From virtualization security issues to cloud protection opportunities: An in-depth analysis of system virtualization models

Maxime Compastié, Rémi Badonnel, Olivier Festor, Ruan He

Version of Record: https://www.sciencedirect.com/science/article/pii/S0167404820301814
Manuscript_48115901a75561b1250dfb03ed832420

From Virtualization Security Issues to Cloud Protection Opportunities: An In-Depth Analysis of System Virtualization Models

Maxime Compastié[a,b,1,*], Rémi Badonnel[a], Olivier Festor[a], Ruan He[b]

[a] *University of Lorraine, CNRS, Inria, Loria, Campus Scientifique 54600 Villers-lès-Nancy*
[b] *Orange Labs, 44 avenue de la République, 92320 Châtillon, France*

Abstract

Virtualization methods and techniques play an important role in the development of cloud infrastructures and their services. They enable the decoupling of virtualized resources from the underlying hardware, and facilitate their sharing amongst multiple users. They contribute to the building of elaborated cloud services that are based on the instantiation and composition of these resources. Different models may support such a virtualization, including virtualization based on type-I and type-II hypervisors, OS-level virtualization, and unikernel virtualization. These virtualization models pose a large variety of security issues, but also offer new opportunities for the protection of cloud services. In this article, we describe and compare these virtualization models, in order to establish a reference architecture of cloud infrastructure. We then analyze the security issues related to these models from the reference architecture, by considering related vulnerabilities and attacks. Finally, we point out different recommendations with respect to the exploitation of these models for supporting cloud protection.

Keywords: Security Management, System Virtualization, OS-level Virtualization, Cloud Infrastructures, Unikernel

1. Introduction

System virtualization is an essential key to the development of cloud infrastructures and their services. 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 consumers [1]. The multiplicity of stakeholders questions the security 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 potential multi-tenancy requires the isolation of virtualized resources, (iii) the growing regulation with respect to data protection puts additional constraints on the CSPs to guarantee the confidentiality and integrity of resources [2]. Recently, important efforts have been focused on performance improvement, leading to the emergence of new virtualization technologies capable of reducing virtualized environment footprints. In particular, containerization methods move the virtualization layer from the OS-hardware border to the OS-application one, in order to share the OS kernel. Also, unikernel-based virtualization permits to decrease the complexity of virtual environments, by elaborating minimal operating systems specifically built for dedicated applications.

The development and deployment of these virtualization models are still in the early stage, while the technologies and related products are not fully mature. It therefore remains unclear what kind of specific security issues it introduces. We envision that the attack surface of virtualization could be impacted by the hypervisor isolation, cross-layer invocation and containerization. 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 misconfiguration 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 from different providers, the trust relationship between these components, either hardware or software, is hard to be established for the assessment of security properties. It is therefore challenging to securely and seamlessly incorporate these components.

Our contribution addresses the security of system virtualization models used for cloud infrastructures. We provide an in-depth description of the virtualization models to conduct a security analysis. Contrary to [3], we do not focus on the security question related to the operation of cloud environments. The analyzed virtualization models

*Corresponding author.
Email address: compastiem@yahoo.fr (Maxime Compastié)
[1]The permanent address of the author is at Activeeon, 36 rue Eugène Freyssinet, 75013 Paris

are not limited to full-fledged VM system virtualization, as considered in [4], but we also include OS-level virtualization and unikernel VM system virtualization. Our approach is close to [5], but our analysis is not bound to any cloud operation use-case. It rather permits to infer recommendations for our software-defined security strategy for distributed clouds, as elaborated in [6]. To evaluate the benefits and limits of virtualization models for cloud security, this article is organized as follows. First, we describe in Section 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 establish a reference architecture that synthesizes these models and serves a support for the security analysis. Second, we analyze in Section 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 Section 4 different counter-measures and recommendations with respect to the exploitation of these models for cloud security. This includes the integration of security mechanisms during the design of resources, the minimization of the attack surface based on the generation of highly specialized resources, and the adaptation of cloud protection based on security programmability.

2. System Virtualization Models

First of all, we will describe the considered system virtualization models, in order to establish a reference architecture of cloud infrastructure, that will serve as a basis for the security analysis. In that context, we will also remind some important principles related to system virtualization.

2.1. Context

According to [7], system virtualization can be defined as the use of an encapsulating software layer surrounding an operating system, which conforms to the behavior expected from a physical hardware. It is first envisioned as an enabler for multi-tasking: it has enabled hardware resource multiplexing, and thus, the concurrent execution of several OSHowever, more efficient approaches dealing with this issue 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 enables the usage of an appliance developed on one operating system with another one. The large-scale deployment of cloud computing during 2010s has induced a new application field. As the cloud paradigm requires the decoupling of software resources from hardware resources,

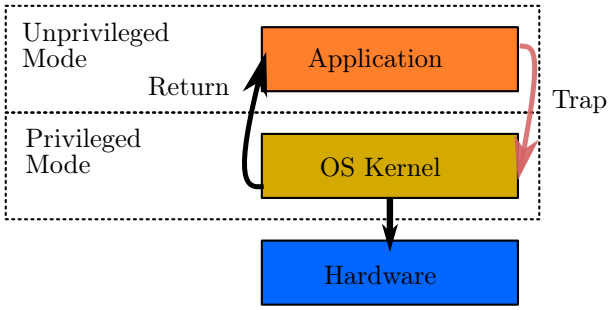

Figure 1: Overview of a regular system architecture.

system virtualization has become one of the enablers of this paradigm.

We usually define a regular system architecture according to a two-state mode execution environment (non privileged mode and privileged mode), as illustrated in Fig. 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 use the whole instruction set. When 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. 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 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 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.

2.2. System Virtualization

The system virtualization architecture derives from the regular system architecture. It enables concurrent execution of multiple OS. Within this architecture, two categories of instructions exist: sensitive and non-sensitive. Non-sensitive instructions can be executed directly on the hardware. However, sensitive need to be controlled since they modify the state of a machine which may impact a concurrent execution. The software module necessary to implement this virtualization architecture is called *Virtual Machine Monitor (VMM)*, whose properties are defined in [7]. A VMM provisions an abstracted and isolated environment for each OS, which is called *Virtual Machine*

(VM). It is a program interfacing between programs in each VM and hardware resources.

2.2.1. Fully and Partially Virtualizable Architectures

A *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 considered as fully virtualizable, if and only if all its sensitive instructions are part of the privileged instructions.Thus, intercepting the instructions from unprivileged mode permits to handle the execution of sensitive instructions, while the privileged ones can be controlled using traps. 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. For instance, the x86 architecture does not comply with the fully virtualizable architecture [8] since some instructions accessing sensitive registers are invokable from non-privileged mode. Such an architecture is called *partially virtualizable architecture*. It is then required to treat these non-virtualizable instructions with other mechanisms than instruction trapping. Three methods are usually considered to address this issue.

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. 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 to *hypercalls*. Considering the case of an OS in a virtual machine, the para-virtualization related code modifications are only located in the kernel of the VM, to make it cope with the VMM through hypercalls.The *hardware-assisted virtualization* consists in a solution based on CPU capabilities. More precisely, when hardware-assisted virtualization is available, the CPU proposes a *root operation mode* intended to host the VMM, while the *non-root operation mode* supports the 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 mode, will trap back.

2.2.2. 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 article, we refer to an 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. A *guest VM* stands for a VM with a restricted access to hardware resources. We distinguish two types of VMMs. The **type-I hypervisor** corresponds 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 allowed by the hypervisor to access the hardware without restriction and has administrative privileges over the hypervisor, while the guest VMs are handled by the virtual hardware environment. The Xen hypervisor [9, 10] and the KVM hypervisor [11] 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 [12] and QEMU [13] are two well-known examples of type-II hypervisors.

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 by 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 the user space (system calls) [15]. The host OS kernel is in charge of required virtual resources (i.e. filesystem access, networking) and container isolation enforcement. The container 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 [16, 17] performances, outrunning the regular system virtualization ones, at the cost of coupling containers with a specific operating system. LXC [18], Docker [19] and gVisor [20] are three thriving examples of container engines supporting the Linux OS.

2.4. Unikernel Virtualization

Containerization has brought an application-centric perspective in the virtualization debate, as the applications

[2]OS-level virtualization is also referred as *containerization*. However, some hypervisor-based solutions such as [14] have started as well to claim this terminology. For the sake of clarity, we dismiss its usage in this article.

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 constraints, it enables the refining of the system layer, reducing footprints of virtualized applications.

2.4.1. 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 any 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 contribute to their efficiency. However, the applications are not likely to implement all of them to support all different types of hardware resources. The Drawbridge [21], the V++ cache kernel [22] and the Exokernel [23] 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 exploits 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 [23], ExOS [23] and XOmB [24].

2.4.2. Unikernels

A unikernel corresponds to an OS architecture featuring a specialized, sealed, single purpose library OS. They can be run in virtual machines on the top of a hypervisor [25]. Each unikernel system addresses an application, its configuration, and the minimal dependencies required to run, and is configured and packed into an image, before their instantiation. Running inside a VM relieves these 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 contributes to lightweight unikernel VMs compared to regular VMs. MirageOS [25] and IncludeOS [26] are two examples of common unikernel solutions.

