Unikernel-based Approach for Software-Defined Security in Cloud Infrastructures

Maxime Compastié, Rémi Badonnel, Olivier Festor, Ruan He, Mohamed Kassi-Lahlou

Unikernel-based Approach for Software-Defined Security in Cloud Infrastructures

Maxime Compastié*†, Rémi Badonnel*, Olivier Festor*, Ruan He† and Mohamed Kassi-Lahlou†
*Madynes Research Team - LORIA/INRIA, France. {remi.badonnel,olivier.festor}@loria.fr
†Orange Labs, France.{maxime.compastie,ruan.he,mohamed.kassilahlou}@orange.com

***Abstract*—The heterogeneity of cloud resources implies substantial overhead to deploy and configure adequate security mechanisms. In that context, we propose a software-defined security strategy based on unikernels to support the protection of cloud infrastructures. This approach permits to address management issues by uncoupling security policy from their enforcement through programmable security interfaces. It also takes benefits from unikernel virtualization properties to support this enforcement and provide resources with low attack surface. These resources correspond to highly constrained configurations with the strict minimum for a given period. We describe the management framework supporting this software-defined security strategy, formalizing the generation of unikernel images that are dynamically built to comply with security requirements over time. Through an implementation based on MirageOS, and extensive experiments, we show that the cost induced by our security integration mechanisms is small while the gains in limiting the security exposure are high.**

***Index Terms*—Security Management, Cloud Infrastructures and Services, Software-Defined Security, Resource Virtualization, Unikernel.**

I. Introduction

Cloud computing provides an architectural model to build elaborated services and applications based on multiple computing resources, such as virtual machines, network devices, software components, themselves given as a service that can be easily deployed through the Internet. Its elasticity and on-demand properties permit to reduce operational costs related to these applications [1]. Pursuing these objectives, the distribution of clouds across different infrastructures (*multi-cloud*) and across stakeholders (*multi-tenancy*) increases management complexity. From a security perspective, it affects both the security policy specification and its enforcement to provide adequate security mechanisms to resources.

Addressing these issues, software-defined security (SDSec) permits to decouple security policy enforcement from their specification by abstracting resources technical considerations. The control plane is in charge of security decisions through business-logic constraints, while data plane represents the resources to be protected and dedicated programmable security mechanisms accounting for technical constraints. The provided security should scrutinize the resources and their context to dynamically adjust the enforcement and keep the compliance to security requirements. While the adequacy of these mechanisms with allocated resources directly impacts the coverage and efficiency of security management, their heterogeneity requires specialized mechanisms, increasing the management costs and reducing the protection exhaustiveness.

We present a software-defined security strategy based on unikernels for protecting cloud infrastructures. These unikernels correspond to lightweight VMs specially built for a dedicated application at the expense of backward compatibility, while their performance enables a short lifespan-based usage [2]. They promote a simplified and analyzable internal functioning, putting on an equal footing an application, its requirements and the OS routines they rely on. We exploit their properties to reduce the attack exposure of cloud resources, through the on-the-fly generation of highly constrained configurations with the strict minimum for a given time-limited period. Our main contributions are: (i) specifying a software-defined strategy based on unikernels to protect cloud infrastructures, (ii) designing a management framework with a dedicated modeling to support the generation of unikernels with security mechanisms, (iii) implementing a proof-of-concept prototype, and (iv) providing a performance evaluation.

The remainder of the paper is structured as follows: we present related work in Section II and provide a background on unikernels in Section III. Section IV describes our software-defined security strategy with the formalization of unikernel image generation over time. We evaluate in Section V the performances in comparison to legacy virtualization, through an implementation prototype. We conclude and point out future research efforts in Section VI.

II. Related Work

Securing cloud infrastructures is a challenge largely explored in the literature. For instance, [3] highlights several challenges related to policy-based security management, such as the specification of a cloud security policy, the assurance of the security decisions, as well as the the certification of security components in that context. In the same manner, the *TCloud* framework [4] proposes to enforce a security policy with a hardened cloud stack. This one provides multi-level security components, that might be compatible with multi-cloud environments, . However, these solutions do not specifically address the self-configuration nor the management issues generated by multi-cloud and multi-tenancy properties. The *Iceman* architecture [5] enables secure federated inter-cloud identity management. The author of [6] proposes a cloud management framework caping with multi-tenancy, but this one is limited to access control policies and cannot support

other security mechanisms. We have already argued in favor of a SDSec framework for distributed computing in [7] where we described the main building blocks and analyzed its benefits.

