Auditing Security Compliance of the Virtualized Infrastructure in the Cloud: Application to OpenStack

Taous Madi
CIISE
Concordia University
Montreal, QC, Canada
t_madi,su_majumencs.concordia.ca

Suryadipta Majumdar
CIISE
Concordia University
Montreal, QC, Canada
su_majum@encs.concordia.ca

Yushun Wang
CIISE
Concordia University
Montreal, QC, Canada
yus_wang@encs.concordia.ca

Yosr Jarraya
Ericsson Security Research
Ericsson Canada
Montreal, QC, Canada
yosr.jarraya@ericsson.com

Makan Pourzandi
Ericsson Security Research
Ericsson Canada
Montreal, QC, Canada
makan.pourzandi@ericsson.com

Lingyu Wang
CIISE
Concordia University
Montreal, QC, Canada
wang@encs.concordia.ca

ABSTRACT

Cloud service providers typically adopt the multi-tenancy model to optimize resources usage and achieve the promised cost-effectiveness. Sharing resources between different tenants and the underlying complex technology increase the necessity of transparency and accountability. In this regard, auditing security compliance of the provider's infrastructure against standards, regulations and customers' policies takes on an increasing importance in the cloud to boost the trust between the stakeholders. However, virtualization and scalability make compliance verification challenging. In this work, we propose an automated framework that allows auditing the cloud infrastructure from the structural point of view while focusing on virtualization-related security properties and consistency between multiple control layers. Furthermore, to show the feasibility of our approach, we integrate our auditing system into OpenStack, one of the most used cloud infrastructure management systems. To show the scalability and validity of our framework, we present our experimental results on assessing several properties related to auditing inter-layer consistency, virtual machines co-residence, and virtual resources isolation.

Keywords

Cloud, Virtualization, OpenStack, Security Auditing, Formal Verification, Co-residence, Isolation

1. INTRODUCTION

Several security challenges faced by the cloud, mainly the loss of control and the difficulty to assess security compliance of the cloud providers, leave potential customers reluctant towards its adoption. These challenges stem from cloud-enabling technologies and characteristics. For instance, virtualization introduces complexity, which may lead to new vulnerabilities (e.g., incoherence between multiple management layers of hardware and virtual components). At the same time, concurrent and frequent updates needed to meet various requirements (e.g., workload balancing) may create even more opportunities for misconfiguration, security failures, and compliance compromises. Cloud elasticity mechanisms may cause virtual machines (VMs) belonging to different corporations and trust levels to interact with the same set of resources, causing potential security breaches [30]. Therefore, cloud customers take great interest in auditing the security of their cloud setup.

Security compliance auditing provides proofs with regard to the compliance of implemented controls with respect to standards as well as business and regulatory requirements. However, auditing in the cloud constitutes a real challenge. First, the coexistence of a large number of virtual resources on one side and the high frequency with which they are created, deleted, or reconfigured on the other side, would require to audit, almost continuously, a sheer amount of information, growing continuously and exponentially [13]. Furthermore, a significant gap between the high-level description of compliance recommendations (e.g., Cloud Control Matrix (CCM) [14] and ISO 27017 [20]) and the low-level raw logging information hinders auditing automation. More precisely, identifying the right data to retrieve from an ever increasing number of data sources, and correctly correlating and filtering it constitute a real challenge in automating auditing in the cloud.

We propose in this paper to focus on auditing security compliance of the cloud virtualized environment. More precisely, we focus primarily on virtual resources isolation based on structural properties (e.g., assignment of instances to physical hosts and the proper configuration of virtualization mechanisms), and consistency of the configurations in different layers of the cloud (infrastructure management layer, software-defined networking (SDN) controller layer, virtual layer and physical layer). Although there already exist var-

CODASPY'16, March 09-11, 2016, New Orleans, LA, USA

DOI: http://dx.doi.org/10.1145/2857705.2857721

ious efforts on cloud auditing (a detailed review of related works will be given in Section 2), to the best of our knowledge, none has facilitated automated auditing of structural settings of the virtual resources while taking into account the multi-layer aspects.

