A TOSCA-Oriented Software-Defined Security Approach for Unikernel-Based Protected Clouds

Maxime Compastié, Rémi Badonnel, Olivier Festor, Ruan He

A TOSCA-Oriented Software-Defined Security Approach for Unikernel-Based Protected Clouds

Maxime Compastié*†, Rémi Badonnel*, Olivier Festor*, Ruan He†
*Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France.
Emails: {remi.badonnel,olivier.festor}@loria.fr
†Orange Labs, Châtillon, France.
Emails: maxime.compastie@loria.fr,heruan@gmail.com

***Abstract*—Cloud infrastructures provide new facilities to build elaborated added-value services by composing and configuring a large variety of computing resources, from virtualized hardware devices to software products. In the meantime, they are further exposed to security attacks than traditional environments. The complexity of security management tasks has been increased by the multi-tenancy, heterogeneity and geographical distribution of these resources. They introduce critical issues for cloud service providers and their customers, with respect to security programmability and scenarios of adaptation to contextual changes. In this paper, we propose a software-defined security approach based on the TOSCA language, to enable unikernel-based protected clouds. We first introduce extensions of this language to describe unikernels and specify security constraints for their orchestrations. We then describe an architecture exploiting this extended version of TOSCA for automatically generating, deploying and adjusting cloud resources in the form of protected unikernels with a low attack surface. We finally detail a proof-of-concept prototype, and evaluate the proposed solution through extensive series of experiments.**

***Index Terms*—Cloud Environments, Software-Defined Security, Policy-Based Management, Security Orchestration, Unikernels**

I. Introduction

Distributed clouds contribute to the building of elaborated services based on multiple computing resources, such as virtual machines, network devices, software components [1], that may be spread across different infrastructures (*multi-cloud*) and stakeholders (*multi-tenancy*). This increases the complexity of management tasks, in particular with respect to security management. To deal with this complexity, software-defined security (SDSec) aims at supporting the programmability of security mechanisms that are used to protect resources offered by cloud infrastructures. It consists in the decoupling of two separate planes. The first one corresponds to the control plane which takes charge of security decisions, while the second one stands for the resource plane which includes the resources to be protected together with dedicated programmable security mechanisms (such as firewalls, intrusion detection systems, control access mechanisms) [2]. We have already analyzed the feasibility of such a security programmability layer for addressing multi-cloud and multi-tenant environments, through different realistic scenarios in [3]. The foundation of this layer relies on the SDSec logic to express and propagate security policies to the considered cloud resources, and on the autonomic paradigm to dynamically configure and adjust these mechanisms to distributed cloud constraints. We have also evaluated the benefits of using unikernel virtualization techniques to build and maintain specific cloud resources embedding security mechanisms in [4]. These lightweight virtual machines are built using a minimal set of libraries, enabling to reduce the attack surface.

We propose in this paper a TOSCA[1]-oriented software-defined security approach for supporting unikernel-based protected clouds. We exploit the TOSCA language, which supports the specification of cloud topologies and their orchestrations, in order to drive the integration and configuration of security mechanisms within cloud resources in an automatic manner. This contributes to leverage a security-by-design cloud, from the specification of multiple levels of security requirements to the generation (and regeneration) of specific unikernel-based virtual machines to address them. We extend the TOSCA language to describe unikernel components and specify these security requirements, and we detail the underlying framework, including a security orchestrator and a generator of unikernel virtual machines. The protected unikernels corresponding to the different orchestrated security levels can be generated in a proactive manner, and are compatible with the elasticity and on-demand properties of cloud resources. Our main contributions in this paper are (i) proposing and formalizing a software-defined security approach based on TOSCA for protecting cloud services, (ii) extending the TOSCA language to support our unikernel and multi-level security requirements, (iii) designing a framework capable of interpreting this extended language to generate and configure protected unikernels, and (iv) evaluating different SDSec strategies based on a proof-of-concept prototyping.

The remainder of this article is organized as follows. Section II presents existing work in the areas related to cloud security. Section III gives an overview of our TOSCA-oriented software-defined security approach. The extensions of the TOSCA language are described in Section IV, while Section V details the underlying framework exploiting them to enable security-by-design clouds. We evaluate the resulting enforcement in Section VI based on a proof-of-concept prototype. Section VII concludes the paper and points out future research perspectives.

[1]Topology and Orchestration Specification for Cloud Applications

II. Related Work

The security of cloud infrastructures and services is a challenge that has already been largely explored in the literature. Our work concerns the enforcement of security requirements that impact both the orchestration and the building of resources. From an orchestration perspective, software-defined networking has already contributed to multiple security management solutions. For instance, [2] introduces an experimental framework for configuring and benchmarking security rules applied to network flows. [5] proposes and evaluates delegation strategies for enforcing security mechanisms at the level of SDN switches, and [6] defines several access control methods taking into account the current risk levels. Such a programmability enables a more flexible composition of security functions. Approaches such as [7] support modular security functions that can be then composed into security chains to protect resources, but are often limited to specific enforcers. We showed in our previous work [3] an architecture for programmable security mechanisms in cloud infrastructures. It relies on the generation of specific resources based on unikernels, that integrate security mechanisms [4]. However, it should take advantage of orchestration languages, such as TOSCA, to drive the building and configuration of protected virtualized resources.

