623.

[113] *OCaml Package Manager*. URL: http://opam.ocaml.org/ (visited on 08/01/2017).

[114] Fuzen Op. *The FU Rootkit*. 2008.

[115] *Open vSwitch*. URL: http://www.openvswitch.org/ (visited on 02/27/2018).

[116] *Oracle VM VirtualBox*. URL: https://www.virtualbox.org/ (visited on 10/16/2018).

[117] Tavis Ormandy. *An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments*. 2007. 10 pp. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.6943&rep=rep1&type=pdf.

[118] Derek Palma and Thomas Spatzier. "Topology and Orchestration Specification for Cloud Applications (TOSCA)". In: *Organization for the Advancement of Structured Information Standards (OASIS), Tech. Rep* (2013). URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf.

[119] M. Pattaranantakul et al. "SecMANO: Towards Network Functions Virtualization (NFV) Based Security MANagement and Orchestration". In: *2016 IEEE Trustcom/BigDataSE/ISPA*. 2016 IEEE Trustcom/BigDataSE/ISPA. Aug. 2016, pp. 598–605. DOI: 10.1109/TrustCom.2016.0115.

[120] Montida Pattaranantakul et al. "A First Step Towards Security Extension for NFV Orchestrator". In: *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. SDN-NFVSec '17. New York, NY, USA: ACM, 2017, pp. 25–30. ISBN: 978-1-4503-4908-6. DOI: 10.1145/3040992.3040995. (Visited on 03/26/2018).

[121] Montida Pattaranantakul et al. "NFV Security Survey: From Use Case Driven Threat Analysis to State-of-the-Art Countermeasures". In: *IEEE Communications Surveys & Tutorials* (2018), pp. 1–1. ISSN: 1553-877X, 2373-745X. DOI: 10.1109/COMST.2018.2859449. URL: https://ieeexplore.ieee.org/document/8419241/ (visited on 08/28/2018).

[122] B. D. Payne et al. "Lares: An Architecture for Secure Active Monitoring Using Virtualization". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. May 2008, pp. 233–247. DOI: 10.1109/SP.2008.24.

[123] Michael Pearce, Sherali Zeadally, and Ray Hunt. "Virtualization: Issues, Security Threats, and Solutions". In: *ACM Comput. Surv.* 45.2 (Mar. 2013), 17:1–17:39. ISSN: 0360-0300. DOI: 10.1145/2431211.2431216. URL: http://doi.acm.org/10.1145/2431211.2431216.

[124] J. Pincus and B. Baker. "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns". In: *IEEE Security Privacy* 2.4 (July 2004), pp. 20–27. ISSN: 1540-7993. DOI: 10.1109/MSP.2004.36.

[125] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. (Visited on 08/16/2017).

[126] Donald E. Porter et al. "Rethinking the Library OS from the Top Down". In: *SIGARCH Comput. Archit. News* 39.1 (Mar. 2011), pp. 291–304. ISSN: 0163-5964. DOI: 10.1145/1961295.1950399.

[127] *QEMU*. URL: http://www.qemu-project.org/ (visited on 10/17/2018).

[128] Feng Qin, Shan Lu, and Yuanyuan Zhou. "SafeMem: Exploiting ECC-memory for Detecting Memory Leaks and Memory Corruption During Production Runs". In: *11th International Symposium on High-Performance Computer Architecture*. Feb. 2005, pp. 291–302. DOI: 10.1109/HPCA.2005.29.

[129] Danny Quist, Val Smith, and Offensive Computing. "Detecting the Presence of Virtual Machines Using the Local Data Table". In: *Offensive Computing* (2006).

[130] *RabbitMQ - Messaging that just works*. URL: https://www.rabbitmq.com/ (visited on 08/25/2018).

[131] Ryan Riley, Xuxian Jiang, and Dongyan Xu. "Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing". In: *Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings*. Ed. by Richard Lippmann, Engin Kirda, and Ari Trachtenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–20. ISBN: 978-3-540-87403-4. DOI: 10.1007/978-3-540-87403-4_1.

[132] Thomas Ristenpart et al. "Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. New York, NY, USA: ACM, 2009, pp. 199–212. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653687.

[133] John S Robin and Cynthia E Irvine. "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor". In: *Proceedings of the 9th USENIX Security Symposium*. 9th USENIX Security. Denver, Colorado, USA: DTIC Document, Aug. 14, 2000, pp. 129–144. URL: https://www.usenix.org/legacy/events/sec00/full_papers/robin/robin_html/.

[134] Rafael Román Otero and Alex A. Aravind. "MiniOS: An Instructional Platform for Teaching Operating Systems Projects". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 430–435. ISBN: 978-1-4503-2966-8. DOI: 10.1145/2676723.2677299. (Visited on 08/08/2017).

[135] Olubisi Atinuke Runsewe. "A Policy-Based Management Framework for Cloud Computing Security". Master's Thesis. University of Ottawa, 2014. URL: https://ruor.uottawa.ca/handle/10393/31503.