Motivating example. The following illustrates the challenges to fill the gap between the high-level description of compliance requirements as stated in the standards and the actual low-level raw audit data. In CCM [14], the control on Infrastructure & Virtualization Security Segmentation recommends *"isolation of business critical assets and/or sensitive user data, and sessions"*. In ISO 27017 [20], the requirement on segregation in virtual computing environments mandates that *"cloud service customer's virtual environment should be protected from other customers and unauthorized users"*. Moreover, the segregation in networks requirements recommends *"separation of multi-tenant cloud service customer environments"*.

Clearly any overlap between different tenants' resources may breach the above requirements. However, in an SDN/-Cloud environment, verifying the compliance with the requirements requires gathering information from many sources at different layers of the cloud stack: the cloud infrastructure management system (e.g., OpenStack [27]), the SDN controller (e.g., OpenDaylight [24]), and the virtual components and verifying that effectively compliance holds in each layer. For instance, the logging information corresponding to the virtual network of tenant `0848cc1999-e542798` is available from at least these different sources:

- Neutron databases, e.g., records from table "Routers" associating tenants to their virtual routers and interfaces of the form `0848cc1999e542798 (tenants_id)` ‖ `420fe1cd-db14-4780` (vRouter_id) ‖ `6d1f6103-9b7a-4789-ab16 (vInterface_id)`.

- Nova databases, e.g., records from table "Instances" associating VMs to their owners and their MAC addresses as follows: `0721a9ac-7aa1-4fa9 (VM_ID)` ‖ `0848cc1999e542798 (tenants_id)` and `fa:16:-3e:cd:b5:e1 (MAC)` ‖ `0721a9ac-7aa1-4fa9(VM_ID)`.

- Open vSwitch databases information, where ports and their associated tags can be fetched in this form `qvo4429c50c-9d (port_name)` ‖ `1084(VLAN_ID)`.

As illustrated above, it is difficult to identify all the relevant data sources and to map information from those different sources at various layers to the standard's recommendations. Furthermore, potential inconsistencies in these layers make auditing tasks even more challenging. Additionally, as different sources may manipulate different identifiers for the same resource, correctly correlating all these data is critical to the success of the audit activity.

To facilitate automation, we present a complied list of security properties relevant to the cloud virtualized environment that maps into different recommendations described in several security compliance standards in the field of cloud computing. Our auditing approach encompasses extracting configuration and logged information from different layers, correlating the large set of data from different origins, and finally relying on formal methods to verify the security properties and provide audit evidence. We furthermore implement the verification of these properties and show how the data can be collected and processed in the cloud environment with an application to OpenStack. Our approach shows scalability as it allows auditing a dataset of 300,000 virtual ports, 24,000 subnets, and 100,000 VMs in less than 8 seconds.

The main contributions of our paper are as follows:

- To the best of our knowledge, this is the first effort on auditing cloud virtualized environment from the structural point of view taking into account consistency between multiple control layers in the cloud.

- We identify a list of security properties from the literature that may fill the gaps between security standards recommendations and actual compliance validation and allows audit automation.

- We report real-life experience and challenges faced when trying to integrate auditing and compliance validation into OpenStack.

- We conducted experiments whose results show scalability and efficiency of our approach.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 describes our methodology. Section 4 provides an overview of our auditing framework. Section 5 describes the formalization of security properties. Section 6 details the integration of our auditing framework into OpenStack. Section 7 experimentally evaluates the performance of our approach. Finally, we conclude our paper discussing future directions in Section 8.

2. RELATED WORK

To the best of our knowledge, no work has been tackling auditing consistency views between different layers and structural configuration of virtualization in the cloud. Several works target the verification of forwarding and routing rules, particularly in OpenFlow networks (e.g., [34, 16]). For instance, Libra [34] uses a divide and conquer technique to verify forwarding tables in large networks. It encompasses a technique to capture stable and consistent snapshots of the network state and a verification approach based on graph search techniques that detects loops, black-holes and other reachability failures. Sphinx [16] enables incremental real-time network updates and constraints validation. It allows detecting both known and potentially unknown security attacks on network topology and data plane forwarding. These works are complementary to our work as they aim at verifying operational properties of networks including reachability, isolation and absence of network misconfiguration (e.g., loops, black-holes, etc.). However, they target mainly SDN environments and not necessarily the cloud.