In the area of programmability, *software-defined networking (SDN)* permits to separate the *control plane* making traffic routing decisions from the *data plane* forwarding packets. This paradigm enables a dynamic and adaptive policy enforcement. It may also serve as a support for chaining functions, as considered by the *Flowtags* framework [8], and be extended to security policy enforcement, as supported in [9] through middleboxes. We have also shown in [10] how to exploit the SDN paradigm to build a chain of security functions to protect smart devices. IETF is also working on security services using interface to network security functions [11]. Such approaches employs SDN with respect to security policy enforcement. Our objective here is to show how unikernel properties can contribute to support security programmability.

System virtualization enables in-VM applications and communications inspection by security mechanisms. Lares [12] is a VM active monitoring architecture based on in-situ mechanism injection. On the contrary, VMWatcher [13] addresses a semantic view reconstruction from VM memory for non-intrusive monitoring purpose. Similarly, Livewire [14] is an programmable IDS inspecting VM state. Slick [15] focuses on VM storage I/O to expose intrusion attempts. Specific architectures also contribute to reducing the attack surface. Microkernel [16] sustains OS compartmentalization approach by dispatching non critical OS features across isolated services, with a serious overhead. Exokernel [17] is a library OS promoting routines injection to prevent inconsistent behavior in hardware resource management due to process switch. Considering system virtualization, Drawbridge [18] is another library OS encapsuling application in a VM but collaborating with a host OS for hardware access and seamless user interface provisioning. ukvm [19] is a monitor embedding only required features for one VM execution. [20] sustains the unikernel usage in cloud environment from a security perspective. Our purpose is to take benefits from unikernels to enable SDSec.

III. Background on unikernels

Virtualization is one important pillar of cloud computing to provide computer resources and enables hardware to be abstracted from programs by a software layer. Hypervisors are in charge of provisioning dedicated virtual hardware environments to each program and interfacing with hardware. They can be distributed according to the manner they operate hardware resources: Type-I (or native) hypervisors operate directly hardware resources while Type-II (or embedded) hypervisors have to rely on host system routines to operates them. They can help enforcing resource isolation and enable configuration flexibility inside virtual machines (VMs). The complexity of system architectures in VMs, relying on full-featured OS kernel and application runtime, gives a glimpse of optimizations through the simplification of these dependencies. OS-level virtualization has relinquished guest OS kernel from virtual environment to solely rely on the host one for hardware resource management and isolation. This comes with the cost of leaving the system virtualization area.

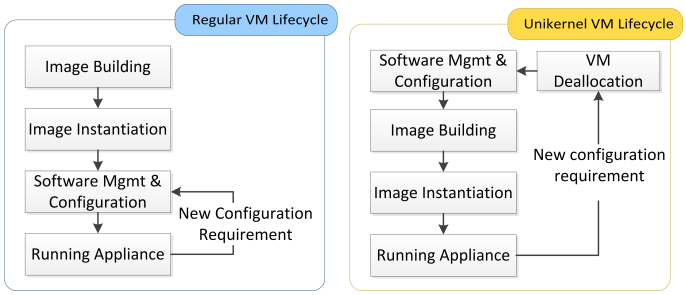


Figure 1: Comparison of regular and unikernel VM lifecycles

Unikernel VMs, whose lifecycle is depicted on Figure 1, also address this issue, but with a system virtualization-based solution. It simplifies the in-VM system architecture by restricting each VM to one application, and embracing the *library OS* paradigm: those OS implement the hardware resource management to a set of libraries that are linked to the applications. This enables them to closely control the hardware resource they rely on, in the most adapted way to their mission but at the cost of restricting hardware support to the linked library. A unikernel system image embeds a single application with its dependencies. Those encompass a minimal runtime together with software and hardware resources management libraries, in accordance with the library OS concept. These components can be either specifically related to a development platform (i.e. Mirage [2] with OCaml, Clive [21] with Go and IncludeOS [22] with C++) or unspecific to any of them (e.g. Rumprun [23] and OSv [24]). Nevertheless, the least case can be assimilated to the first one if a specific runtime environment is packed in the images. The software management of unikernel is performed externally through provided building tool-stack, directly on system image before instantiation. This approach alleviates unikernel from the in-situ software management constraint. Management can still rely on package manager (e.g. OPAM [25] for Mirage) or solely on source code inclusion mechanisms (e.g. IncludeOS). An instance proposes a single address space for its application and its runtime. It does not implement process to outwit the context switch overhead. Unikernel images can be designed to cope with an hypervisor and be launched in a VM. For performance motivation, all the routines in unikernel VMs run in privileged mode to ensure an access to the hardware environment without the trapping overhead. From our security perspective, unikernels offer interesting properties to reduce the attack surface by generating highly constrained configurations limited to the strict minimum in a dynamic manner.