A single unikernel image 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 may be tightly related to a development platform or may remain independent. The first option requires the usage of a dedicated tooling and programming language to design 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 performance and lightweight properties 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, it is 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 permits to cope with the overheads of context switches issued by process scheduling and system calls. 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 relies on the inclusion of hypervisor-specific hardware resource management routines into unikernel images, in order to address a specific virtual hardware environment. In the case of the exploitation of unikernel VMs, type-I hypervisors are usually privileged to minimize the layers mitigating the unikernel VM access to hardware resources. The routines of the unikernel VMs are executed in the privileged mode to avoid the trapping overhead, when accessing the virtual hardware environment.

2.5. Synthesis

We have presented different virtualization models, including virtualization based on hypervisors of type-I, virtualization based on hypervisors of type-II, OS-level virtualization and unikernel-based virtualization. We infer a virtualization reference architecture, depicted on Fig. 2,

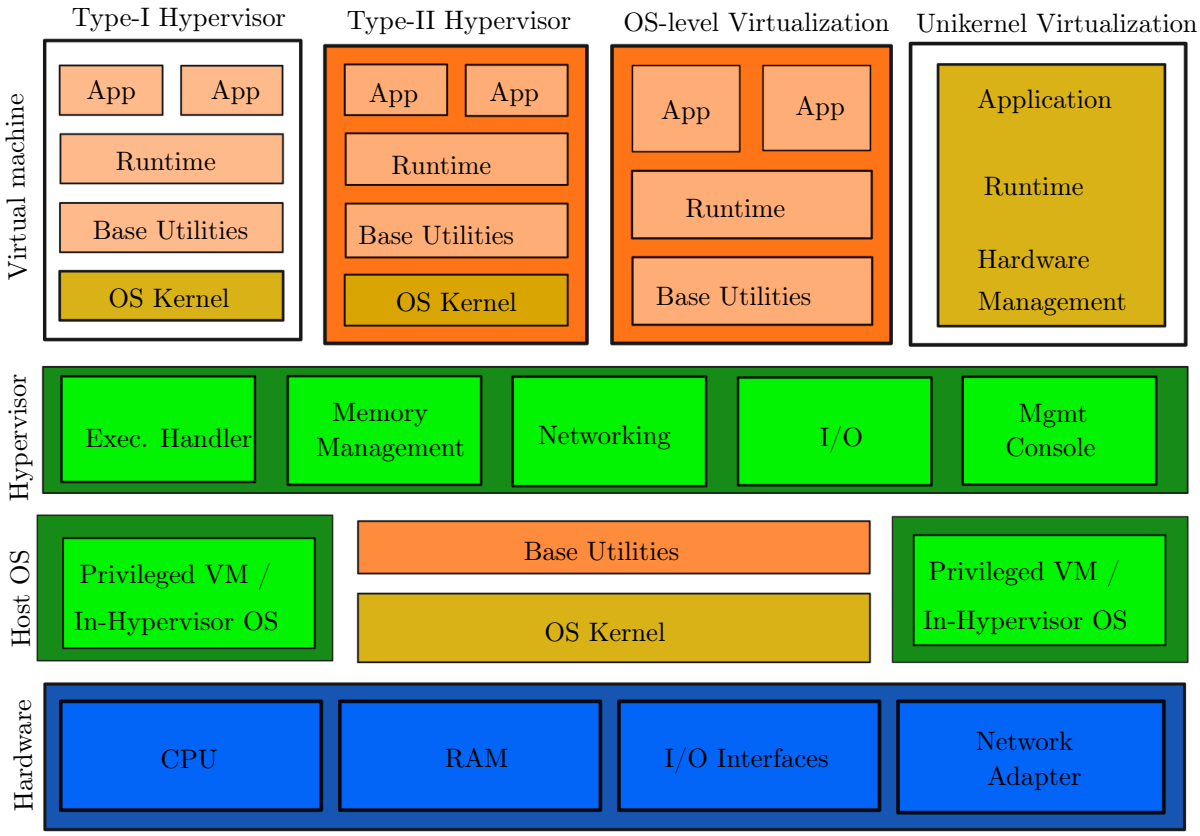

Figure 2: Virtualization reference architecture.

that synthesizes these virtualization models. 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 resources composing the host machine, such as the CPU, the volatile memory (RAM), the I/O interfaces (for persistent storage and extension cards) and the network adapter for networking purposes.

- The **host OS level**, detailed at the homonym 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 used by these VM softwares. Type-I hypervisor and unikernel-related hypervisor may completely manage themselves hardware resources, or delegate this management to privileged VM (e.g. storage or network device handling).

- The **hypervisor level**, mentioned in the figure in the next row, refers to hypervisors and container engine softwares. We assume it handles VMM capabilities (VM instruction trapping and sensitive instructions handling), VM memory management, VM networking (virtual network adapter and inter-VM networks) and VM I/O interfaces. The hypervisor is configured 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 corresponds to the top row of the figure. This level includes applications, their configurations, their runtime environment and the libraries they depend on. Utilities are not systematically embedded in unikernels, since their features are embedded as application libraries, when they are required. In the same manner, OS kernel is neither integrated in containers (containers rely on host OS kernels) nor in unikernels (the OS kernel is reshaped in a set of libraries, provided with the application).

Virtualization provides important properties with respect to security [4], 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 capabilities to virtualization. Moreover, by allocating quotas to the physical resources to VMs, virtualization also promotes resource consumption isolation.

- **Oversight/introspection:** the hypervisor is able to inspect a VM resource usage and thus observe its internal state, leading to the oversight capability. As

the hypervisor is also in charge of VM resource allocation, internal state modifications may be performed, defining the introspection capability.

- **Snapshotting:** the hypervisor enforces access control amongst a VM and physical resources. This positioning enables 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. From a security perspective, this feature permits reversing back a VM from a given insecured state to a previous secure one.

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

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 vulnerabilities (in terms of occurrences) with respect to virtualization models. We then determine the threats affecting the architecture, and analyze related attacks in view of identified vulnerabilities.

3.1. Identification of Vulnerabilities

We first identify and classify vulnerabilities related to the reference architecture, as depicted on Fig. 3. The criticality of these vulnerabilities with respect to virtualization models is synthetically exposed in Table 1, and is justified with the presentation of vulnerabilities. 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 the following four main categories: the VM application, the VM guest OS, the hypervisor, and the execution environment of the hypervisor.

3.1.1. VM Applications

The vulnerabilities of VM applications are typically related to memory management (with respect to runtime variable type checking, memory deallocation, kernel inference in user space, and developpement software flaws), and to 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 [27, 28] 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 environments equipped with a memory garbage collector, no memory space is ensured to be automatically deallocated when stopped being used. This may lead to process memory leaks [27, 28, 29], and data persistence issues [30]. 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 of unikernels leaves physical memory cleaning to the hypervisor at the application termination. **Kernel inferences in userspace** may also be possible. The OS kernel is in charge of system management, by providing routines that are used by application processes. Nevertheless, the kernel is also able to manage this address space for various purposes, such as process swapping, free space reclamation [31], and exception handling. All virtualization models are exposed to this vulnerability. While it is obvious in a regular system architecture, unikernels rely on a built-in runtime for the whole support of an application. Finally, we should remind that any program comes with **software flaws** and bugs issued from their development cycles, inducing new exploitable vulnerabilities [32]. This echoes to the unikernel pleading to limited code base, while OS-level containers benefit from not embedding an OS kernel.

Software Interfaces. In VMs, the software programs rely on interfaces to communicate with the users and other programs. These ones are able to send 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 privilege assignment (configuration) or weak authentication [33, 34]. While virtualization models exploiting regular system architectures have to carry multiple applications and base utilities with their own interfaces, unikernel models only expose the 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