In the context of cloud auditing, several works (e.g., [9, 29]) focus on firewalls and security groups. Probst et al. [29] present an approach for the verification of network access controls implemented by stateful firewalls in cloud computing infrastructures. Their approach combines static and dynamic verification with a discrepancy analysis of the obtained results against the clients' policies. However, the proposed approach does not address challenges related to the cloud such as virtualization and resources sharing. Bleikertz [9] analyzes Amazon EC2 cloud infrastructure using reachability graphs and vulnerability discovery and builds attack

graphs to find the shortest paths, which represents the critical attack scenarios against the cloud. These works are complementary to our work as they only focus on layer 3 and layer 4 components, whereas layer 2 components can be at the origin of several problems, which is addressed in our work.

Other works focus on virtualization aspects. For instance, Bleikertz et al. [10] propose a general purpose security analysis of the virtualized cloud infrastructure based on formal verification techniques (e.g., model checking, theorem proving, etc.). Therein, the configuration of the infrastructure is captured using graph-based representations and security goals are expressed using VALID specifications [**?**]. The automated analysis mechanisms allow checking configuration states (against zone isolation and single point of failure problems) and configuration states changes against high-level security policies. In contrast to our work, their work is more oriented towards detecting attack states than auditing security controls compliance.

Bleikertz et al. [11] extend the previous work to tackle near-real time security analysis of the virtualized infrastructure in the cloud. Their objective is mainly the detection of configuration changes that impact the security. A differential analysis based on computing graph deltas (e.g, added/removed nodes and edges) is proposed based on change events. The graph model is maintained synchronized with the actual configuration changes through probes that are deployed over the infrastructure and intercept events that may have a security impact. In contrast to our work, they aim at the verification of operational properties such as reachability analysis. Furthermore, their analysis relies only on the information on the virtualized infrastructure configuration provided by the cloud infrastructure management system, namely VMware, and thus they do not verify consistency between the cloud infrastructure management system and the actual virtual implementation. In our case, we use direct querying of virtual resources to assess multi-layer configurations consistency.

In [17], an autonomous agent-based incident detection system is proposed. The system detects abnormal infrastructure changes based on the underlying business process model. The framework is able to detect cloud resource and account misuse, distributed denial of service attacks and VM breakout. This related work is more oriented towards monitoring changes in cloud instances and infrastructures and evaluating the security status with respect to security business flow-aware rules.

Xu et al. [33] investigate network inconsistencies between network states extracted from OpenStack and the configuration of network devices. They use Binary Decision Diagrams (BDDs) to represent and verify these states. Similarly to our work, they tackle inconsistency verification. However, this represents only one example of the problem we tackle. Furthermore, we are interested in auditing, thus our approach supports a wider view than simple verification, where log files are as important source of information as configuration.

Congress [2] is an open policy framework for cloud services. It enforces policies expressed by tenants and then monitors the state of the cloud to check its compliance. Furthermore, Congress attempts to correct policy violations when they occur. Our work shares the policy inspection aspect with Congress. Thus, the properties we are audit in the current paper can be integrated in Congress. The multi-domain cloud at the user level with OpenStack as an application is audited in Majumdar et al. [21], whereas this paper deals with different layers and structural configuration of virtualization.

3. METHODOLOGY

In this section, we present some preliminaries and describe our approach for auditing and compliance validation.

3.1 Threat Model

We assume that the cloud infrastructure management system has implementation flaws and vulnerabilities, which can be potentially exploited by malicious entities. For instance, a reported vulnerability in OpenStack Nova networking service, OSSN-0018/2014 [26], allows a malicious VM to reach the network services running on top of the hosting machine, which may lead to serious security issues. We trust cloud providers and administrators, but we assume that some cloud users and operators may be malicious [12]. We trust the cloud infrastructure management system for the integrity of the audit input data (e.g., logs, configurations, etc.) collected through API calls, events notifications, and database records (existing techniques on trusted auditing may be applied to establish a chain of trust from TPM chips embedded inside the cloud hardware to auditing components, e.g., [7]). We assume that not all tenants trust each other. They can either require not to share any physical resource with all the other tenants, or provide a white (or black) list of trusted (or untrusted) customers that they are (not) willing to share resources with. Although our auditing framework may catch violations of specified security properties due to either misconfiguration or exploits of vulnerabilities, our focus is not on detecting specific attacks or intrusions.