IV. A software-defined security framework based on unikernels for cloud infrastructures

We propose a SDSec framework that exploits unikernel properties for protecting cloud infrastructures. This framework permits to generate on-the-fly unikernel images that integrate

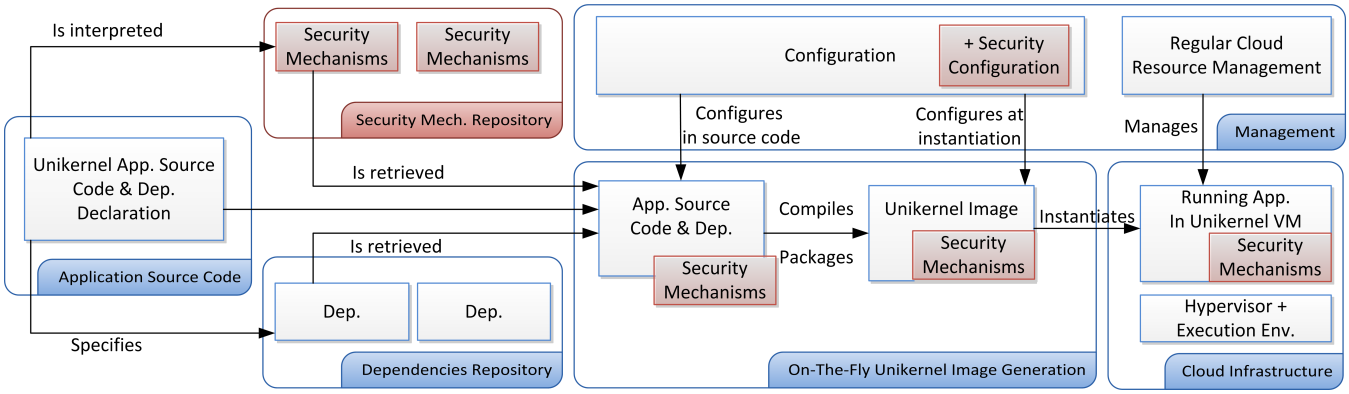


Figure 2: SDSec framework for cloud infrastructures with on-the-fly generation of unikernel images

protection mechanisms. Unikernels have a less flexible configuration in comparison to regular virtual machines, but allow significant reduction of the attack surface. In that context, we exploit unikernels to build highly-constrained configurations limited to the strict minimum and with a time-limited validity. Configuration changes are supported by the generation of new unikernel images in a dynamic manner.

As shown on Figure 2, the framework is layered on top of the regular building chain of unikernels. A unikernel application source code defines the main behavior of the considered appliance and its library dependencies. Those are stored in a dedicated repository. Security mechanisms, stored on another repository, have their requirements interpreted from source code dependencies declaration. During the image generation, the dependencies and required security mechanisms are fetched, and unikernel source code is compiled and assembled in a virtual machine image. This virtual machine can be instantiated in a cloud infrastructure, on the top of a system virtualization environment. The building and start of the virtual machines are performed by a dedicated plane. While the regular cloud resource management addresses the running application in the unikernel virtual machine, a specific configuration management concerns security requirements. We added the capability to handle the configuration of security mechanisms to meet security requirements.