Virtualization methods constitute an interesting security enabler for protecting cloud environments [8]. In addition to isolation properties, they contribute to increase the control on resources by relying on hypervisors. Solutions such as [9], [10] define intrusion detection and integrity verification strategies for analyzing and controlling the behavior of virtualized resources. The minimization of virtual machines to the strict necessary components and libraries permits to reduce the attack surface. For instance, [11] proposes a Xen-based approach for removing unecessary components from virtual machines. This may also rely on containerization, but introducing new security issues, such as pointed in [12] and [13]. Library OSes, and more recently unikernels, offer new alternatives for such minimization. For instance, the LKL library [14] provides OS drivers as ready-to-go libraries. This enables legacy applications to have their own hardware resource management. Unikernels have abandoned any legacy OS support, and have fully redesigned the system architecture. Applications are capable to run as independent virtual machines [15], contributing to a simplified management as discussed in [16]. We argue in favor of exploiting unikernels to minimize the attack surface and showed how such unikernel-based virtual machines can be generated in an on-the-fly manner in [4].

It is essential to take into account the generation of such protected virtual machines into orchestration languages, in order to support cloud security, right from the design phase. Extensions are therefore required to describe the software components that compose unikernel virtual machines. Description languages have already been proposed in software engineering and service design areas. Historically, software programming has contributed to several description standards [17] to address internal software interactions amongst routines. Extensions have also been specified to integrate security requirements, such as [18]. These descriptions are often too fine-grained and do not address exploitation considerations. In the meantime, service design efforts provide another description scale. For instance, [19] provides service descriptions, by considering packages and their dependencies on Linux operating systems. Cloud orchestration languages, such as TOSCA [20], should take into account such descriptions. While they support the specification of the topology and orchestration of distributed cloud services, they only rely on off-the-shelves software descriptions [21]. This integration is an important lack to support security requirements from the design to the orchestration of services.

III. TOSCA-Oriented Software-defined Security Approach for Protecting Clouds

We propose a software-defined security approach based on the TOSCA language, in order to protect cloud services by using unikernel resources. We consider the TOSCA orchestration language rather than other alternatives, as it is fully in phase with the software-defined paradigm. It supports the description and orchestration of distributed cloud services, while abstracting technical implementation details. We extend it to describe unikernel resources and to specif security requirements according to different orchestrated levels. It then serves as an input for our security framework, which drives the design, deployment and operation of protected cloud resources, as depicted on Fig. 1. These resources rely on unikernels,

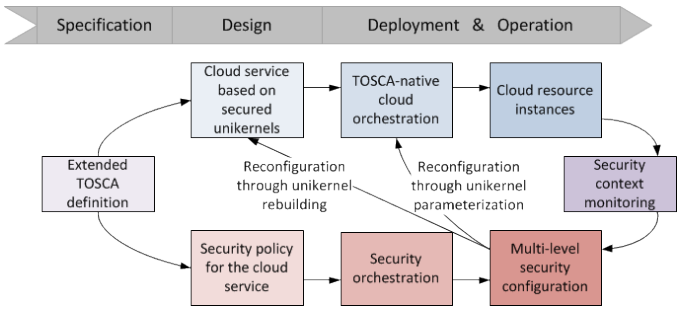

Figure 1. From the specification of TOSCA-based security requirements to the generation and operation of secured unikernel virtual machines.

that are lightweight virtual machines characterized by a low attack surface. They only contain the strict necessary software components and libraries, and integrate security mechanisms for their protection. The adaptation to contextual changes, in line with security levels, can be performed through the on-the-fly rebuilding of unikernel resources. In the following of the paper, we describe the extensions considered for the TOSCA language, as well as the framework supporting them to enforce security requirements on distributed cloud services.

IV. Extensions of the TOSCA Language

In order to support our software-defined security solution, we have first extended the TOSCA orchestration language,

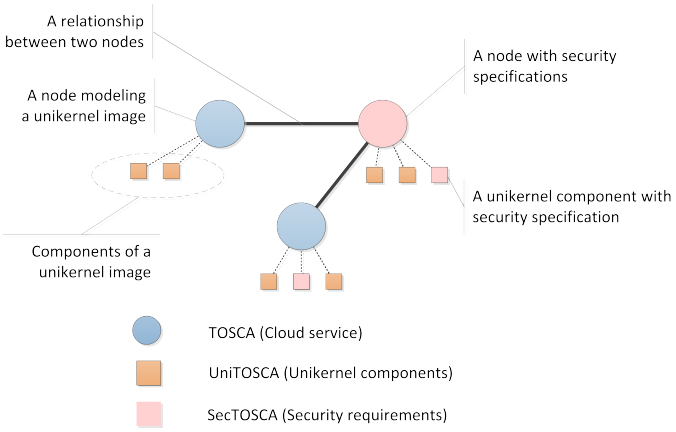

Figure 2. Extensions of the TOSCA language for describing unikernels (UniTOSCA) and specifying security requirements (SecTOSCA).

which provides a baseline for describing distributed and orchestrated cloud services. A topology representing these extensions is shown on Fig. 2. The first extension, called UniTOSCA, permits to refine the description level of TOSCA in the context of services implemented based on unikernels, by specifying them as a composition of software components. This description is then exploited to generate unikernels required by a cloud service. The second extension, called SecTOSCA, permits to specify security constraints in the TOSCA language. As previously mentioned, the scope of these constraints goes from a single unikernel to a whole cloud service. These constraints are used to enforce security over resources using dedicated mechanisms (firewalls, intrusion detection systems, access control). This enforcement is performed in a dynamic manner to adapt to contextual changes and takes benefit from the orchestration facilities offered by TOSCA. In particular, it is possible to specify different security levels to cope with various contexts. After presenting the key concepts of the TOSCA language, we will detail successively the two UniTOSCA and SecTOSCA specifications, with illustrative examples.