EXAMPLE 1. *For illustrating purposes in our running example, we consider two tenants. Tenant Alpha can be exposed to malicious outsiders and insiders. A malicious insider could be either an adversary (tenant Beta) sharing the same cloud resources with tenant Alpha or a malicious operator with a higher access privilege.*

3.2 Modeling the Virtualized Infrastructure

In a multi-tenant cloud Infrastructure as a Service (IaaS) model, the provider's physical and virtual resources are pooled to serve on demands from multiple customers. The IaaS cloud reference model [32] consists of two layers: The physical layer composed of networking, storage, and processing resources, and the virtualization layer that is running on top of the physical layer and enabling infrastructure resources sharing. Figure 1 refines the virtualization layer abstraction in [32] by considering tenant specific virtual resources such as virtual networks and VMs. Accordingly, a tenant can provision several VM instances and virtual networks. VMs may run on different hosts and be connected to many virtual networks through virtual ports. Virtualization techniques are used to ensure isolation among multiple tenants' boundaries. Host virtualization technologies enable running many virtual machines on top of the same host. Network virtualization mechanisms (e.g., VLAN and VXLAN) enable tenants' network traffic segregation, where virtual networking devices (e.g., Open vSwitches) play a vital role in connecting VM instances to their hosting machines and to virtual networks.

In addition to these virtual and physical resources illustrated as nodes, Figure 1 shows the relationships between tenants' specific resources and cloud provider's resources. These relations will be used in section 5 for the formalization of both the virtualized infrastructure model and the security properties. For instance, *IsAttachedOnPort* is a relationship with arity 3. It attaches a VM to a virtual subnet through a virtual port. This model can be refined with several levels of abstraction based on the properties to be checked.

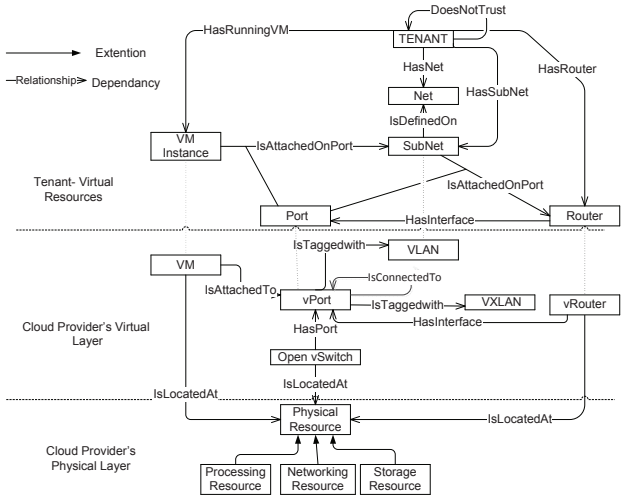


Figure 1: A generic model of the virtualized infrastructures in the cloud

3.3 Cloud Auditing Properties

We classify virtualization related-properties into two categories: Structural and operational properties. Structural properties are related to the static configuration of the virtualized infrastructure such as the assignment of instances to physical hosts, the assignment of virtual networking devices to tenants, and the proper configuration of isolation mechanisms such as VLAN configuration of each port. Operational properties are related to the forwarding network functionality. Those are mainly reachability-related properties such as loop-free forwarding and absence of black holes. Since the latter category has received significant attention in the literature (e.g. [34], [9], [16]), the former category constitutes the main focus of the current paper. As the major goal of this work is to establish a bridge between high-level guidelines in the security standards and low-level logs provided by current cloud systems, we start by extracting a list of concrete security properties from those standards and the literature in order to more clearly formulate the auditing problem. Table 1 presents an excerpt of the list of security properties we consider for auditing relevant standards (e.g., ISO 27002 [19], CCM [14]). Therein, we also classify properties based on their relevance to the stakeholders. In the following, we provide a brief description followed by an illustrating example for the sample properties, namely, absence of common ownership of resources, no co-residence, and topology consistency. More detailed properties' descriptions are provided in Appendix A.