This strategy based on unikernels enables a more comprehensive protection perimeter. These mechanisms are expected to address the protection of all the accessible resources of the unikernel VM, beneficing of the library OS paradigm to include the runtime components hardware management in its scope. This extends the potential protection perimeter of security component over resources in an instantiated unikernel VM. This also enables a highly-coupled enforcement, by deploying security mechanisms at the building step of unikernel images. This approach dwarfs the enforcement overhead by enabling explicit collaboration between protected resources and mechanisms. Moreover, addressing the modification of every component of the unikernel extends the variety of potential enforcement features. We consider here the proactive programmability of security mechanisms: we restrict the alteration of their configuration before their instantiation.

A. *On-the-fly unikernel generation*

We formalize unikernels and their on-the-fly generation supporting our SDSec approach for cloud infrastructures. We consider in that context that software component operations are enabled by software manager capabilities currently used in the unikernel ecosystem. We also take into account the programmability constraints related to SDSec by addressing configuration management through a non-binding model, in respect with the lack of consensus on the configuration specification model in the unikernel ecosystem. We introduce a unitary software component integrable in a unikernel image as a module noted *m*. This terminology can refer to both software packages or injectable source code files, which is consistent with considered software inclusion mechanisms for unikernels. The whole set of usable software modules is identified as $\mathscr{M}$, with $m \in \mathscr{M}$. In practice, this matches all the packages in the configured repositories or all the libraries in the include directories. We also refer to c as a configuration option, and $\mathscr{C}(m)$ as the whole set of options applicable to a given module m. We expect c to be a condition that can be met by the code of the module m. Each constructible unikernel image u are encompassed in a set $\mathscr{U}$. We point out u_0 to be the nil element of this set, alluding to be the minimal image on which any other is rooted. This elementary modeling enables the definitions of basic operations on unikernels. The *Install* operation is the insertion of a module m in a image u, resulting in a new kernel image u', as given by Equation 1.

$$\begin{aligned} Install : &\mathscr{U} \times \mathscr{M} \longrightarrow \mathscr{U}, \\ &(u, m) \longmapsto Install(u, m) = u' \end{aligned} \tag{1}$$

In the meantime, the *Configure* operation, given by Equation 2, permits to modify or alter the code embedded within a module m installed in a unikernel image u to meet a configuration option c, resulting in a new image u'

$$\begin{aligned} Configure : &\mathscr{U} \times \mathscr{M} \times \mathscr{C}(\mathscr{M}) \longrightarrow \mathscr{U}, \\ &(u, m, c) \longmapsto Configure(u, m, c) = u' \end{aligned} \tag{2}$$

Complementarily, we introduce observation operations to describe unikernel images. In particular, we define the *Modules*

operation to get the set of inserted modules, and the *Configuration* operation to get the set of activated configuration options for a given module on a unikernel image. These operations are respectively given by Equations 3 and 4.

$$\begin{aligned} Modules : &\mathscr{U} \longrightarrow \mathscr{P}(\mathscr{M}), \\ &u \longmapsto Modules(u) \end{aligned} \tag{3}$$

$$\begin{aligned} Configuration : \mathscr{U} \times Modules(\mathscr{U}) \longrightarrow \mathscr{P}(\mathscr{C}(\mathscr{P}(\mathscr{M}))), \\ (u, m) \longmapsto Configuration(u, m) \end{aligned} \tag{4}$$

Several constraints are inferred from the unikernel properties. First, as formalized in Equation 5, the installation operation does not remove any of the modules previously installed. This is a necessary condition to comply with the insertion mechanism, which only increases addressable routines in in-compilation objects and package managers that do not support conflict management. Unikernel code patching is therefore assimilable to a module insertion method. We also assume that the re-insertion of a module does not commit any modifications to the image, to layer the behavior of most package manager. This assertion is acceptable in the case of code inclusion, as most libraries protect themselves against multiple inclusions based on preprocessor directive usages.

$$\begin{aligned} Modules(u) \subseteq Modules(Install(u, m)) \\ \text{(equality if } m \in Modules(u)) \end{aligned} \tag{5}$$

In addition, we conversely assume that fulfilling a particular configuration option can prevent meeting the requirement of another one. A typical example is expecting a unikernel application variable to respect a value, while reconfiguring it with another one. This second constraint is given by Equation 6.

$$\begin{aligned} Configuration(u, m) \nsubseteq \\ Configuration(Configure(u, m, c), m) \end{aligned} \tag{6}$$

We also assume that the minimal unikernel image does not embed any module, as provided by Equation 7. This condition is required to permit the building of any unikernel image with solely the additive installation operation.