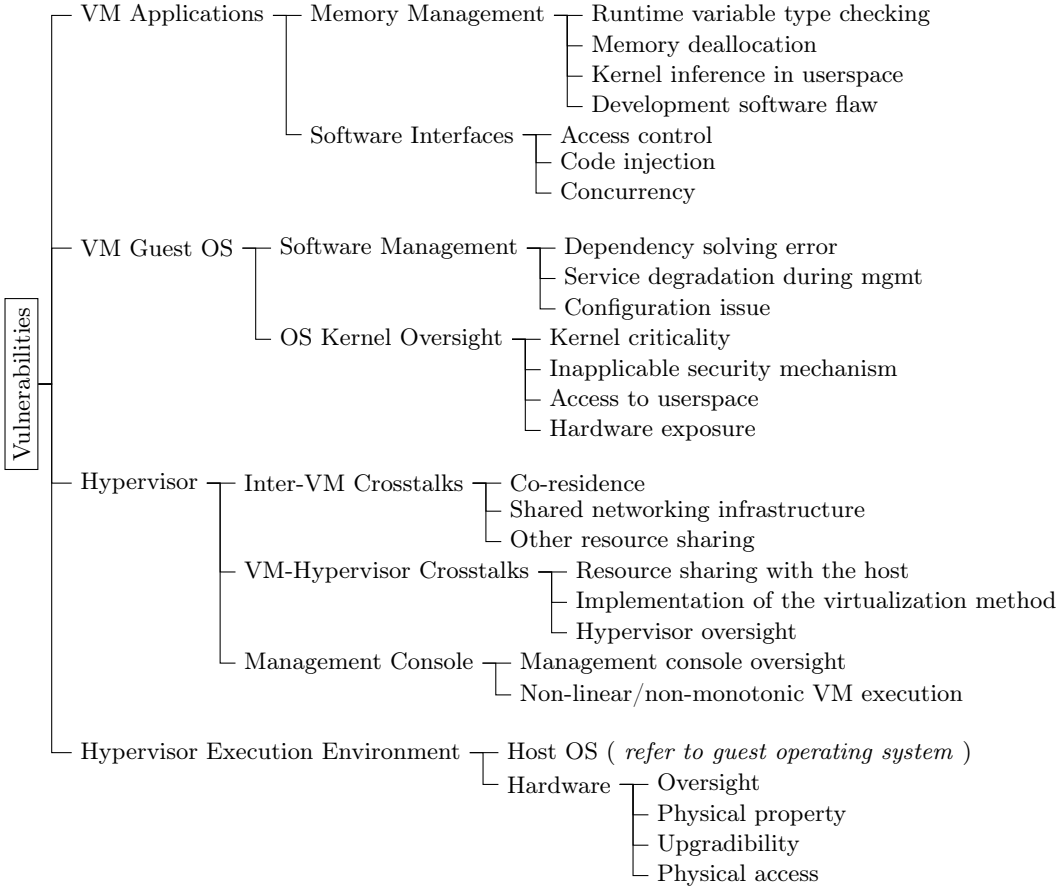

Figure 3: Classification of considered vulnerabilities.

exploitation of inconsistencies in data structures [35, 36], non-controlled string format exploitation [37] and, more directly, **code injection** [38, 39]. When an interface can be accessed by several programs at a time, synchronization mechanisms are mandatory. If not, this interface may be concerned by **concurrency** vulnerabilities [38, 40]. As all virtualization models support multi-threading in their virtualized environment, they are all equally affected by these vulnerabilities.

3.1.2. VM Guest OS

The vulnerabilities of the VM guest OS concern the software management (including dependency solving, service degradation and configuration issues) and the OS kernel 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 [41]) or by a dedicated installer (E.g. 0install [42]) especially shaped for dedicated appliances. Nevertheless, **solving dependency** may generate vulnerabilities [43]. This particularly affects virtualization models supporting legacy system architectures. Software management can be performed at 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 uses a pool of software components provided by the unikernel itself, and is performed only at build time [25]. Besides, software management processes may lead to the **degradation of services**, as the upgrade usually requires to restart and reload configurations [44]. This can be exploited to make the service unavailable during a certain time period. In all the considered virtualization models, this vulnerability can be mitigated by performing upgrades on virtual environments that are not in production. This mitigation is usual 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 [45]. 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

Table 1: Exposure to considered vulnerabilities with respect to virtualization models

	Type-I Hypervisor	Type-II Hypervisor	OS-level Virtualization	Unikernel Virtualization
Runtime variable type checking	●	●	◖	○
Memory deallocation	●	●	●	◖
Kernel inference in userspace	◖	◖	◖	◖
Development software flaw	●	●	◖	○
Access control	●	●	●	○
Code injection	●	●	●	○
Concurrency	◖	◖	◖	◖
Dependency solving error	●	●	●	○
Service degr. during mgmt	◖	◖	◖	○
Configuration issue	●	●	○	○
Kernel criticality	◖	◖	○	●
Inapplicable sec. mech.	●	●	○	○
Access to userspace	●	●	○	●
Hardware exposure	●	●	○	◖
Co-residence	◖	◖	●	○
Shared networking	◖	◖	◖	◖
Other resource sharing	◖	◖	◖	◖
Resource sharing with the host	○	○	●	○
Implem. of virtualization method	◖	◖	●	◖
Hypervisor oversight	◖	◖	●	◖
Management console oversight	◖	◖	◖	◖
Non-linear/non-monotonic VM exec.	◖	◖	◖	◖
H-OS - Dependency solving error	◖	●	●	◖
H-OS - Service degr. during mgmt	◖	●	●	◖
H-OS - Configuration issue	◖	●	●	◖
H-OS kernel - Kernel criticality	◖	●	●	◖
H-OS kernel - Inapplicable sec. mech.	◖	●	●	◖
H-OS kernel - Access to userspace	◖	●	●	◖
H-OS kernel - Hardware exposition	◖	●	●	◖
Hardware oversight	◖	◖	◖	◖
Hardware physical property	◖	◖	◖	◖
Hardware upgradibility	◖	◖	◖	◖
Hardware physical access	◖	◖	◖	◖

(Notation: ○, ◖, ● stand respectively for low, medium and high.)

absence of an 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** for a 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 [46]. These vulnerabilities are bounded to the resilience of OS kernels and their 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 [29]. Unikernel is one known exception, as its design is based on a common framework for both the development of hardware resource management and applications [47]. 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 another source of vulnerabilities. For instance, the 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 [29]. 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 the 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.

3.1.3. 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 thus not able to access the resource, or communicate with another one, 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** [48]. 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 [49]. 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 [50]). 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 VMs may be an instance of a common image issued from a registry [51]. 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 VMs. Hypervisors and VMs communicate through interfaces. The virtual hardware environment 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 [52, 53, 54]. Hypervisors may also provide private communication channels (with their VMs), that may not enforce proper security checks [55]. 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 [56]. It may also affect their implementations [57]. 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 a 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 [58]. 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 configuration 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 as cloud orchestrators.The first vulnerability relates to the **oversight based on the management console**. The users of the management console have a complete control over VM life-cycle, 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 [59]. 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 [4, 60]. 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.

3.1.4. Execution Environment of the Hypervisor

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

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 also affected by these vulnerabilities, as it is an application supported by a host OS. We consider that host OS vulnerabilities are the same than the ones detailed in Section 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 VMs, and their own storage. This makes hardware an attractive target to compromise VMs. The **physical properties** of hardware can be exploited as side channel sources [61]. 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 [62]. 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.

3.2. Considered Threats and Attacks

We exploit the STRIDE threat assessment methodology [63] to analyze the different threats and attacks related to our reference architecture. In accordance with it, we consider a set of six main threats, namely spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privileges. In that context, we assume several hypotheses regarding this 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 negatively impact the availability (denial of service), the integrity (alteration) or the confidentiality (retrieving) of an asset.
- A component is considered secured if not compromised, but which may contain inherent software flaws known by the attacker. The attacker can exploit them to compromise a component, and gain influence or control over it.

We classify attacks with regard to threats, according to the notion of compromises: we consider a component as compromised, when the attacker is capable of executing an arbitrary code. We introduce a three-level classification based on the extension of [3]: compromise-free attacks (that are not related to compromises), compromising attacks (that enable compromises), and compromise-based attacks (that are based on compromises). Fig. 4 provides an overview of this classification, while Table 2 details the relationships with the threats with regard to the reference architecture.

3.3. Compromise-Free Attacks

We first detail compromise-free attacks, 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.

3.3.1. 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 [59]. A more discrete 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) 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 unavailable. This can typically consist in a distributed denial of service attack. Software flaws are also sources of software failures (e.g. [64] 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 [65], linux kernel faults are mainly due to unsecure software development.

We then focus on attacks exploiting **software flawed 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 [66, 67].

Table 2: Relationships among attacks, leveraged threats and affected components

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

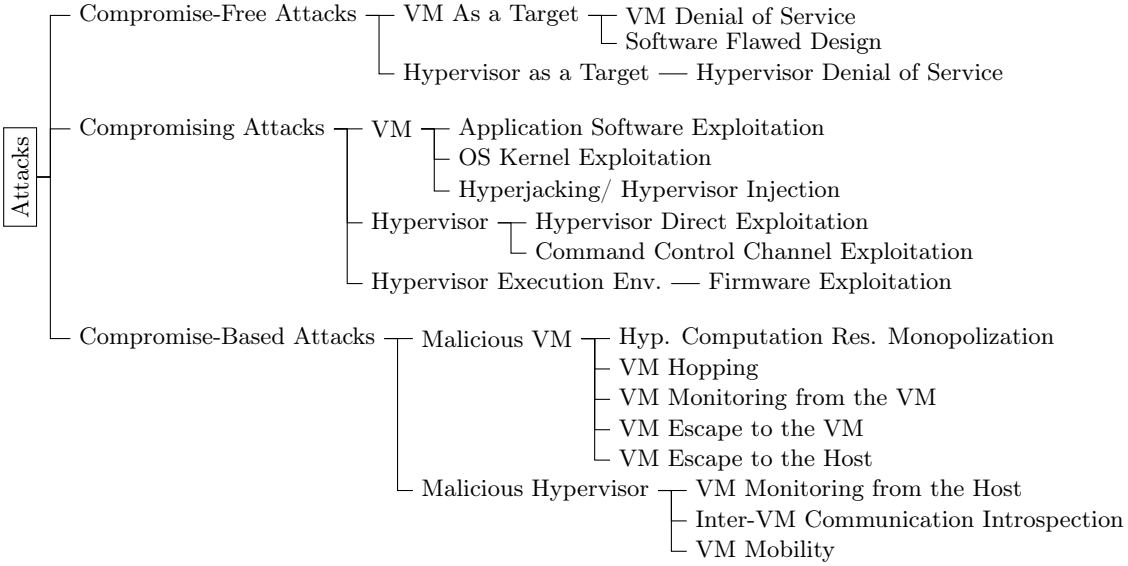

Figure 4: Classification of attacks.