A. The TOSCA language

Typically, the TOSCA language serves as a support to cloud orchestrators for determining the resources to be instantiated and the operations to configure and operate them, in order to provide a given service. A cloud service is described by this language as a topology of resources (also called nodes) that are interconnected among them through links (also called relationships), as shown on Fig. 2 with the circle elements and their interconnections. It is then possible to specify orchestration procedures over this topology, such as starting, shutting down a node or changing a relationship. Each element (node or relationship) takes benefits from inheritance and template mechanisms offered by the language. Following an object-oriented paradigm, each type defines a class of elements sharing a common set of properties and interfaces, while a template defines a type with a set of pre-defined

```
1  unikernel_modules:
2    webserver_unikernel_type:
3      cohttp_lwt_server:
4        capabilities:
5          http_processing: tosca.capabilities.
               Endpoint
6        requirements:
7          - available_nic: unikernel.virtualenv.
               networking.nic
8      log:
9        capabilities:
10         console_logging: unikernel.virtualenv.
               console
11     dispatch:
12       properties:
13         https_port:
14           type: integer
15         http_port:
16           type: integer
17       capabilities:
18         webserver_front: tosca.capabilities.
               Endpoint
19       requirements:
20         - http_processing: tosca.capabilities.
               Endpoint
21         - console_logging: unikernel.
               virtualenv.console
22 [...]
23
24   node_templates:
25     webserver_unikernel:
26       type: webserver_unikernel_type
27       properties:
28         https_port: 4433
29         http_port:  8080
30 [...]
```

Figure 3. Extract of a UniTOSCA specification.

values affected to properties. An instance can be obtained from a template and corresponds to an implementation of the resources in a given contextual environment. Therefore, a node type, noted T_{node}, permits to infer a node template, noted T'_{node} and in turn a node instance, noted I_{node}. The language also permits to specify relationships in an implicit manner, using requirements (specifying what the node expects from other nodes on hosting infrastructures) and capabilities (specifying what the node may provide to other nodes on the infrastructures). A relationship type, noted $T_{relationship}$, permits to infer a relationship template, noted $T'_{relationship}$ and in turn a relationship instance, noted $I_{relationship}$. In its current form, the TOSCA specification is non normative with respect to the orchestration policy. Interfaces are typically used to define the operations performed on the nodes, following traditional workflow and process formalisms.

B. Describing unikernels

We propose an extension, called UniTOSCA, for describing unikernel virtual machines, as shown on Fig. 2 with the square elements. The purpose is both to increase the granularity of the language, and to drive the building of unikernels before their instantiation. For that, we introduce an additional element, called unikernel component m, in order to describe routines and compose them to elaborate unikernels. The relationships

among these components correspond to the dependencies that may exist among routines. These components are characterized by attributes and values that are configurable. However, taken separately, each of them cannot lead individually to a resource instance. A minimal set of routines is required to be composed in order to generate such an instance. In phase with our approach developed in [4], we take benefits from the simplified system architecture of unikernels to compose and build resources. This description of unikernel resources enables the orchestrator to take charge of the building and parameterization of these resources. The consistency of images generated from the descriptions relies on the dependency relationships among components, the satisfaction of these dependencies is checked before generating unikernel images that are used to elaborate the services. We consider a description of unikernel resources in phase with the description of the other TOSCA resources. An example of such a specification is given in Fig. 3, where we detail a unikernel resource, called `webserver_unikernel_type`, composed of three unikernel components. These components detail each routine (`cohttp_lwt_server`, `log`, `dispatch`) on which the resource is built, considering a finer granularity than regular TOSCA resources. The unikernel resource modeling infers the TOSCA type `webserver_unikernel_type`, which is used to define the TOSCA node `webserver_unikernel`. This fine granularity enables us to drive our software-defined security solution based on unikernels.

C. Specifying security requirements

We then introduce a SecTOSCA extension, represented by the striped elements in Fig. 2. This extension serves as a support to define the security policy, according to different security levels to be orchestrated. We consider a security orchestrator, complementary to the cloud resource orchestrator, taking charge of the security policy and the configuration of security functions. A security function (access control, firewalls, encryption mechanisms) corresponds to a feature aiming at enforcing a set of security requirements (access control lists, firewall rules) on the cloud resources. The security requirements can be specified at different scales, from a single unikernel to a whole cloud service. Some efforts have already shown the benefits of exploiting the non-normative orchestration policy of TOSCA in specific use cases, such as access control in [20]. We argue in favor of exploiting TOSCA to specify a security policy capable of covering the deployment and operation phases, but also the building of unikernel resources. This enables a security enforcement at an early stage through the generation of specific unikernel components and resources. In addition, we take benefits from the orchestration facilities of the TOSCA language in order to specify several security levels, as shown on Fig. 4. The extract showcases two orchestrated security levels, noted `default_security_level` and `critical_security_level` specified on nodes, but they might also apply on relationships. Our SecTOSCA extension permits an adaptation to contextual changes in two different manners. First, security mechanisms expose interfaces,

```
1  node_templates:
2    webserver_unikernel:
3     type: webserver_unikernel_type:
4      properties:
5        https_port: 4433
6        http_port:  8080
7        vm_security_level:
8          multi_level_security:
9            default_security_level:
                 regular_access_control
10           critical_security_level:
                 restricted_access_control
11
12 front_portal:
13    type: tosca.type.loadbalancer
14     properties:
15       lb_ddos_mitigation_level:
16         multi_level_security:
17           default_security_level:
                 regular_mitigation
18           critical_security_level:
                 paranoid_mitigation
19 [...]
```