[136] Matt Rutkowski and Luc Boutier. "TOSCA Simple Profile in YAML Version 1.1". In: *OASIS Committee Specification Draft* (2016). URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/csprd01/TOSCA-Simple-Profile-YAML-v1.1-csprd01.html.

[137] J. Sahoo, S. Mohapatra, and R. Lath. "Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues". In: *2010 Second International Conference on Computer and Network Technology*. Apr. 2010, pp. 222–226. DOI: 10.1109/ICCNT.2010.49.

[138] R. Sailer et al. "Building a MAC-based security architecture for the Xen open-source hypervisor". In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. Dec. 2005, 10 pp.–285. DOI: 10.1109/CSAC.2005.13.

[139] Team Teso Scut. *Exploiting Format String Vulnerabilities*. Jan. 9, 2001. URL: http://julianor.tripod.com/bc/formatstring-1.2.pdf (visited on 10/17/2018).

[140] Hovav Shacham et al. "On the Effectiveness of Address-space Randomization". In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. ISBN: 1-58113-961-6. DOI: 10.1145/1030083.1030124.

[141] Ehab Al-Shaer and Saeed Al-Haj. "FlowChecker: Configuration Analysis and Verification of Federated Openflow Infrastructures". In: *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*. SafeConfig '10. New York, NY, USA: ACM, 2010, pp. 37–44. ISBN: 978-1-4503-0093-3. DOI: 10.1145/1866898.1866905.

[142] Seungwon Shin et al. "FRESCO: Modular Composable Security Services for Software-Defined Networks." In: *NDSS*. 2013. URL: http://koasas.kaist.ac.kr/bitstream/10203/205914/1/fresco_ndss13.pdf.

[143] Jason Solomon et al. "User Data Persistence in Physical Memory". In: *Digital Investigation* 4.2 (2007), pp. 68–72. ISSN: 1742-2876. DOI: 10.1016/j.diin.2007.03.002.

[144] Stephen Soltesz et al. "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors". In: *SIGOPS Oper. Syst. Rev.* 41.3 (Mar. 2007), pp. 275–287. ISSN: 0163-5980. DOI: 10.1145/1272998.1273025.

[145] Udo Steinberg and Bernhard Kauer. "NOVA: A Microhypervisor-based Secure Virtualization Architecture". In: *Proceedings of the 5th European Conference on Computer Systems*. 5th European Conference on Computer Systems. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 209–222. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755935.

[146] J. Szefer and R. B. Lee. "A Case for Hardware Protection of Guest VMs from Compromised Hypervisors in Cloud Computing". In: *2011 31st International Conference on Distributed Computing Systems Workshops*. June 2011, pp. 248–252. DOI: 10.1109/ICDCSW.2011.51.

[147] *The Solo5 Unikernel Project*. Solo5, Sept. 2017. URL: https://github.com/djwillia/solo5 (visited on 09/04/2017).

[148] *The Xen Project, the Powerful Open Source Industry Standard for Virtualization*. URL: https://xenproject.org/.

[149] P. Torr. "Demystifying the Threat Modeling Process". In: *IEEE Security Privacy* 3.5 (Sept. 2005), pp. 66–70. ISSN: 1540-7993. DOI: 10.1109/MSP.2005.119.

[150] U. Tupakula and V. Varadharajan. "TVDSEC: Trusted Virtual Domain Security". In: *2011 Fourth IEEE International Conference on Utility and Cloud Computing*. Dec. 2011, pp. 57–64. DOI: 10.1109/UCC.2011.18.

[151] Filippo Valsorda. *Escaping a chroot jail/1*. 2013. URL: https://filippo.io/escaping-a-chroot-jail-slash-1/ (visited on 10/17/2018).

[152] Yiannis Verginadis et al. "PaaSword: A Holistic Data Privacy and Security by Design Framework for Cloud Services". In: *Journal of Grid Computing* 15.2 (June 1, 2017), pp. 219–234. ISSN: 1570-7873, 1572-9184. DOI: 10.1007/s10723-017-9394-2. URL: https://link.springer.com/article/10.1007/s10723-017-9394-2 (visited on 06/11/2018).

[153] P. Verissimo, A. Bessani, and M. Pasin. "The TClouds architecture: Open and resilient cloud-of-clouds computing". In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012). June 2012, pp. 1–6. DOI: 10.1109/DSNW.2012.6264686.

[154] *VMware Virtualization for Desktop & Server, Application, Public & Hybrid Clouds*. URL: http://www.vmware.com/ (visited on 10/17/2018).

[155] Carl A. Waldspurger. "Memory Resource Management in VMware ESX Server". In: *SIGOPS Oper. Syst. Rev.* 36 (SI Dec. 2002), pp. 181–194. ISSN: 0163-5980. DOI: 10.1145/844128.844146.

[156] Alf-Andre Walla. "Live Updating in Unikernels". Master's Thesis. 2017. 118 pp. URL: https://www.duo.uio.no/bitstream/handle/10852/59240/live-updating-unikernels.pdf?sequence=45.