They may be 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 some VM appliances, can also introduce vulnerabilities. The upgrade process can also lead to man-in-the-middle (MITM) attacks during the collection of packages, as pointed out in [68, 69]. 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* vulnerabilities. As exposed in [4], an attack might restore a VM to a vulnerable state, by using the hypervisor facilities.

3.3.2. 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 denials of service. For instance, authors of [4] pointed out VM sprawling flaws, enabling the rapid spread of misbehaving VMs. This leads to the exhaustion of the hypervisor resources.

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.

3.4.1. 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 [70], the *code injection* in software interfaces may also be used for the insertion of payloads. The *dependency solving error* [71] 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 [72] show that 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 works in [68, 69] also investigate a MITM attack against the software management system, with the objective of inserting or maintain-

ing 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 benefit from the privileged execution mode, which extends the attack surface. A concrete example of such an attack is the use of rootkits [73], 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 process loading modules 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 [74]. Its introspection capability enables it to overpass any security mechanisms [4]. King et al. propose in [75] a persistent VMBRs targeting both Windows OS and Linux, and hosting malicious services undetectable from these OSes. The blue pill attack, described in [74], is a VMBR attack capable of infecting a host without any reboot. The VMBR attack takes advantage from the *hypervisor oversight* vulnerability to get introspection capabilities, and to be undetectable by OS-level detection mechanisms.

3.4.2. Hypervisor

The compromising attacks may also target the hypervisor of the virtualization architecture. Amongst these attacks, we can highlight **hypervisor direct exploitation** attacks [76]. Even if they are not expected to be exposed as much as their VM appliances, 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 [77] illustrates the Xen hypervisor (and the privileged domain) compromise by a rootkit using a weakness in the DMA implementation and the Xen debug register. [78] 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 [78], 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 [79]. This attacks targets the hypervisor by using this channel as a medium to compromise it. The work in [80] investigates 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.

3.4.3. 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 [81], in order to compromise the firmware of network interface cards. 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.

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.

3.5.1. 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 [82]. 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.

Table 3: Relationships among attacks and vulnerabilities

	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 Res. 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 flaw	●	●		●													
Access control	●	●									●						
Possible code injection	●			●							●						
Concurrency vulnerability	●										●						
Dependency solving error		●		●													
Service degradation during mgmt																	
Configuration issue		●		●							●						
Kernel criticality	●																
Inapplicable security mechanisms	●				●												
Access to userspace	●			●													
Hardware exposition	●										●						
Co-residence										●		●					
Common networking infrastructure			●								●	●					
Other resource sharing			●								●	●					
Resource sharing with the host			●				●	●					●	●	●	●	
Virtualization method implementation							●			●		●	●	●			
Hypervisor oversight						●							●	●	●	●	
Management console oversight	●		●														●
Non-linear/monotonicity VM execution		●															●
Host OS - Dependency solving error																	
Host OS - Service degradation during mgmt																	
Host OS - Configuration issue																	
Host OS kernel - Kernel criticality			●														
Host OS kernel - Inapplicable security mechanism			●														
Host OS kernel - Access to userspace			●														
Host OS kernel - Hardware exposition			●														
Hardware oversight									●								
Hardware physical property												●					
Hardware upgradibility									●								
Hardware physical access																	

The **VM hopping** [83] 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. They 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. The work in [84] illustrates the by-passing of an OS protection based on hypervisor side channels. The work in [85] 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 [48] details the steps to reach a VM co-residence with the targeted VM. Finally, authors of [86] explore how co-residence can contribute to cryptography key leaks based on CPU L1-cache. These attacks are based on the *hardware physical property*, the *virtualization method implementation* and the *inter-VM crosstalks* vulnerabilities.

The **VM escape to a VM** attack [4] 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 to compromise the hypervisor, such as the VENOM [87] and the Cloudburst attack [54]. These attacks take advantage of the *VM-hypervisor crosstalks* vulnerabilities.

3.5.2. 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 and to infer its internal state. Different introspection techniques are presented in [58] to track the activities of a virtual machine. For instance, [59] demonstrates an approach for extracting a secret key from a memory dump. Authors of [88] 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 [59], 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 informations 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 informations. The *hypervisor oversight* and *resource sharing with the host* vulnerabilities contribute to these attacks. The **VM 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.

Table 3 describes the relationships among the vulnerabilities of the reference architecture and the attacks that they leverage.

4. Counter-Measures and Recommendations

After having classified security attacks, we propose different counter-measures and recommendations with regard to the reference architecture. Therefor, we consider two major requirements for counter-measures. To limit their cost on resource addressing protection, these ones should have a minimal impact on the operability of protected resources. Moreover, to ensure that counter-measures contributes to the protection effort, they should not impact on the security benefits brought by the virtualization itself.We provide a threefold classification for these counter-measures. Fig. 5 describes this classification, while Table 4 exposes the benefits and limits of identified counter-measures in terms of coverage, requirements and costs. Table 5 specifies the relationships among threats and counter-measures/recommendations. This classification includes counter-measures consisting in integrating security mechanisms at the design of resources, counter-measures aiming at reducing the attack surface of resources, and counter-measures enabling a higher adaptation through security programmability.

4.1. Integration of security mechanisms at design time

A first category of counter-measures consists of 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.

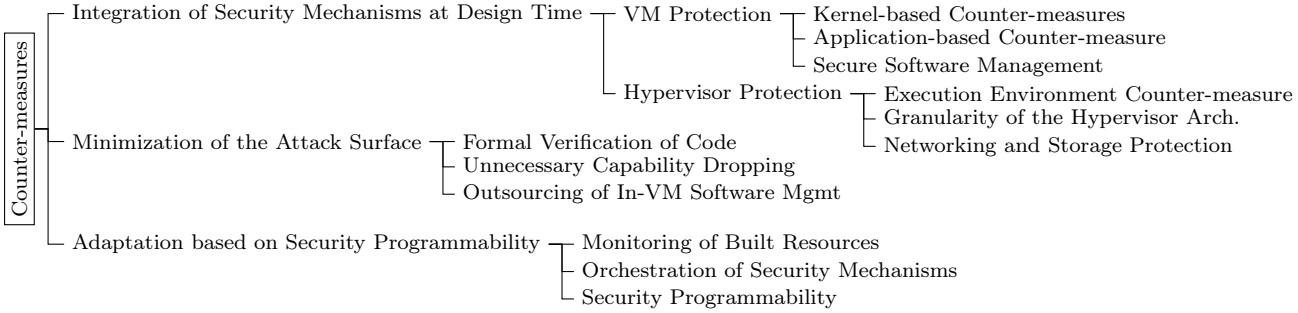

Figure 5: Classification of Countermeasures (or Recommendations).

4.1.1. 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 [89]. From an architectural viewpoint, monolithic OS kernels may facilitate security attacks due to the lack of isolation amongst kernel subsystems. PerspicuOS [90] 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. However, the ability to enforce kernel-based counter-measure is subjected either to the existence of relevant implementations in the OS kernel, or the ability of the latter to be extended. This statement jeopardize the security of operating system restricting the access to their programming interfaces. 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 our analysis. Without constrains on the runtime environment, this often supposes specialized security mechanisms adapted to each applications. This comes with the burden to secure a more important code base. In the meantime, the protection of applications from untrusted execution environments can be addressed generically. OS-based solutions such as the XOMOS OS [91] are part of the solution to this issue. Shielding the application execution is also an interesting approach to protect applications from a malicious OS. For instance, Haven [92] exploits the Intel SGX technology to provide a protective layer against OS-based and physical attacks. However, both of them comes with heavy requirements on the the hardware operating the application. Therefore, constraining the application runtime is a reasonable trade off to enable a extensive protection on application with a limited requirements on the platform. The usage of unikernels constitutes a solution to have a minimal code base and to 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 secured and may contribute to prevent attacks on virtual machines. Approaches such as [69] introduce package signature mechanisms to deal with man-in-the-middle attacks, as well as package alterations. Complementarily, [71] exploits solving techniques for the satisfiability problem (SAT) 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. But, these counter-measures are firmly bound to the usage of a package manager implementing these counter-measures.Any software management operation made outside this framework put at stake the security of the system.