Figure 4. Extract of a SecTOSCA specification.

enabling the security orchestrator to adjust their configuration parameters. This parameterization is performed on the instance of a node I_{node}, where the security level ($level_{sec}$) can be dynamically changed by the security orchestrator, as given by Equation 1.

$$configure_{sec} : \langle I_{node}, level_{sec} \rangle \longmapsto I_{node} \quad (1)$$

Second, the resources themselves can be rebuilt at runtime to cope with security constraints. In particular, this concerns unikernel resources embedding security mechanisms, which can be dynamically re-generated to cope with security constraints. To that purpose, we introduce a $building_{sec}$ operation to build a node type T_{node} based on unikernel modules $\langle m_1, ..., m_k \rangle$ according to a given security level, as shown by Equation 2. In that case, the result is not an instance, but a node type T_{node}, from which we can infer a node template T'_{node} and then a node instance I_{node}.

$$building_{sec} : \langle \langle m_1, ..., m_k \rangle, level_{sec} \rangle \longmapsto T_{node} \quad (2)$$

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

The SecTOSCA specification serves as a basis to define our security policy. We will then successively infer from it an enriched UniTOSCA specification, and a TOSCA specification. When we refer to Fig. 2 presenting the different extensions, it looks like we will taked the reverse path to obtain a TOSCA specification. The purpose is to guarantee the compatibility of our solution with TOSCA-native architectures.

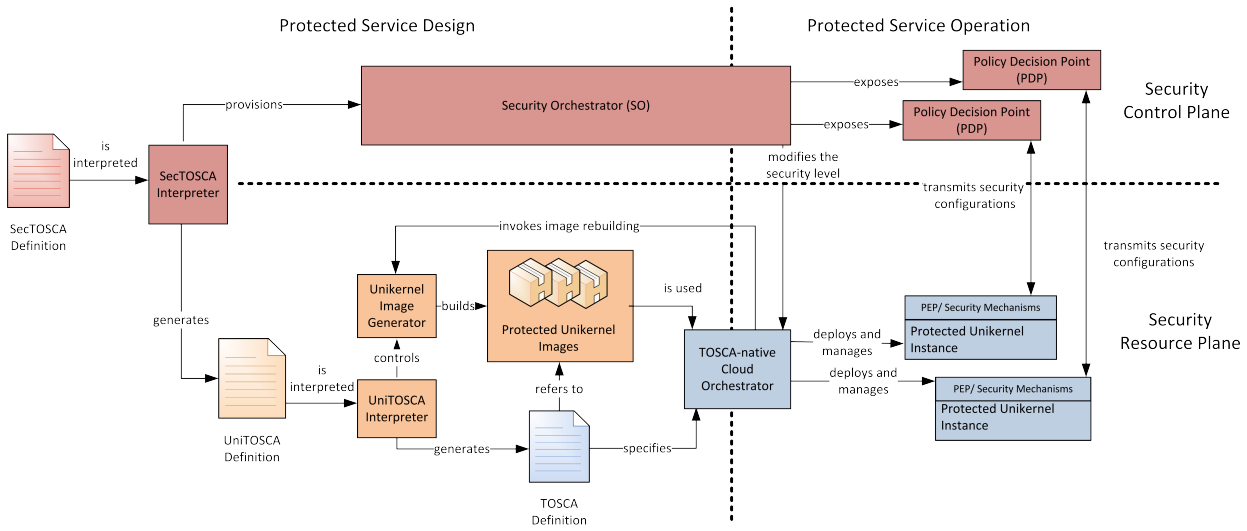

Figure 5. Overview of the TOSCA-oriented software-defined security framework for protecting cloud services.

V. Security Framework

We will now detail a security framework exploiting these extensions to protect cloud services. This framework, depicted in Fig. 5, is organized into three different tasks. First, it takes charge of the building of protected resources implementing the cloud service, as represented by yellow blocks on the figure. These resources have to be compatible with security programmability. Considering such a resource-centric strategy enables a fine-grained security enforcement. The UniTOSCA specification supports the design of protected unikernel images, embedding SDSec-capable security mechanisms. Second, it permits the management of security mechanisms with respect to security requirements specified by the SecTOSCA specification. Decoupling this management from protected resources, as represented by the two planes (security management plane and security resource plane) facilitates the support of distributed and heterogenous environments. This task corresponds to the red blocks in the figure. Third, the framework supports the adaptation to contextual changes. The purpose is to maintain security enforcement when changes occur over resources and their environments. The TOSCA-based orchestration, represented by blue blocks on the figure, addresses the whole life cycle of resources and can notify any changes that may occur on resources. We can also observe two axes on the figure: one horizontal axis referring to security programmability and distinguishing the security management plane from the security resource plane, and another vertical axis distinguishing the design/building of protected resources from their deployment and operation.