Virtual resources isolation (no common ownership). The no common ownership property aims at verifying that no virtual resource is co-owned by multiple tenants.

EXAMPLE 2. *(No common ownership) Neutron OpenStack service allows tenant administrators to create virtual routers to connect their subnets. It also allows creating interfaces on those routers. OSSA-2014-008 [25] is a Neutron vulnerability that allows a tenant to create a virtual port on another tenant's router without checking his identity. Exploiting such vulnerability leads to the violation of no common ownership property, as virtual ports are tenant specific resources that should not be shared among tenants. As illustrated in Figure 2, Port_84 belongs to Beta as he is the initiator of the command for port creation. Since the port is connected to Router_A5 initially belonging to Alpha, the port would be considered as a common resource for both tenants.*

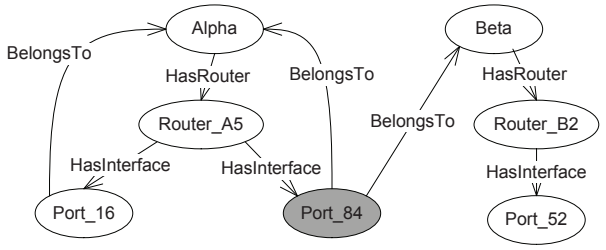


Figure 2: Model instance for *No common ownership*

No co-residence. This property consists of verifying the effective physical isolation of tenants' resources.

EXAMPLE 3. *(No co-residence) According to [36], it is possible to successfully identify the location of a target VM and to trigger the creation of malicious VMs to co-reside in the same host as the target VM. Once co-located with its target, a malicious VM can exploit vulnerabilities in the hypervisor[28] or use side channel techniques to violate other guests confidentiality and integrity and the hypervisor's availability. Suppose that VM_01 and VM_02 are two VMs belonging to tenant Alpha and running initially at Compute Node_85, and VM_03 is owned by tenant Beta and runs initially at Compute Node_96. Assume that tenant Alpha requires to physically isolate its VMs from those of tenant Beta, however, for load balancing reasons, VM_02 is migrated from Compute Node_85 to Compute Node_96. It is clear that this VM migration event will lead to a violation of the physical isolation property (See Figure 3).*

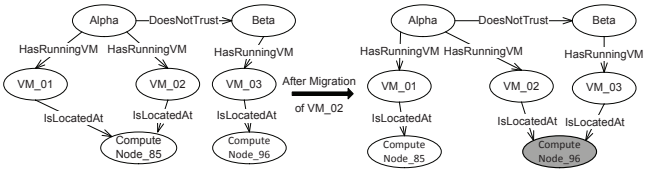


Figure 3: Model instance for *No co-residence*

Topology consistency. Topology consistency consists of checking whether the topology view in the cloud infrastructure management system, matches the actual implemented topology, while considering different mappings between the physical infrastructure, the virtual infrastructure, and the tenants' boundaries.

EXAMPLE 4. *(Port consistency) We suppose that a malicious insider managed to deliberately create a virtual port vPort_40 on Open_vSwitch_56 and label it with the VLAN identifier VLAN_100 that is already assigned to tenant Alpha. This would allow the malicious insider to sniff tenant's Alpha traffic by mirroring the VLAN_100 traffic to the created port vPort_40. This clearly would lead to the violation of the network isolation property.*

As illustrated in Figure 4, we build two views of the virtualized topology: The actual topology is built based on data collected directly from the networking devices running at the virtualization layer (Open vSwitches), and the perceived topology is obtained from the infrastructure management layer (Nova and Neutron OpenStack databases). The dashed lines map one to one the entities between the two topologies (not all the mappings are shown for more readability). We can observe that vPort_40 is attached to VLAN_100, which maps to Net_01 (tenant Alpha's network), but there is no entity at the infrastructure management layer that maps to the entity vPort_40 at the virtualization layer, which reveals a potential security breach.