4.1.2. Protection of the hypervisor

The hypervisor provides an **execution environment** to the VMs, enabling them to run in accordance with the VMM properties [7]. In that context, [93] 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, [59] introduces a method to protect the hypervisor from host VMs by filtering management hypercalls and checking their integrity at the invocation. Virtual CPU context integrity checks as well as virtual memory encryption are enforced during the execution of the hypervisor. The overhead affecting the management operation can be more remarkable on VMs with a short lifespan. In the area of OS-level virtualization, the Intel safeguard extension is used by SCONE [94] to protect containers against untrusted execution environments. This framework provides the necessary building blocks to design enclaved linux containers, that are re-

Table 4: Pros and Cons Comparison Table with Respect to Counter-Measures

Counter-measures	Coverage	Requirements	Cost
Kernel-based Counter-measure	OS-Kernel, Host-OS Kernel	Existence of mechanism implementations for the operated kernels	Medium
Application-based Counter-measure	Application, Runtime	Existence of mechanism implementations for all the operated applications and their runtime	Medium
Secure Software Management	OS-Kernel, Application, Runtime	Software management only made in secure package managers	Medium
Execution Environment Counter-measure	Hypervisor, Host-OS Kernel	Existence of mechanism implementations in the operated hypervisors	Low
Granularity of the Hypervisor Arch.	Hypervisor	Modular architecture of the hypervisor	Low
Networking and Storage Protection	Hypervisor	Exposition of storage and networking interfaces by the hypervisor	Low
Formal Verification of Code	OS Kernel, Hypervisor, Host-OS Kernel	Limited code base, design and maintainability of the formal proof	High
Unecessary Capability Dropping	All Software Components	Modular design and/or support for disabling features at runtime	Medium
Outsourcing of In-VM Software Mgmt	Application, Runtime, OS Kernel	Short-living VM instance, secured image build environment	Low
Monitoring of Built Resources	Application, Runtime, OS Kernel	Existence of interfaces to monitor and semantic knowledge of monitored data	Medium
Orchestration of Security Mechanisms	Application, Runtime, OS Kernel	Specification of security requirements and mechanism features	Low
Security Programmability	Application, Runtime, OS Kernel	Exposure of reconfiguration interfaces, or possibility to restart components	Medium

*The cost category evaluates qualitatively the counter-measure configuration process: once during virtualization environment set-up (**low**), once per VM image building (**medium**), or at each software update (**high**).*

silient against tampering attacks from the OS, with a limited trusted computing base (TCB). The system calls are performed asynchronously with the assistance of a module outside the enclave, and therefor requires a modification of the host OS. Complementary, the exploitation of command channels can be avoided by obfuscation methods, as detailed in [80], but this mitigation degrades the functionality of the hypervisor. [82] also describes a solution to CPU monopolization attacks, by switching virtual CPU scheduling to alternative methods (e.g. exact scheduler, Bernouilli scheduler and uniform scheduler) to limit flaw exploitations. This counter-measure can heavily impact VM usage, and the scheduling method should be assessed to meet operationnal constrains (e.g. service level agreement).

The **granularity** of the hypervisor architecture also impacts on security. In order to avoid a monolithic architecture, [95] 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 [96] 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 [97]. 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 however limited to access control enforcement with the MAC model on a single hypervisor instance. TVDSEC [98] 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. For instance, the NICE framework [99] exploits a network controller to put in a quarantine state VMs that are potentially compromised. This framework is however only compatible with Openflow-configured networks.

The integration of security mechanisms at design time

is an important topic that induces several research challenges in terms of heterogeneity and distribution. The identified counter-measures clearly highlighted this heterogeneity of resources requiring security mechanisms and the interfaces to be used for their configuration. While the light-weight architecture of unikernels may facilitate the integration of security mechanisms, we believe that other virtualization models could benefits of on-going research efforts in the system management area. In particular, the research efforts currently done in building and developing Infrastructure-as-Code solutions [100] could contribute to the design of security-constrained applications with complex runtimes in other virtualization models. In addition, we think that orchestration languages, such as the Topology and Orchestration Specification for Cloud Applications (TOSCA) or the OpenStack Heat Orchestration Template (HOT), constitute interesting supports to be extended in order to facilitate this integration at design time. They permit to specify and orchestrate cloud services whose resources may be deployed over several infrastructures, considering potential inter-tenant and inter-cloud collaborations. The services are described in the form of topologies with a set of resources and their relationships. It is possible to specify orchestration processes related to a service, these ones impacting on the state of resources and relationships. Research efforts should be considered to enable the specification of security requirements with different security levels that could be associated to orchestration and instanciation processes, and the extension of these languages for defining and automating the integration of security mechanisms, when elaborating the services.

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. [101] introduce 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 [102] project proposes an OS kernel that can be assessed using the Z3 satisfiability modulo theories (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, the virtual ghost framework [103] permits 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. The protection against control flow injections is established at the application compilation time by LLVM intermediate code inspections. Formal verification methods are also applied to check the hypervisors. For instance, work in [104] has formally specified parts of an hypervisor based on Xen. But the authors acknowledge that this approach is too expensive to be applied to an entire product.

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 imposing a system to restrict parameter values and use specific (security) software components. Most applications come with their very specific recommendations for hardening. This requires a specific knowledge of the applications to be protected while making configuration with multiple applications and runtimes complex to be extensively handled. 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 [105], facilitating the usage of constrained and ephemeral virtualized resources. This approach is suited only for virtualized resources with featuring a very short boot time, excluding other virtualized resources than unikernel. 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 [106], 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 been performed on other hypervisors than KVM.

The **outsourcing of in-VM software management** contributes also to the code base limitation efforts in virtualized environments. Most operating system images are provided with their software management tools. Therefore, the presence of these tools in the VM images can be questioned, as most of the software management operations are performed before the operation of applications. Delegating the software management to the image manufacturing process contributes to protect VM applications against related attacks and compromises. An example of such outsourcing is given in [69], in accordance with security criteria. However, this approach supposes that (i)

Table 5: Relationships among Threats and Countermeasures (or Recommendations)

	Application	Runtime	OS Kernel	Hypervisor	H-OS Kernel
Kernel-based Counter-measure			T,I,D,E		T,I,D,E
Application-based Counter-measure	S,T,R,I,D,E	S,T,R,I,D,E			
Secure Software Management	S,T,D	S,T,D	S,T,D		S,T,D
Execution Environment Counter-measure	D	D	D	T,R,I,E	
Granularity of the Hypervisor Arch.				D,E	
Networking and Storage Protection	T,I,E	T,I,E	T,I,E	T,I,E	
Formal Verification of Code	T,R,I,D,E	T,R,I,D,E	T,R,I,D,E	T,R,I,D,E	T,R,I,D,E
Unecessary Capability Dropping	T,I,E	T,I,E	T,I,E	T,I,E	
Outsourcing of In-VM Software Mgmt	T,I,E	T,I,E	T,I,E		
Monitoring of Built Resources	S,T,R,I,D,E	S,T,R,I,D,E	S,T,R,I,D,E	S,T,R,I,D,E	S,T,R,I,D,E
Orchestration of Security Mechanisms	S,T,R,I,D,E	S,T,R,I,D,E	S,T,R,I,D,E	S,T,R,I,D,E	S,T,R,I,D,E
Security Programmability	S,T,R,I,D,E	S,T,R,I,D,E	S,T,R,I,D,E	S,T,R,I,D,E	S,T,R,I,D,E

*Notation: **S**poofing, **T**ampering, **R**epudiation, **I**nformation Disclosure, **D**enial of Service and **E**levation of Privilege.*

the manufacturing chain of VM image 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.

The minimization of the attack surface also poses important research challenges in terms of adaptation to new hardware facilities and in terms of automation for resource (re-)generations. These efforts are of course constrained by the hosting environment providing the applications and by the services to be provided. These constraints define a baseline that is specific to both the infrastructures and the applications to be operated. For instance, the new opportunities offered by hardware acceleration, in particular the exploitation of Graphical Processing Units (GPU) for virtualization, is challenging this minimization. The application has to handle the specificities of GPU processing, leading to more heavy virtual machines. Moreover, new solutions have to be investigated to make sure the hypervisor properly control the access to the GPU, considering the multiple competing and vendors-specific solutions [107] to implement GPU virtualization. Another major issue is to automate the generation of virtualized resources with a minimal attack surface. 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. New management frameworks and algorithms are required to generate these virtualized resources, including unikernels but not limited to them, in an on-the-fly manner. This automation will provide a better and dynamic adaptation to contextual changes, and may also contribute to moving target defense strategies.