A. Main Components

The proposed framework includes several components depicted on the figure. It takes as input a SecTOSCA specification, serving as a starting point to build and orchestrate protected resources embedding security mechanisms. We detail now the role of the main components:

- **SecTOSCA Interpreter.** The role of this interpreter is to analyze a SecTOSCA specification and provide security requirements to the security orchestrator. It also produces a UniTOSCA specification detailing the unikernel resources to be generated with the embedded security mechanisms supporting security enforcement.
- **Security Orchestrator.** This component is responsible for translating security requirements into a consistent orchestration of security mechanisms that are distributed over resources to be protected. It interacts with policy decision points (PDP) capable of taking into account specific tenant and host requirements. These PDPs are then in charge of parameterizing Policy Enforcement Points (PEP) corresponding to the security mechanisms embedded on protected resources. As an example related to access control, security orchestrator is input with groups of entities allowed to access each others. The security orchestrator establishes access control lists aligned with these groups. The PDPs pick up the entries they require to establish their own list and configure the PEPs they are in charge of.
- **UniTOSCA Interpreter.** This interpreter analyzes a UniTOSCA specification and is capable to infer a TOSCA-native specification. It drives the generator of unikernels which builds protected unikernel resources from the description of unikernel components. The TOSCA specification refers to the unikernel images that are produced by the unikernel generator.
- **Generator of Unikernel Images.** This component is in charge of building unikernel images based on the description of unikernel components [4]. This description includes the components required to build the service, but also the ones required to protect it (e.g. embedded security mechanisms). It may also be invoked by the cloud orchestrator to address changes that may occur during the operation phase.

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

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

B. Interpreting SecTOSCA specifications

The SecTOSCA interpreter is responsible for extracting the security requirements from the SecTOSCA specification, semantically interpreting them into a set of rules and mechanisms bound to their enforcement, and enriching the TOSCA topology in order to integrate those mechanisms. We can distinguish two major tasks: (i) the enrichment of the TOSCA topology to support the enforcement of security functions, that can be seen as a policy refinement step enabling the integration of security mechanisms to the topology; and (ii) the provisioning of the security orchestrator with security rules to parameterize these mechanisms during their operation. The SecTOSCA interpreter is in charge of determining whether security functions can be enforced on a given topology representing a cloud service. For that purpose, it relies on the properties, capabilities and requirements of resources composing this topology. This includes both the TOSCA nodes and their TOSCA relationships. In a more formalized manner, it interprets a SecTOSCA specification, noted $D_{secTOSCA}$, and generates a UniTOSCA specification, noted $D_{uniTOSCA}$ as well as a policy P_{SO} for the security orchestrator, in accordance with Equation 3.

$$translate : D_{secTOSCA} \longmapsto \langle D_{uniTOSCA}, P_{SO} \rangle \quad (3)$$

The P_{SO} can typically correspond to control acces rules. This refinement is only possible if the set of security functions, noted $S(D_{secTOSCA})$, described in the SecTOSCA specification is enforceable on the SecTOSCA topology, noted $L(D_{secTOSCA})$, for a given execution environment e. This supposes that these functions are supported by the different types of resources of the topology, as described by Equation 4.

$$\begin{aligned} &\forall sf \in S(D_{secTOSCA}), \\ &\quad isEnforceable(sf, L(D_{secTOSCA}), e) \equiv \\ &\quad \forall T \in L(D_{secTOSCA}), isSupported(sf, T, e) \end{aligned} \quad (4)$$

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

C. Building and orchestrating unikernel resources

The UniTOSCA interpreter is in charge of driving the generator of protected unikernel images and of providing a TOSCA-native specification to the cloud orchestrator. The UniTOSCA specification describes the different modules required to build the unikernel images. This also includes modules implementing security mechanisms. The UniTOSCA interpreter therefore takes a UniTOSCA specification, noted $D_{uniTOSCA}$, and produces a TOSCA-native specification, noted D_{TOSCA}, together with the generation policy, noted P_{UG} for building protected unikernel images, in accordance with Equation 5.

$$translate : D_{uniTOSCA} \longmapsto \langle D_{TOSCA}, P_{UG} \rangle \quad (5)$$

The generation of a unikernel image relies on a set of k modules $m_1, m_2 ... m_k$, as previously given in Equation 2. This image permits to define a TOSCA type T_{node}, which is referred by the TOSCA-native specification D_{TOSCA}. The composition of modules (or unikernel components) to build a TOSCA type is only possible if these modules are consistent among them and with the execution environment. This means that all the requirements of modules, including security mechanisms, are satisfied by the capabilities provided by other modules or by the execution environment. The resulting type corresponds to the building of the unikernel images integrating security mechanisms. These node types, referred by the TOSCA specification, can then be deployed and orchestrated by the cloud orchestrator.