4. AUDIT READY CLOUD FRAMEWORK

Figure 5 illustrates a high-level architecture of our auditing framework. It has five main components: data collection and processing engine, compliance validation engine, audit report engine, dashboard, and audit repository database. The framework interacts mainly with the cloud management system, the cloud infrastructure system (e.g., OpenStack), and elements in the data center infrastructure to collect various types of audit data. It also interacts with the cloud tenant to obtain the tenant requirements and to provide the tenant with the audit result. Tenant requirements encompass both general and tenant-specific security policies, applicable standards, as well as audit queries. For the lack of space, we will only focus on the following major components.

Our data collection and processing engine is composed of two sub-engines: the collection engine and the processing engine. The collection engine is responsible for collecting the required audit data in a batch mode, and it relies on the cloud management system to obtain the required data. The role of the processing engine is to filter, format, aggregate, and correlate this data. The required audit data may be distributed throughout the cloud and in different formats. The processing engine must pre-process the data in order to provide specific information needed to verify given properties. The last processing step is to generate the code for compliance validation and then store it in the audit repository database to be used by the compliance validation engine. The generated code depends on the selected back-end verification engine.

The compliance validation engine is responsible for performing the actual verification of the audited properties and the detection of violations, if any. Triggered by an audit request or updated inputs, the compliance validation engine invokes our back-end verification and validation algorithms. We use formal methods to capture formally the system model and the audit properties, which facilitates automated reasoning and is generally more practical and effective than manual inspection. If a security audit property fails, evidence can be obtained from the output of the verification back-end. Once the outcome of the compliance validation is ready, audit results and evidences are stored in the audit repository database and made accessible to the audit reporting engine. Several potential formal verification engines can serve our needs, and the actual choice may depend on the property being verified.

5. FORMAL VERIFICATION

As a back-end verification mechanism, we propose to formalize audit data and properties as Constraint Satisfaction Problems (CSP) and use a constraint solver, namely Sugar [31], to validate the compliance. CSP allows formulation of many complex problems in terms of variables defined over finite domains and constraints. Its generic goal is to find a vector of values (a.k.a. assignment) that satisfies all constraints expressed over the variables. If all constraints are satisfied, the solver returns SAT, otherwise, it returns UNSAT. In the case of a SAT result, a solution to the problem is provided. The key advantage of using CSP comes from the fact that it enables uniformly presenting the system's setup and specifying the properties in a clean formalism (e.g., First Order Logic (FOL) [8]), which allows to check a wide variety of properties [35]. Moreover using CSP avoids the state space traversal, which makes our approach more scalable for large data sets.

5.1 Model Formalization

Depending on the properties to be checked, we encode the involved instances of the virtualized infrastructure model as CSP variables with their domains definitions (over integer), where instances are values within the corresponding domain. For example, $Tenant$ is defined as a finite domain ranging over integer such that *(domain* $TENANT$ 0 max_tenant*)* is a declaration of a domain of tenants, where the values are between 0 and max_tenant. Relations between classes and their instances are encoded as relation constraints and their supports, respectively. For example, $HasRunningVM$ is encoded as a relation, with a support as follows: (relation $HasRunningVM$ 2 ($supports(vm1, t1)(vm2, t2)$)). The support of this relation will be fetched and pre-processed in the data processing step. The CSP code mainly consists of four parts:

- *Variable and domain declaration.* We define different entities and their respective domains. For example, t is a variable defined over the domain $TENANT$, which range over integers.
- *Relation declaration.* We define relations over variables and provide their support from the audit data.
- *Constraint declaration.* We define the negation of each property in terms of predicates over the involved relations to obtain a counter-example in case of a violation.
- *Body.* We combine different predicates based on the properties to verify using Boolean operators.