[157] Adrian Waller et al. "Policy Based Management for Security in Cloud Computing". In: *Secure and Trust Computing, Data Management, and Applications*. Ed. by Changhoon Lee et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 130–137. ISBN: 978-3-642-22365-5. DOI: 10.1007/978-3-642-22365-5_16.

[158] David Waltermire et al. "The Technical Specification for the Security Content Automation Protocol (SCAP): SCAP version 1.2". In: *NIST Special Publication* 800 (Sept. 30, 2011), p. 126. DOI: 10.6028/NIST.SP.800-126r2.

[159] Dan Williams. "Run Mirage Unikernels on KVM/QEMU with Solo5". In: (Jan. 7, 2016). URL: https://mirage.io/blog/introducing-solo5 (visited on 10/17/2018).

[160] Dan Williams and Ricardo Koller. "Unikernel Monitors: Extending Minimalism Outside of the Box". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, 2016. URL: https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_williams.pdf.

[161] Rafal Wojtczuk. "Subverting the Xen hypervisor". In: *Black Hat USA*. Vol. 2008. Aug. 7, 2008. URL: https://www.blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf.

[162] Rafal Wojtczuk and Joanna Rutkowska. "Attacking Intel Trusted Execution Technology". In: *Black Hat DC* 2009 (2009).

[163] Rafal Wojtczuk and Joanna Rutkowska. *Following the White Rabbit: Software attacks against Intel VT-d technology*. 2011. URL: https://invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf.

[164] *Xen MiniOS*. original-date: 2015-05-11T00:44:47Z. Mar. 7, 2017. URL: https://github.com/mirage/mini-os (visited on 09/04/2017).

[165] Yuan Xiao et al. "One Bit Flips, One Cloud Flops: Cross-vm Row Hammer Attacks and Privilege Escalation". In: *Proceedings of the 25th USENIX Security Symposium*. 25th USENIX Security Symposium. Austin, TX, Aug. 10, 2016, p. 18. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_xiao.pdf.

[166] *Y.2060 : Overview of the Internet of Things*. URL: https://www.itu.int/rec/T-REC-Y.2060-201206-I/en (visited on 09/13/2018).

[167] Yinqian Zhang et al. "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. New York, NY, USA: ACM, 2012, pp. 305–316. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382230.

[168] Fangfei Zhou et al. "Scheduler Vulnerabilities and Coordinated Attacks in Cloud Computing". In: *Journal of Computer Security* 21.4 (2013), pp. 533–559. DOI: 10.3233/JCS-130474.

Résumé

Dans cette thèse, nous proposons une approche pour la sécurité programmable dans le cloud distribué. Plus spécifiquement, nous montrons de quelle façon cette programmabilité peut contribuer à la protection de services cloud distribués, à travers la génération d'images unikernels fortement contraintes. Celles-ci sont instanciées sous forme de machines virtuelles légères, dont la surface d'attaque est réduite et dont la sécurité est pilotée par un orchestrateur de sécurité. Les contributions de cette thèse sont triples. Premièrement, nous présentons une architecture logique supportant la programmabilité des mécanismes de sécurité dans un contexte multi-cloud et multi-tenant. Elle permet l'alignement et le paramétrage de ces mécanismes pour des services cloud dont les ressources sont réparties auprès de différents fournisseurs et tenants. Deuxièmement, nous introduisons une méthode de génération à la volée d'images unikernels sécurisées. Celle-ci permet d'aboutir à des ressources spécifiques et contraintes, qui intègrent les mécanismes de sécurité dès la phase de construction des images. Elles peuvent être élaborées réactivement ou proactivement pour répondre à des besoins d'élasticité. Troisièmement, nous proposons d'étendre le langage d'orchestration TOSCA, afin qu'il soit possible de générer automatiquement des ressources sécurisées, selon différents niveaux de sécurité en phase avec l'orchestration. Enfin, nous détaillons un prototypage et un ensemble d'expérimentations permettant d'évaluer les bénéfices et limites de l'approche proposée.

Mots-clés: Sécurité, Programmabilité, Cloud Distribué, Orchestration, Unikernel.

Abstract

In this thesis, we propose an approach for software-defined security in distributed clouds. More specifically, we show to what extent this programmability can contribute to the protection of distributed cloud services, through the generation of secured unikernel images. These ones are instantiated in the form of lightweight virtual machines, whose attack surface is limited and whose security is driven by a security orchestrator. The contributions of this thesis are threefold. First, we present a logical architecture supporting the programmability of security mechanims in a multi-cloud and multi-tenant context. It permits to align and parameterize these mechanisms for cloud services whose resources are spread over several providers and tenants. Second, we introduce a method for generating secured unikernel images in an on-the-fly manner. This one permits to lead to specific and constrained resources, that integrate security mechanisms as soon as the image generation phase. These ones may be built in a reactive or proactive manner, in order to address elasticity requirements. Third, we propose to extend the TOSCA orchestration language, so that is is possible to generate automatically secured resources, according to different security levels in phase with the orchestration. Finally, we detail a prototyping and extensive series of experiments that are used to evaluate the benefits and limits of the proposed approach.

Keywords: Security, Programmability, Distributed Cloud, Orchestration, Unikernel.