D. Adapting to contextual changes

In order to maintain or change the security level of the topology implementing a cloud service, the security orchestrator may proceed in two different ways. It may adjust the security rules exposed to the policy decision points (PDP), in order to modify the security configuration of resources. This corresponds to the $configure_{sec}$ operation previously defined. The scope of this option is relatively limited in a unikernel context, where we try to minimize the configurability of resources. It may also regenerate the unikernel-based resources in most cases. This corresponds to scenarios where most of rules may be statically implemented over the resources, integrating an internal PDP. This regeneration enables to modify the parameterization of the resources, but also to insert or remove security mechanisms from unikernel resources. This corresponds to the $building_{sec}$ operation previously defined. The regeneration of unikernel images is triggered by the security orchestrator through the cloud orchestrator. Unikernel images corresponding to different security levels may be generated in a proactive manner. Further information regarding the generation of unikernels can be found in [4], where we detail a generator framework.

VI. Implementation and Evaluation

Our prototyping has focused on the design and implementation of security mechanisms integrated to unikernels, which serve as a support for the evaluation of our framework through extensive series of experimentations. The prototype relies on a Policy Decision Point (PDP) taking local security decisions in

phase with the security orchestrator (SO), a Policy Enforcement Point (PEP) enforcing these decisions on a unikernel resource and a generator responsible for generating unikernel images embedding security mechanisms. Technically, the considered resource is a secured HTTP server over a MirageOS unikernel [22], which was extended to implement an access control mechanism including authorization and authentication. The security orchestrator comes from the MOON project. The generation of unikernel images is supported by a generator prototyped in Java, based on the RabbitMQ message broker. The PEP is implemented as an external OCaml module serving as a hook for control access on the resource. Unikernels are supervised by a uKVM monitor running over a KVM hypervisor. This permits to generate a specific VM monitor for each unikernel, whose unused features can be preventively disabled. Three different approaches have been implemented for the PDP: (i) an internal PDP, which is directly integrated to the unikernel image, (ii) a pushing PDP, which is external to the unikernel and interacts in a push-based manner, and (iii) a pulling PDP, which is external to the unikernel and interacts in a pull-based manner. From a qualitative perspective, we have demonstrated that TOSCA can be extended to both constrain the design and orchestrate the resources according to security requirements. The distributed cloud service they support therefore complies with security-by-design approach as its is framed by these requirements all along its life-cycle. We have also performed experiments on the following testbed. The host system features an Intel Xeon E5-1620 CPU at 8x3.6 GHz with 8 GB of RAM. It executes an up-to-date version of the Ubuntu 16.04 LTS distribution with the Linux kernel 4.4.0-112. Unikernel images are built with MirageOS 3.0.8 development kit. The virtual machines are instantiated with the uKVM monitor 0.2.2-1, with 1 vCPU and 512 MB of RAM. The PDP interacts with the security orchestrator based on the `moon_bouchon` interface (version 2018-01-30) from the MOON project. The performance evaluation has been done using the ApacheBench framework.

In a first series of experiments, we wanted to evaluate the performance of the three approaches, and in particular quantify the overhead induced by the outsourcing of the PDP (pull or push approaches) from the protected unikernel virtual machines. In our TOSCA-based security framework, the PDP outsourcing is a decision which is taken by the SecTOSCA interpreter, for a given security mechanism and according to a given service topology to be protected. The security requirements are transmitted by the SecTOSCA interpreter to the security orchestrator through the security policy P_{SO}. In the case of an internal PDP, the security requirements can be directly integrated to the unikernel images, through dedicated modules, enforcing a decision process internal to the protected unikernels. We first quantified the size of generated unikernel images implementing the three approaches. We expected the size of the image corresponding to an internal approach to be higher than the two other external approaches. In fact, it appears that the size of the internal approach is of 7.5 MB, with the size of the external approaches reaches 8.1 MB. This

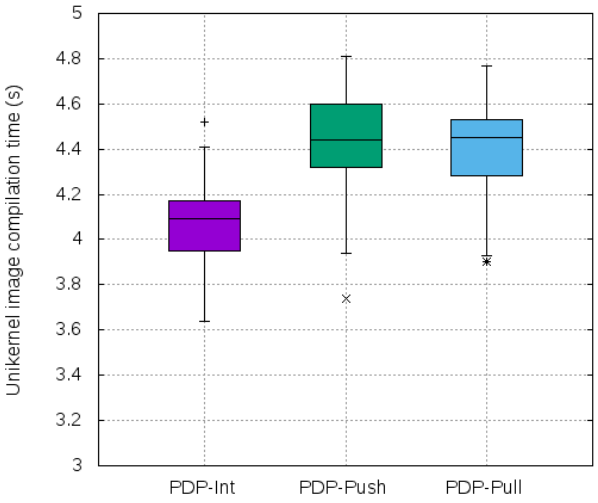

Figure 6. Generation time of protected unikernel images.

Figure 7. Network performance with the different approaches.

overhead of 600 KB for the external approaches is due to the additional modules required to support the interactions between the PDP and the PEP components of our solution. We also evaluated the time required for generating protected unikernels from the source code, with these different approaches, as detailed in Fig. 6. We observed a generation time of around 4.1 seconds with the internal approach, while it reaches 4.45 and 4.47 seconds with respectively the push-based and pull-based approaches. The overhead induced by the external approaches is again observed here, and can be correlated with the sizes of protected unikernel images.