$$Modules(u_0) = \oslash \tag{7}$$

Our modeling introduces also two types of relationships to express the compatibility amongst software components in a unikernel image, inherited from package managers [26]. For a given module *m*, the dependency relationship, addressed by the $requires(m)$ operation, identifies the set of modules expected in a unikernel image before its installation, while the $conflicts(m)$ operation points the set of modules that prevents its installation. As a consequence, we consider that a given module is installable in a unikernel image when all its dependencies are already installed and no conflicting ones are present, as given by Equation 8.

$$\begin{aligned} IsInstallable(u, m) \Leftrightarrow (Modules(u) \supseteq requires(m)) \wedge \\ (Modules(u) \cap conflicts(m) = \oslash) \end{aligned} \tag{8}$$

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

Algorithm 1 Unikernel image generation for cloud SDSec

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

The security mechanisms are therefore directly integrated to unikernel VM, as part of the application dependencies. This minimizes the enforcement disturbances due to external factors, contributes to dwarf the attack surface, and prevents the semantic gap [27] issue with respect to security enforcement.

The compilation time processing permits an intrinsic integration of protective and protected mechanisms as they both undergo the same code optimization and linking processes, leading to minimize security mechanism overhead. This integration also goes in favor of security-by-design properties, on the condition that the remainder of the application code supports it. Currently, the configuration of security mechanisms themselves is performed before their instantiation in unikernel

VMs in a pro-active manner: it requires image re-building to cope with a new given context, but contributes to reduce the complexity of a further security orchestration [28], [29].

B. Benefits of unikernels for SDSec

Independently from the integration of security mechanisms at compilation time, unikernels provide multiple benefits for SDSec. We detail them below.

a) Reduced attack surface: Unikernels reduce the required code base for applications, contributing to reduce the attack surface by circumventing software flaws and exploitable entry points, but and decrease the capability of an attacker having succeed in jeopardizing one instance. The clearance of any memory isolation in unikernel VMs makes the system virtualization the only remaining memory barrier. This approach dismisses the in-VM multi-tenancy support, leaving them as the atomic resources to address tenancy and infrastructure location which is still acceptable as long as unikernel embeds a single application instance for one tenant.

b) Image building sanitization: The construction tooling loads the unikernel source code to process it into a bootable VM unikernel image through source code static analysis, compilation and objects linking. The static analysis step attests the coherence of the code and its compliance with the programming language properties. The nature of the latter directly impacts the processing of this verification. By scoping the codes of an application, its dependencies and its runtime, unikernels transpose these verifications to a broader, more inclusive field than regular system stack.

c) Code portability: As each unikernel image embeds an application and all its dependencies, its outwits traditional compatibility issues between in-VM components. This restricts compatibility issues to the ones amongst the unikernel VMs and the hypervisor and their execution environment (i.e. computing resources, storage and networking prerequisites). Configured accordingly, several hypervisors can be supported. By considering hypervisors across several infrastructures, unikernels can embrace the multi-cloud constraint.

V. Performance evaluation

We designed and implemented a proof-of-concept prototype. It serves as a basis to quantify the performances of our approach through extensive series of experiments. The considered security mechanisms concern both the authentication and the access control for HTTP servers.

A. Prototype implementation

We have implemented a proof-of-concept prototype based on an HTTP server over the MirageOS unikernel. As unikernels do not propose of-the-shelve applications but ought the developers to redesign them, we could not refer to a standard solution (e.g. Apache webserver). Instead, we have considered the demonstrative HTTP server provided by the Mirage project itself [30], that has been complemented to insert the required security mechanisms. MirageOS is a unikernel solution based on the OCaml programming language, featuring the compilation of VM images for Xen or KVM hypervisors or Unix executables. OCaml provides object-oriented programming and integrates a strong static and inferred typing model.

The web server implementation is based on the CoHTTP library [31]. It listens for connections on ports 443. The security mechanism configuration is specified through its source code, in two main configuration points: (i) a boolean specifying the behavior for anonymous or unrecognized credentials, (ii) a data structure for each enforced resources, enumerating themselves the accepted credential list. This mechanism is inserted into the unikernel image. Its main procedure is input by the HTTP request, and return a boolean reflecting the access granting defined in the configuration. To integrate it with the server, we had to insert one hook in the client request processing and add the support for the 403 HTTP error code.