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 [108] 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, but require minor extension of the hypervisor to work. Authors of [109] propose similar techniques, but applied from the guest OS during the execution phase. The proposed solution relies on a whitelist to identify authorized applications, making it complex to set up in virtual environments operating multiple applications. Active monitoring approaches, such as the *LARES* architecture [110], are also applied to observe virtual machines based on dedicated agents or probes. This approach requires to modify accordingly the monitored resources. The monitoring from hypervisor also enables the detection of compromised OS-kernel. Approaches such as the NICKLE framework [111], permit to detect rootkits in the OS kernel by operating a dedicated hypervisor. Finally, the hypervisor introspection is also applicable to VM virtual hardware environments, as developed by Slick [112] and TVDSEC [98]. 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. This approach permits to detect compromised VMs only when malicious interactions with external resources are initiated. The monitoring may also consist in the identification of vulnerable configurations related to components of the virtualization architecture. For instance, the approach in [67] proposes a solving method exploiting the satisfiability problem (SAT) for supporting resource management. It permits to detect the presence of configuration vulnerabilities in a preventive manner, considering the maintenance operations that are available. The database of the the vulnerabilities has to be provided and be adapted to the configurations of inspected VMs.

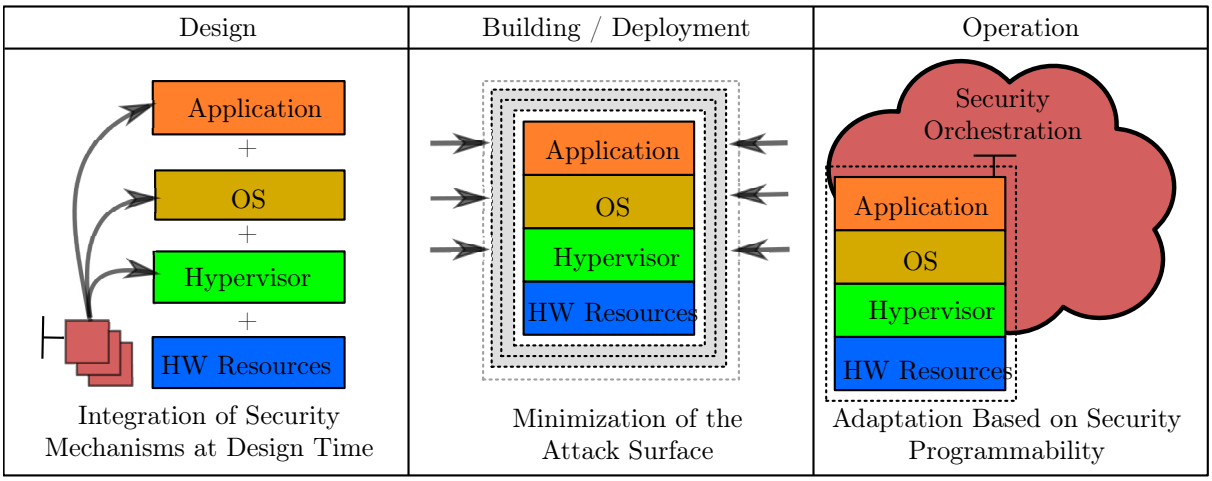

Figure 6: Illustrated Synthesis of Recommendations with Respect to Cloud Protection.

The analysis of monitoring results can then drive the **orchestration** of security mechanisms in virtualization and cloud environments. An efficient and consistent orchestration is an important aspect to deploy and coordinate security mechanisms. Some efforts, such as [113], exploit orchestration languages to support access control policies, specifically in the area of virtualized network functions (VNF). Policy mining methods may also contribute to assess the consistency of orchestration activities, by extracting high-level policies from entities enforcing low-level ones. Work in [114] applies this approach to extract network-wide access control policies from the configuration of multiple distributed firewalls.

Orchestration activities depend on the **programmability** of resources and of 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 [115] using IncludeOS unikernels. Therefore, the orchestration has to acknowledge the resource of the resources to be reconfigured to know which reconfiguration strategy is to be employed. We can also notice some recovery techniques, such as the TASER intrusion recovery system [116], capable of tracking the activity of a virtual machine, and recovering its configuration to a clean state after an attack. Nonetheless, this strategy requires to adapt the virtual machines by inserting a surveillance agent.

Security programmability offers new opportunities to protect resources, but also introduces new research challenges, in particular in the context of the permanently-growing Internet-of-Things and the development of services at the edge. In that context, research efforts are required to elaborate new strategies and algorithms taking into account the complementary of endogeneous and exogeneous security mechanisms. Endogeneous mechanisms consist in reducing the exposure to attacks by modifying internally the resources. They consist in hardening the configuration of virtualized resources at different levels (as previously presented), and may be driven by multiple security information data sources, such CVE[3] and OVAL[4] definitions, in order to prevent the exploitation of vulnerabilities. These mechanisms may not be fully implementable in the case of scarce-resource devices, such as the ones of the Internet-of-Things. Exogeneous mechanisms consist in complementing the resources based on external security functions. The programmability of network resources brings flexibility to the deployment and adaptation of such mechanisms. In particular, it enables to dynamically building and orchestrating chains that are composed of different security functions. These security functions may typically include firewalls, intrusion detection systems, and data leakage prevention mechanisms. They may be provided as middleboxes potentially deployed at the edge, or be directly implemented on the software-defined networking layer. Important efforts are required to support an adequate usage of both endogeneous and exogeneous security mechanisms in the case of Internet-of-Things devices, this including off-loading management methods as well as verification techniques for checking the consistency of these mechanisms.

[3]Common Vulnerability Enumeration
[4]Open Vulnerability and Assessment Language

5. Conclusions

System virtualization is both a source of vulnerabilities and an enabler for supporting the protection of cloud infrastructures and services. In this article, we have described different virtualization models, namely virtualization based on type-I hypervisors, virtualization based on type-II hypervisors, OS-level virtualization and unikernel virtualization. We have detailed their design principles as well as their 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 finally highlighted several counter-measures and recommendations with respect to their exploitation for cloud protection. Those counter-measures have been selected with the treats impacting virtualization architect in a cloud infrastructure. These recommendations are synthesized in Fig. 6, in line with different phases related to the life-cycle of cloud resources. Our work has highlighted the insufficiency of architectural counter-measures in hypervisor and virtual machines design. It pleads for a more integrative virtualized resource preparation before their allocation taking into account security requirements. We also argue in favor of the simplification of the system architecture to dwarf their attack surface and alleviate their management burden. Finally, deployed security mechanisms should be actively managed according to the security context of the protected resources and its evolution all along the exploitation. Those recommendations apply to the protection of the multiple assets involved in operation of a cloud service, and can comply with very specific security requirements.

Amongst the analyzed virtualization models, unikernels offer interesting 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. In addition, the programmability of resources, in phase with the security orchestration, can drive the generation of such unikernel resources in the context of cloud environments. The implementation of this orchestration depends also on the proper specification of security requirements regarding the security features and rules to be considered. Research efforts in the area of policy-based management should take benefits from the orchestration languages and exploit them as a support to integrate security mechanisms at design time. The maturity of technologies and tools related to these virtualization models constitute an important lever to the instantiation of these recommendations. The current efforts on Infrastructure-as-Code solutions may facilitate the efficient protection of resources more complex than unikernels. It is important to investigate new strategies for automating the generation of these resources with a proper control of the attack surface, and contributing to a better adaptation to contextual changes. Finally, the development of services at the edge and the integration of Internet of Things (IoT) resources to the Cloud introduces new important challenges in terms of heterogeneity and distribution. In that context, new management and programmability strategies should take into account the complementarity of endogeneous and exogeneous security mechanims to cope with scarce resources.

References

[1] Mell P, Grance T. The NIST Definition of Cloud Computing;.

[2] EU GDPR Information Portal;. Available from: `http://eugdpr.org/eugdpr.org-1.html`.

[3] Sahoo J, Mohapatra S, Lath R. Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. In: Proc. of the 2010 Second International Conference on Computer and Network Technology (ICCNT);. p. 222–226.

[4] Pearce M, Zeadally S, Hunt R. Virtualization: Issues, Security Threats, and Solutions. ACM Computing Surveys (CSUR);45(2):17:1–17:39.

[5] Pattaranantakul M, He R, Song Q, Zhang Z, Meddahi A. NFV Security Survey: From Use Case Driven Threat Analysis to State-of-the-Art Countermeasures. IEEE Communications Surveys & Tutorials;p. 1–1.

[6] Compastié M. Software-defined Security for Distributed Clouds. University of Lorraine; 2018. PhD Thesis.

[7] Popek GJ, Goldberg RP. Formal Requirements for Virtualizable Third Generation Architectures. Communications of the ACM (CACM);17(7):412–421.

[8] Robin JS, Irvine CE. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In: Proceedings of the 9th USENIX Security Symposium. DTIC Document;. p. 129–144.

[9] The Xen Project, the Powerful Open Source Industry Standard for Virtualization;. Available from: `https://xenproject.org/`.

[10] Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, et al. Xen and the Art of Virtualization. Proc of the 19th ACM symposium on Operating systems principles (ASOSP);37(5):164–177.

[11] Kivity A, Kamay Y, Laor D, Lublin U, Liguori A. KVM: the Linux Virtual Machine Monitor. In: Proc. of the 2007 Linux symposium. vol. 1;. p. 225–230.