We were also interested in quantifying the resource consumption of a protected unikernel virtual machine based on these different images. A particular focus has been given to the network performance presented on Fig. 7 and to the memory

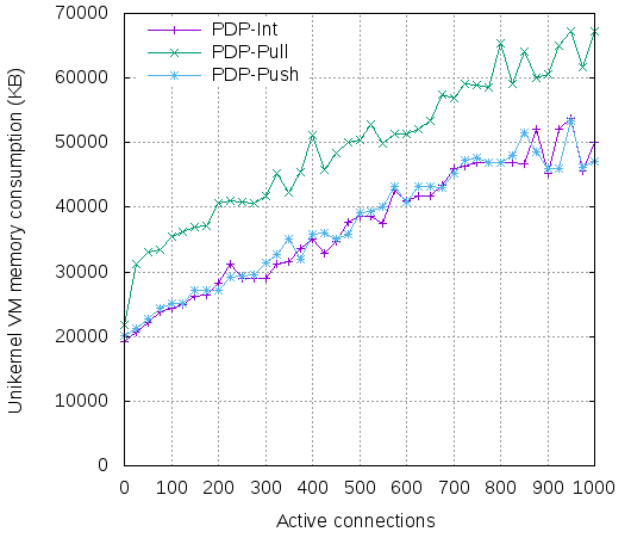

Figure 8. Memory consumption with the different approaches.

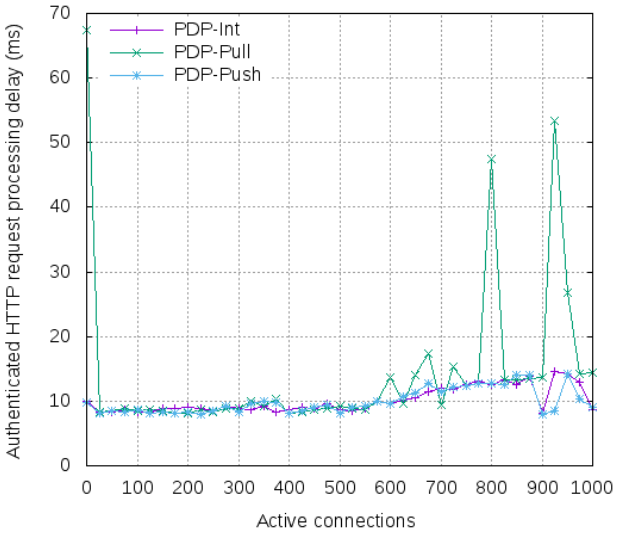

Figure 9. Authenticated HTTP processing time with the different approaches.

consumption given on Fig. 8. In both cases, we are varying the number of active connections from 0 to 1000 connections during experiments. We can observe on the first figure that the push-based and pull-based approaches introduce an overhead of respectively 41.3% and 1.2% in average, in comparison to the internal approach. The number of requests sent to the protected unikernel virtual machines is supposed to induce the same number of responses from them. The differences at high load are due to the congestion of the machine. A similar phenomenon is observable on the second figure, where the memory consumption related to the push-based and pull-based approaches generate an overhead of 32.7% and 1.2%, in comparison to the internal approach. For instance with 500 active connections, we obtain a resource consumption of 38 KB with the internal scenario, against 39 KB and 51 KB with respectively the push-based and pull-based scenarios. The overhead percentages among the different approaches are relatively stable, while varying the number of active connections. Finally, we compared the average time required for processing HTTP requests, including the authentication and authorization processing. The different approaches produce experimental results that are quite similar, with an average authenticated HTTP processing time of 11 ms, as shown on Fig. 9. Several peaks can be observed with the pull-based approach, in particular on the range from 600 and 1000 connections. This can be due to the external PDP which becomes a bottleneck with respect to authenticated requests, in such a pull-based scenario.

In a second series of experiments, we wanted to evaluate the performance of our solution with a pool of protected unikernels. It can be composed of unikernels implementing the pull-based and push-based approaches. We performed the same experiments than previously to quantify the memory consumption, the networking performance, and the authenticated HTTP request processing time with such a pool of 100 protected unikernels. During experiments, we varied the ratio of unikernel implementing each of the two approaches, and considered a workload from 1 to 1000 incoming concurrent connections. As we expected, the memory consumption increases when the incoming workload is growing. In particular, we noticed that a higher proportion of push-based unikernel virtual machines can shrink the memory consumption by 22% on average during experiments. These results can be explained by the requirement for unikernel virtual machines using the pull-based approach to include further features in their networking stack (e.g. DNS resolver, HTTP client library) compared to the push-based approach. The same observation has been done for the networking performance. A higher ratio of push-based unikernel virtual machines improves the performances. However, the impact might be less significant, when considering an enhanced pull-based approach integrating caching facilities. We also evaluated the performance on authenticated HTTP processing with the different workload scenarios. The experimental results are detailed on The results showed that increasing the number of push-based unikernels decreases such a processing time for all the considered scenarios. Most of the time, the processing time is growing with the number of concurrent active connections. Except for the scenarios with a ratio of more than 90% of push-based unikernels, the one connection workload induces a significantly longer processing time than any other workload scenarios. These results may be due to the DNS resolution time, which takes a more important part in the overall request processing.