B. Qualitative and quantitative evaluations

From a qualitative viewpoint, this implementation has proven the feasibility to integrate security mechanisms in unikernel VMs, with a limited base modification from protected resources. Their performances together with hardware resource management and self-scalability capability through fast boot time identify them as considerable newcomers for cloud infrastructures, but also for virtual network functions design. We address their configurability before resource instantiation, sealing their configuration options values during their lifespan. In practice, this security module implements authentication and access control to resources: supplying the correct credentials results in a 200-typed HTTP server response while the wrong credentials raises a 403-typed HTTP answer.

We have performed the quantitative evaluation on the following testbed. As host virtualization system and image building environment, we have used a Intel i7-5500U CPU at 4x2.40GHz equipped with 16GB of RAM, running Ubuntu 16.04 with Linux 4.4.0-93, Virtualbox 5.0.40 for regular VM experimentation, Docker 1.12.6 as the container engine and ukvm 0.2.2 as unikernel monitor [19]. Each guest system is allocated with 1 vCPU, 512MB of RAM, and the latest versions of their software components (i.e. up-to-date Ubuntu 16.04 for Virtualbox VM and Docker image, and Mirage 3.0.4 for unikernel). Virtualbox VM and Docker container provides Apache2. Unikernel implements CoHTTP-based webserver. HTTP workload tests have been performed with WRK [32] and execution times are measured with `time` utility.

In a first series of experiments, we quantified the HTTP server performances with securized unikernels, and compare them to unikernels and other virtualization solutions. Figure 3 illustrates workload concerns by measuring successful responses modulated by the number of concurrent connections: we state that, although containers provide a more important workload support than unikernels, the latters have a substantially lighter memory footprint, as illustrated by Figure 5. The insertion of our security mechanisms slightly lowers the workload support (an average reduction of 6.5%) with almost no impact on VM memory consumption (an overhead of 2 MBio with the 200-connections workload scenario, no overhead when idle). In second and third series of experiments,

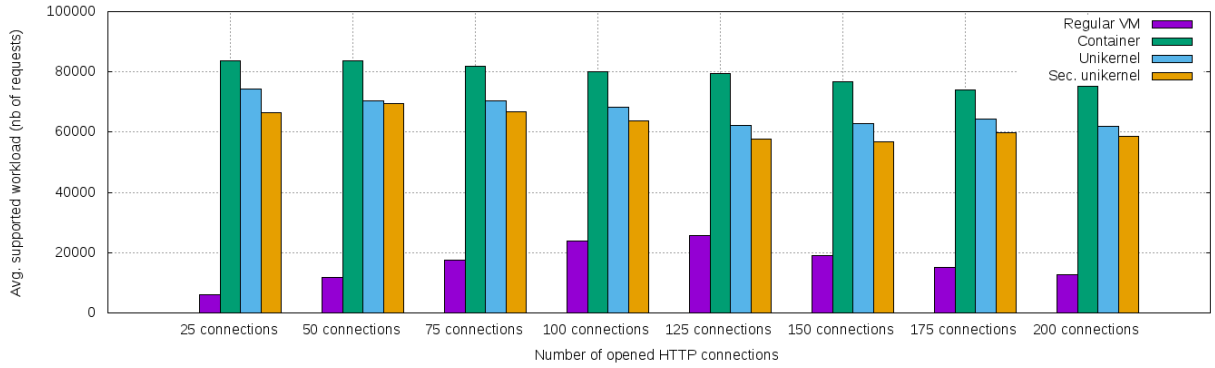
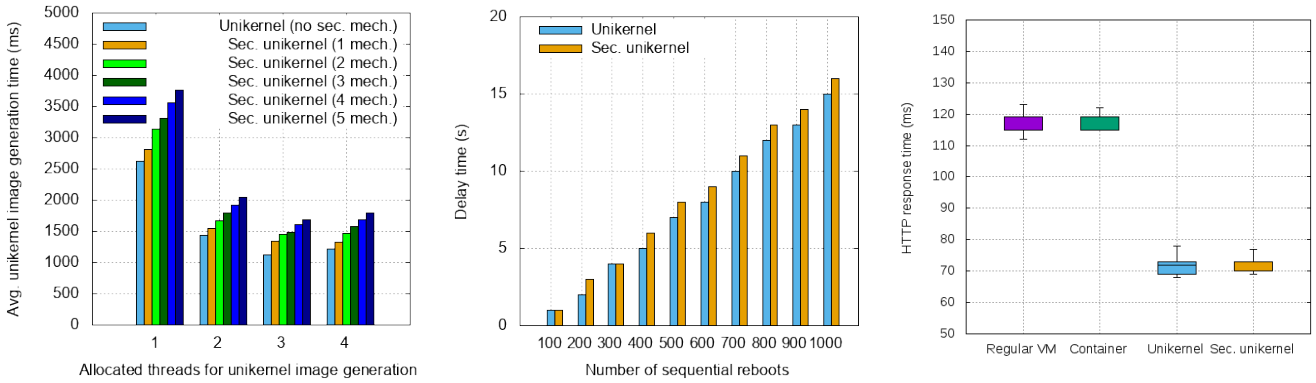