[12] Oracle VM VirtualBox;. Available from: `https://www.virtualbox.org/`.

[13] QEMU;. Available from: `http://www.qemu-project.org/`.

[14] Kata Containers - The Speed of Containers, the Security of VMs;. Available from: `https://katacontainers.io/`.

[15] Kolyshkin K. Virtualization in Linux. White paper, OpenVZ;3.

[16] Felter W, Ferreira A, Rajamony R, Rubio J. An Updated Performance Comparison of Virtual Machines and Linux Containers. In: Proc. of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS);. p. 171–172.

[17] Soltesz S, Pötzl H, Fiuczynski ME, Bavier A, Peterson L. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. Proc of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys);41(3):275–287.

[18] Linux Containers - LXC - Introduction;. Available from: `https://linuxcontainers.org/fr/lxc/introduction/`.

[19] Docker - Build, Ship, and Run Any App, Anywhere;. Available from: `https://www.docker.com/`.

[20] gVisor: Container Runtime Sandbox; 2018. Available from: `https://github.com/google/gvisor`.

[21] Porter DE, Boyd-Wickizer S, Howell J, Olinsky R, Hunt GC. Rethinking the Library OS from the Top Down. Proc of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (SIGPLAN);39(1):291–304.

[22] Cheriton DR, Duda KJ. A Caching Model of Operating System Kernel Functionality. In: Proc. of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI). OSDI '94. USENIX Association;. .

[23] Engler DR, Kaashoek MF, O'Toole J Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. Proc of the 15th ACM symposium on Operating systems principles (SOSP);29(5):251–266.

[24] Larkby-Lahet J, Madden B, Wilkinson D, Mosse D. Xomb: an Exokernel for Modern 64-bit, Multicore Hardware. In: Proc. of 7th workshop of operating system (WSO);. p. 1991–1998.

[25] Madhavapeddy A, Mortier R, Rotsos C, Scott D, Singh B, Gazagnaire T, et al. Unikernels: Library Operating Systems for the Cloud. Proc of the 18th international conference on Architectural support for programming languages and operating systems (ASPLOS);48(4):461–472.

[26] Bratterud A, Walla A, Haugerud H, Engelstad PE, Begnum K. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In: Proc. of the IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom);. p. 250–257.

[27] Hastings R, Joyce B. Purify: Fast Detection of Memory Leaks and Access Errors. In: Proc. of the Winter 1992 USENIX Conference;. p. 125–138.

[28] Qin F, Lu S, Zhou Y. SafeMem: Exploiting ECC-memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In: Proc. of the 11th International Symposium on High-Performance Computer Architecture (HPCA);. p. 291–302.

[29] Bovet DP, Cesati M. Understanding the Linux Kernel: from I/O ports to Process Management. 3rd ed. O'Reilly Media, Inc.;.

[30] Solomon J, Huebner E, Bem D, Szeżynska M. User Data Persistence in Physical Memory. Digital Investigation;4(2):68 – 72.

[31] Kook J, Hong S, Lee W, Jae E, Kim J. Optimization of out of Memory Killer for Embedded Linux Environments. In: Proc. of the 2011 ACM Symposium on Applied Computing (SAC). ACM;. p. 633–634.

[32] CWE - 2011 CWE/SANS Top 25 Most Dangerous Software Errors;. Available from: `http://cwe.mitre.org/top25/`.

[33] MITRE. CVE-2017-3791;. Available from: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-3791`.

[34] Jaeger T, Sailer R, Zhang X. Analyzing Integrity Protection in the SELinux Example Policy. In: Proc. of the 12th Conference on USENIX Security Symposium (SSYM). USENIX Association;. p. 5–5.

[35] MITRE. CVE-2016-9919;. Available from: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9919`.

[36] MITRE. CVE-2013-0273;. Available from: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0273`.

[37] Scut TT. Exploiting Format String Vulnerabilities;.

[38] MITRE. CVE-2016-3172;. Available from: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9962`.

[39] Johns Martin. Code-injection Vulnerabilities in Web Applications — Exemplified at Cross-site Scripting. it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik;53(5):256–160.

[40] Netzer RHB, Miller BP. What Are Race Conditions?: Some Issues and Formalizations. ACM Letters on Programming Languages and Systems (LOPLAS);1(1):74–88.

[41] Apt - Debian Wiki;. Available from: `https://wiki.debian.org/Apt`.

[42] 0install: Overview;. Available from: `http://0install.net/`.

[43] Di Cosmo R, Zacchiroli S, Trezentos P. Package Upgrades in FOSS Distributions: Details and Challenges. In: Proc. of the 1st International Workshop on Hot Topics in Software Upgrades (HotSWUp). HotSWUp '08. ACM;. p. 7:1–7:5.

[44] Crameri O, Knezevic N, Kostic D, Bianchini R, Zwaenepoel W. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. In: Proc. of the Twenty-first ACM Symposium on Operating Systems Principles (SIGOPS). SOSP '07. ACM;. p. 221–236. Event-place: Stevenson, Washington, USA.

[45] Heyens J, Greshake K, Petryka E. MongoDB Databases at Risk. Center for IT-Security, Privacy, and Accountability;.

[46] Liedtke J. On Micro-kernel Construction. In: Proc. of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP). SOSP '95. ACM;. p. 237–250.

[47] Madhavapeddy A, Scott DJ. Unikernels: Rise of the Virtual Library Operating System. Queue;11(11):30:30–30:44.

[48] Ristenpart T, Tromer E, Shacham H, Savage S. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In: Proc. of the 16th ACM Conference on Computer and Communications Security (CCS). CCS '09. ACM;. p. 199–212.

[49] Bazm M, Lacoste M, Südholt M, Menaud J. Side-channels Beyond the Cloud Edge: New Isolation Threats and Solutions. In: Proc. of the 1st Cyber Security in Networking Conference (CSNet);. p. 1–8.

[50] Open vSwitch;. Available from: `http://www.openvswitch.org/`.

[51] Tak B, Isci C, Duri S, Bila N, Nadgowda S, Doran J. Understanding Security Implications of Using Containers in the Cloud. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association;. p. 313–319. Available from: `https://www.usenix.org/conference/atc17/technical-sessions/presentation/tak`.

[52] MITRE. CVE-2015-3456;. Available from: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456`.

[53] Matousek P. VENOM, Don't Get Bitten;. Available from: `https://access.redhat.com/blogs/product-security/posts/1976633`.

[54] Kortchinsky K. Cloudburst. Black Hat USA;Available from: `http://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-PAPER.pdf`.

[55] MITRE. CVE-2007-1744;. Available from: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1744`.

[56] MITRE. CVE-2012-0217;. Available from: `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0217`.

[57] Quist D, Smith V, Computing O. Detecting the Presence of Virtual Machines Using the Local Data Table. Offensive Computing;.

[58] Nance K, Bishop M, Hay B. Virtual Machine Introspection: Observation or Interference? IEEE Security & Privacy;6(5):32–37.

[59] Li C, Raghunathan A, Jha NK. Secure Virtual Machine Execution under an Untrusted Management OS. In: Proc. of the 3rd IEEE International Conference on Cloud Computing (CLOUD);. p. 172–179.

[60] Garfinkel T, Rosenblum M, others. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proc. of the 2003 Network & Distributed System Security Symposium (NDSS). vol. 3;. p. 191–206.

[61] Kocher P, Jaffe J, Jun B. Differential Power Analysis. In: Wiener M, editor. Proc. of the 19th Annual International Cryptology Conference (CRYPTO). Springer Berlin Heidelberg;. p. 388–397.

[62] Wojtczuk R, Rutkowska J. Attacking Intel Trusted Execution Technology. Black Hat DC;2009.

[63] Torr P. Demystifying the Threat Modeling Process. IEEE Security & Privacy;3(5):66–70.

[64] MITRE. CVE-2014-0230;. Available from: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0230`.

[65] Gu W, Kalbarczyk Z, Iyer RK, Yang Z. Characterization of Linux Kernel Behavior under Errors. In: Proc. of the International Conference on Dependable Systems and Networks (DSN). vol. 00;. p. 459–468.

[66] Lin CH, Chen CH, Laih CS. A Study and Implementation of Vulnerability Assessment and Misconfiguration Detection. In: Proc. of the 2008 IEEE Asia-Pacific Services Computing Conference (APSCC);. p. 1252–1257.