In a last series of experiments, we were interested in evaluating the time required for the propagation and enforcement of a security policy with the different approaches. We quantified the delay time between the update of the security policy specification and its enforcement on the cloud resource. The obtained results showed a behavior close to linearity with respect to the size of the policy (control access list). We

expected that the delay times obtained with the push-based approach will be better than the ones induced by the pull-based approach, which was the case. The push-based approach leads to a shorter delay time, in comparison to the pull-based approach showing an overhead of 48,6% on average. The delay times obtained with the internal approach are sensitively higher than the two other approaches, and are more distributed. The overhead induced by the internal approach is 5.82 times higher due to the compilation time. The update of the security policy requires to regenerate the unikernel images implementing the internal approach. Proactive strategies permit to anticipate changes in the security policy and to regenerate unikernel images at an early stage. From a scalability viewpoint, the pull-based approach operates successfully for the different experimental scenarios from 0 to 50,000 security rules, while it generated additional network traffic. The push-based approach was capable of supporting up to 33,500 rules with a single unikernel, due to memory depletion, while the internal approach supported up to 21,750 rules with a single unikernel, due to unikernel compilation. This corresponds to a high number of security rules with respect to a single unikernel scenario. These experiments have shown the compared performances of three different approaches to implement our TOSCA-oriented security framework.

VII. Conclusions

We have proposed in this paper a software-defined security approach based on the TOSCA language, in order to support the protection of cloud resources using unikernel techniques. The TOSCA language enables the specification of cloud services and their orchestration. We have extended this language to drive the integration and configuration of security mechanisms within cloud resources, at the design and operation phases, according to different security levels. We rely on unikernel techniques to elaborate cloud resources using a minimal set of libraries, in order to reduce the attack surface. We have designed a framework to interpret this extended language and to generate and configure protected unikernel virtual machines, in accordance with contextual changes. The adaptation is typically performed through the regeneration of protected unikernel virtual machines in a dynamic manner. Unikernel images corresponding to the different security levels can be proactively generated by the security framework. We have implemented a proof-of-concept prototype to evaluate the performances of our solution, with a focus on unikernel security mechanisms. In particular, we have quantified the benefits and limits of three different approaches to support their integration. As future work, we are interested in performing complementary experiments of our security framework in a massively distributed environment. We are also planning to investigate further optimization methods and techniques for supporting the regeneration and/or the reconfiguration of security mechanisms. The purpose is to minimize the costs induced by the regeneration of unikernels, while keeping a low attack surface in line with the required security levels.

References

[1] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

[2] A. Darabseh, M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, M. Vouk, and A. Rindos. SDSecurity: A Software Defined Security experimental framework. In *Proc. of the 2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 1871–1876, June 2015.

[3] M. Compastié, R. Badonnel, O. Festor, R. He, and M. Kassi-Lahlou. Towards a Software-Defined Security Framework for Supporting Distributed Cloud. In *Proc. of the 11th IFIP International Conference on Autonomous Infrastructure, Management and Security (AIMS)*, pages 47–61, July 2017.

[4] M. Compastié, R. He, M. Kassi-Lahlou, R. Badonnel, and O. Festor. Unikernel-based Approach for Software-Defined Security in Cloud Infrastructures. In *Proc. of the 2018 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Taipei, Taiwan, April 2018.

[5] J. Liu, Y. Li, H. Wang, D. Jin, L. Su, L. Zeng, and T. Vasilakos. Leveraging Software-defined Networking for Security Policy Enforcement. *Information Sciences*, 327:288 – 299, 2016.

[6] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic Access Control for Enterprise Networks. In *Proc. of the ACM Workshop on Research on Enterprise Networking*, New York, NY, USA.

[7] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Proc. of the NDSS Symposium*, 2013.

[8] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. In *Proc. of the Second International Conference on Comp. and Net. Technology, 2010*.

[9] A. Bacs, C. Giuffrida, B. Grill, and H. Bos. Slick: An Intrusion Detection System for Virtualized Storage Devices. In *Proc. of the 31st Annual ACM Symposium on Applied Computing*, pages 2033–2040.

[10] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proc. of the ACM Symposium on Operating Systems Principles (SIGOPS)*.

[11] F. Manco, Costin Lupu, Florian S., J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My VM is Lighter (and Safer) Than Your Container. In *Proc. of the 26th Symposium on Operating Systems Principles*, pages 218–233.

[12] Kata containers - the speed of containers, the security of VMs, July 2018. https://katacontainers.io/. Last visited in July 2018.

[13] Gvisor: Container runtime sandbox. https://github.com/google/gvisor. Last visited in July 2018.

[14] O. Purdila, L. A. Grijincu, and N. Tapus. LKL: The Linux Kernel Library. In *Proc. of the IEEE RoEduNet Int. Conference, 2010.*

[15] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proc. of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 461–472.

[16] A. Walla. Live updating in Unikernels. Master's thesis, 2017.

[17] K. K. Lau and Z. Wang. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, October 2007.

[18] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. *«UML» 2002—The Unified Modeling Language*, pages 426–441, 2002.

[19] 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 *Proc. of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 199–208, September 2006.

[20] D. Palma and T. Spatzier. Topology and orchestration specification for cloud applications (TOSCA). *Organization for the Advancement of Structured Information Standards (OASIS), Tech. Rep*, 2013.

[21] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. TOSCA: Portable Automated Deployment and Management of Cloud Applications. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Advanced Web Services*, pages 527–549. New York, NY, 2014.

[22] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. *SIGPLAN Notices*, 48(4):461–472, March 2013.