Figure 3: Average supported workload of securized unikernels compared to unikernels and other virtualization solutions



(a) Image generation (b) Unikernel VM reboot (c) HTTP request for an authorized access

Figure 4: Delay time measurement for on-the-fly unikernel VM re-instantiation

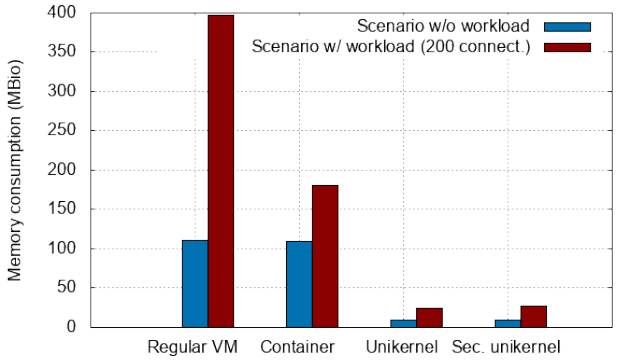


Figure 5: Compared memory consumption measurement

we evaluated the incidence of adding security mechanisms on the delay times required for generating unikernel images (Figure 4a) and rebooting unikernel VMs (Figure 4b). The generation process is analyzed by parameterizing the number of security mechanisms to be inserted and the allocated threads. We observe that security mechanism insertion induces an average overhead of 7.87% for image generation and preserves the linearity of unikernel reboot delays toward the number of performed ones. In a last series of experiments related to Figure 4c, we compared the performances with respect to HTTP requests delay time over a unikernel with and without the built security mechanisms. The observed overhead induced by security mechanisms over the unikernel solution is limited to 0.2 ms on average during experiments. Securized unikernels sustain a 38% delay decreases over regular VMs.

VI. Conclusions

We have proposed a unikernel-based strategy for supporting SDSec in cloud infrastructures. We have specified the underlying framework and formalized the on-the-fly generation of unikernel images that support this solution, with highly-constrained configurations limited to the strict minimum with a time-limited validity. A proof of concept prototype has been developed and the performance of such a strategy have been evaluated through series of experiments, and compared to other virtualization solutions. Our results show that the costs induced by security mechanisms integration are relatively limited, and unikernels are well suited to minimize risk exposure.

As future work, we will extend this unikernel-based approach to partially dynamic configurations of security mechanisms accounting multi-tenancy and multi-cloud configurations. Later, we plan to pursue investigating the integration of such a SDSec solution with a dedicated orchestrator.

REFERENCES

[1] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

[2] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGPLAN Not.*, 48(4):461–472, March 2013.

[3] Adrian Waller, Ian Sandy, Eamonn Power, Efthimia Aivaloglou, Charalampos Skianis, Antonio Muñoz, and Antonio Maña. Policy based management for Security in Cloud Computing. In *FTRA International Conference on Secure and Trust Computing, Data Management, and Application*, pages 130–137. Springer, 2011.

[4] Alysson Bessani, Leucio A Cutillo, Gianluca Ramunno, Norbert Schirmer, and Paolo Smiraglia. The TClouds Platform: From the Concept to the Implementation of Bench. Scenarios. *ACM SIGOPS Operating Systems Review*, 48(2):13–22, 2014.