[67] Barrère M, Badonnel R, Festor O. A SAT-based Autonomous Strategy for Security Vulnerability Management. In: Proc. of the 2014 IEEE Network Operations and Management Symposium (NOMS);. p. 1–9.
[68] Luettmann BM, Bender AC. Man-in-the-middle attacks on auto-updating software. Bell Labs Technical Journal;12(3):131–138.
[69] Cappos J, Samuel J, Baker S, Hartman JH. A Look in the Mirror: Attacks on Package Managers. In: Proc. of the 15th ACM Conference on Computer and Communications Security (CCS). CCS '08. ACM;. p. 565–574.
[70] Pincus J, Baker B. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. IEEE Security & Privacy;2(4):20–27.
[71] Abate P, Cosmo RD, Treinen R, Zacchiroli S. A Modular Package Manager Architecture. Information and Software Technology;55(2):459 – 474.
[72] Checkoway S, Shacham H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In: Proc. of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM;. p. 253–264.
[73] Op F. The FU Rootkit;.
[74] Carbone M, Zamboni D, Lee W. Taming Virtualization. IEEE Security & Privacy;6(1):65–67.
[75] King ST, Chen PM. SubVirt: implementing malware with virtual machines. In: Proc. of the 2006 IEEE Symposium on Security and Privacy (S&P);. p. 14 pp.–327.
[76] Horn J. Pandavirtualization: Exploiting the Xen hypervisor;. Available from: https://googleprojectzero.blogspot.fr/2017/04/pandavirtualization-exploiting-xen.html.
[77] Wojtczuk R. Subverting the Xen hypervisor. In: Black Hat USA. vol. 2008;. Available from: https://www.blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf.
[78] Ormandy T. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments;.
[79] Waldspurger CA. Memory Resource Management in VMware ESX Server. Proc of the 5th Symposium on Operating Systems Design and Implementation (OSDI);36:181–194.
[80] Carpenter M, Liston T, Skoudis E. Hiding Virtualization from Attackers and Malware. IEEE Security Privacy;5(3):62–65.
[81] Wojtczuk R, Rutkowska J. Following the White Rabbit: Software attacks against Intel VT-d technology;. Available from: https://invisiblethingslab.com/resources/2011/SoftwareAttacksonIntelVT-d.pdf.
[82] Zhou F, Goel M, Desnoyers P, Sundaram R. Scheduler Vulnerabilities and Coordinated Attacks in Cloud Computing. Journal of Computer Security;21(4):533–559.
[83] Jasti A, Shah P, Nagaraj R, Pendse R. Security in multitenancy cloud. In: 44th Annual 2010 IEEE International Carnahan Conference on Security Technology;. p. 35–41.
[84] Barresi A, Razavi K, Payer M, Gross TR. CAIN: Silently Breaking ASLR in the Cloud. In: Proc. of the 2015 Workshop on Offensive Technologies (WOOT);. .
[85] Xiao Y, Zhang X, Zhang Y, Teodorescu M. One Bit Flips, One Cloud Flops: Cross-vm Row Hammer Attacks and Privilege Escalation. In: Proc. of the 25th USENIX Security Symposium (Security);. p. 19–35.
[86] Zhang Y, Juels A, Reiter MK, Ristenpart T. Cross-VM Side Channels and Their Use to Extract Private Keys. In: Proc. of the 2012 ACM Conference on Computer and Communications Security (CCS). CCS '12. ACM;. p. 305–316.
[87] Geffner J. VENOM Vulnerability;. Available from: http://venom.crowdstrike.com/.
[88] Morbitzer M, Huber M, Horsch J, Wessel S. SEVered: Subverting AMD's Virtual Machine Encryption. In: Proc. of the 11th European Workshop on Systems Security (EuroSec). ACM;. p. 1:1–1:6.
[89] Shacham H, Page M, Pfaff B, Goh EJ, Modadugu N, Boneh D. On the Effectiveness of Address-space Randomization. In: Proc. of the 11th ACM Conference on Computer and Communications Security (CCS). CCS '04. ACM;. p. 298–307.
[90] Dautenhahn N, Kasampalis T, Dietz W, Criswell J, Adve V. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. Proc of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS);43(1):191–206.
[91] Lie D, Thekkath CA, Horowitz M. Implementing an Untrusted Operating System on Trusted Hardware. In: Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP). SOSP '03. ACM;. p. 178–192.
[92] Baumann A, Peinado M, Hunt G. Shielding Applications from an Untrusted Cloud with Haven. ACM Transactions on Computer Systems (TOCS);33(3):8:1–8:26.
[93] Szefer J, Lee RB. A Case for Hardware Protection of Guest VMs from Compromised Hypervisors in Cloud Computing. In: Proc. of the 31st International Conference on Distributed Computing Systems Workshops (ICDCSW);. p. 248–252.
[94] Arnautov S, Trach B, Gregor F, Knauth T, Martin A, Priebe C, et al. SCONE: Secure Linux Containers with Intel SGX. In: Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). vol. 16;. p. 689–703.
[95] Colp P, Nanavati M, Zhu J, Aiello W, Coker G, Deegan T, et al. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In: Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP). SOSP '11. ACM;. p. 189–202.
[96] Steinberg U, Kauer B. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In: Proc. of the 5th European Conference on Computer Systems (EuroSys). ACM;. p. 209–222.
[97] Sailer R, Jaeger T, Valdez E, Caceres R, Perez R, Berger S, et al. Building a MAC-based security architecture for the Xen open-source hypervisor. In: Proc. of the 21st Annual Computer Security Applications Conference (ACSAC);. p. 10 pp.–285.
[98] Tupakula U, Varadharajan V. TVDSEC: Trusted Virtual Domain Security. In: Proc. of the Fourth IEEE International Conference on Utility and Cloud Computing (UCC);. p. 57–64.
[99] Chung CJ, Khatkar P, Xing T, Lee J, Huang D. NICE: Network Intrusion Detection and Countermeasure Selection in Virtual Network Systems. IEEE Transactions on Dependable and Secure Computing;10(4):198–211.
[100] Artac M, Borovssak T, Nitto ED, Guerriero M, Tamburri DA. DevOps: Introducing Infrastructure-as-Code. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C); 2017. p. 497–498.
[101] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, et al. seL4: Formal Verification of an OS Kernel. In: Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP). ACM;. p. 207–220.
[102] Nelson L, Sigurbjarnarson H, Zhang K, Johnson D, Bornholt J, Torlak E, et al. Hyperkernel: Push-Button Verification of an OS Kernel. In: Proc. of the 26th Symposium on Operating Systems Principles (SOSP). SOSP '17. ACM;. p. 252–269.
[103] Criswell J, Dautenhahn N, Adve V. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In: Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM;. p. 81–96.
[104] Freitas L, McDermott J. Formal methods for security in the Xenon hypervisor. International Journal on Software Tools for Technology Transfer. 2011 May;13(5):463.
[105] Madhavapeddy A, Leonard T, Skjegstad M, Gazagnaire T, Sheets D, Scott DJ, et al. Jitsu: Just-In-Time Summoning of Unikernels. In: Proc. of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI).;. p. 559–573.
[106] Williams D, Koller R. Unikernel Monitors: Extending Minimalism Outside of the Box. In: Proc. of the 8th USENIX

Workshop on Hot Topics in Cloud Computing (HotCloud). USENIX Association;. .

[107] Hong CH, Spence I, Nikolopoulos DS. GPU Virtualization and Scheduling Methods: A Comprehensive Survey. ACM Computing Surveys. 2017 Jun;50(3):1–37. Available from: http://dl.acm.org/citation.cfm?doid=3101309.3068281.

[108] Jiang X, Wang X, Xu D. Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction. In: Proc. of the 14th ACM Conference on Computer and Communications Security (CCS). CCS '07. ACM;. p. 128–138.

[109] Christodorescu M, Sailer R, Schales DL, Sgandurra D, Zamboni D. Cloud Security Is Not (Just) Virtualization Security: a Short Paper. In: Proc. of the 2009 ACM workshop on Cloud computing security (CCSW). ACM Press;. p. 97–102.

[110] Payne BD, Carbone M, Sharif M, Lee W. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In: Proc. of the 2008 IEEE Symposium on Security and Privacy (S&P);. p. 233–247.

[111] Riley R, Jiang X, Xu D. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In: Lippmann R, Kirda E, Trachtenberg A, editors. Proc. of the 11th International Symposium of Recent Advances in Intrusion Detection (RAID). Springer Berlin Heidelberg;. p. 1–20.

[112] Bacs A, Giuffrida C, Grill B, Bos H. Slick: An Intrusion Detection System for Virtualized Storage Devices. In: Proc. of the 31st Annual ACM Symposium on Applied Computing (SAC). ACM;. p. 2033–2040.

[113] Pattaranantakul M, Tseng Y, He R, Zhang Z, Meddahi A. A First Step Towards Security Extension for NFV Orchestrator. In: Proc. of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFVSec). SDN-NFVSec '17. ACM;. p. 25–30.

[114] Hachana S, Cuppens-Boulahia N, Cuppens F. Mining a high level access control policy in a network with multiple firewalls. Journal of Information Security and Applications;20:61–73.

[115] Walla AA. Live Updating in Unikernels [Master's Thesis];. Available from: https://www.duo.uio.no/bitstream/handle/10852/59240/live-updating-unikernels.pdf?sequence=45.

[116] Goel A, Po K, Farhadi K, Li Z, de Lara E. The Taser Intrusion Recovery System. Proc of the 20th ACM symposium on Operating systems principles (SOSP);39(5):163–176.