[5] G. Dreo, M. Golling, W. Hommel, and F. Tietze. ICEMAN: An Architecture for Secure Federated Inter-cloud Identity Management. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IFIP/IEEE IM 2013)*, May 2013.

[6] Olubisi Atinuke Runsewe. A Policy-Based Management Framework for Cloud Computing Security. Master's thesis, University of Ottawa, 2014.

[7] Maxime Compastié, Rémi Badonnel, Olivier Festor, Ruan He, and Mohamed Kassi-Lahlou. Towards a Software-Defined Security Framework for Supporting Distributed Cloud. In *Proceedings of the 11th IFIP International Conference on Autonomous Infrastructure, Management and Security (IFIP AIMS 2017), Zurich, Switzerland*, pages 47–61. Springer, July 2017.

[8] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey Mogul. Flowtags: Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in Software Defined Networking*, pages 19–24. ACM, 2013.

[9] Tommy Koorevaar. Dynamic Enforcement of Security Policies in Multi-Tenant Cloud Networks. Master's thesis, École Polytechnique de Montréal, 2012.

[10] Gaëtan Hurel, Rémi Badonnel, Abdelkader Lahmadi, and Olivier Festor. Behavioral and Dynamic Security Functions Chaining for Android Devices. In *Proceedings of the 11th IFIP/IEEE/In Assoc. with ACM SIGCOMM International Conference on Network and Service Management (IFIP/IEEE CNSM 2015)*, 2015.

[11] J. Park J. Jeong, H. Kim. Software-Defined Networking Based Security Services using Interface to Network Security Functions, October 2015.

[12] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *2008 IEEE Symposium on Security and Privacy (IEEE SP 2008)*, pages 233–247, May 2008.

[13] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ACM CCS 2007, pages 128–138, New York, NY, USA, 2007. ACM.

[14] Tal Garfinkel, Mendel Rosenblum, and others. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Ndss*, volume 3, pages 191–206, 2003.

[15] Andrei Bacs, Cristiano Giuffrida, Bernhard Grill, and Herbert Bos. Slick: An intrusion detection system for virtualized storage devices. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, (SAC 2016), pages 2033–2040, New York, NY, USA, 2016. ACM.

[16] J. Liedtke. On Micro-kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[17] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. *SIGOPS Oper. Syst. Rev.*, 29(5):251–266, December 1995.

[18] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. *SIGARCH Comput. Archit. News*, 39(1):291–304, March 2011.

[19] Dan Williams and Ricardo Koller. Unikernel monitors: extending minimalism outside of the box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, 2016.

[20] Alfred Bratterud, Andreas Happe, and Robert Anderson Keith Duncan. Enhancing Cloud Security and Privacy: The Unikernel Solution. In *Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, 19 February 2017-23 February 2017, Athens, Greece*. Curran Associates, 2017.

[21] Francisco J Ballesteros. The clive operating system. Technical report, Technical report, October, 2014.

[22] A. Bratterud, A. A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 250–257, Nov 2015.

[23] The rumprun unikernel and toolchain for various platform. Last visited in august 2017.

[24] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. OSv-optimizing the operating system for virtual machines. In *Proceedings of USENIX ATC'14: 2014 USENIX Annual Technical Conference*, page 61, 2014.

[25] Ocaml package manager. Last visited in august 2017.

[26] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 199–208, Sept 2006.

[27] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 128–138, New York, NY, USA, 2007. ACM.

[28] M. Pattaranantakul, R. He, A. Meddahi, and Z. Zhang. Secmano: Towards network functions virtualization (nfv) based security management and orchestration. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 598–605, Aug 2016.

[29] Song Luo and M. Ben Salem. Orchestration of software-defined security services. In *2016 IEEE International Conference on Communications Workshops (ICC)*, pages 436–441, May 2016.

[30] Mirage Skeleton: Examples of simple MirageOS Applications - Static website TLS, September 2017. https://github.com/mirage/mirage-skeleton/tree/master/applications/static_website_tls.

[31] Cohttp: Very lightweight HTTP Server using Lwt or Async, September 2017. https://github.com/mirage/ocaml-cohttp.

[32] Will Glozer. Wrk: Modern HTTP Benchmarking Tool, September 2017. https://github.com/wg/wrk.