5.2 Properties Formalization

Security properties would be expressed as predicates over relation constraints and other predicates. We express the sample properties in FOL. The corresponding CSP formalization is given in Appendix B. Table 2 summarizes the predicates required for expressing the properties. Those predi-

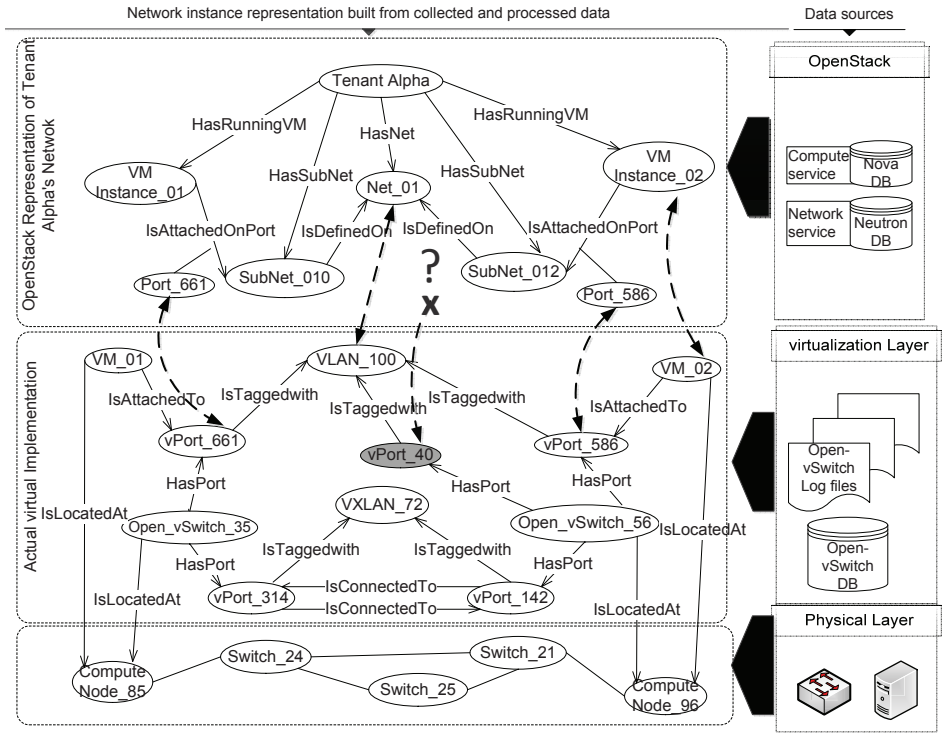


Figure 4: Virtualized infrastructure model instance showing an OpenStack representation and the corresponding actual virtual layer implementation. VXLAN_72 and its ports are part of the infrastructure implementation and do not correspond to any component in tenant Alpha's resources.

Subject	Properties and Sub-Properties		Standards			
			ISO27002 [19]	ISO27017 [20]	NIST800 [23]	CCM [14]
Tenant	Data and proc. location correctness		18.1.1	18.1.1	IR-6, SI-5	SEF-01, IVS-04
	Virt. resource isolation (e.g., No Common Ownership)		-	-	-	STA-5
	Physical isolation (e.g., No Co-residency)		-	13.1.3	SC-2	IVS-8, IVS-9
	Fault tolerance	Facility duplication	17.1, 17.2	12.1.3, 17.1, 17.2	PE-1, PE-13	BCR-03
		Storage service duplication				
		Redundant network connectivity				
Provider	No abuse of resources	Max number of VMs	-	-	-	IVS-11
		Max number of virtual networks				
	No resource exhaustion		-	-	-	IVS-05
Both	Topology consistency	inf. management view/virtual inf.	-	13.1.3	SC-2	IVS-8, IVS-9
		SDN controller view/ virtual inf.				

Table 1: An excerpt of security properties

Relations in Properties	Evaluate to $True$ if
$BelongsTo(r,t)$	The resource r is owned by tenant t
$HasRunningVM(vm,t)$	The tenant t has a running virtual machine vm
$DoesNotTrust(t1,t2)$	Tenant $t2$ is not trusted by tenant $t1$ which means that $t1$'resources should not share the same hardware with $t2$' instances
$IsLocatedAt(vm,cn)$	The instance vm is located at the compute node cn
$IsAssignedPortVLAN(p,v,t)$	the port p is assigned to the VLAN v which is in turn assigned to tenant t
$HasPortVLAN(vs,p,v)$	The port p is created at the virtual switch vs and assigned to VLAN v

Table 2: First Order Logic predicates

cates correspond to CSP relation constraints used to describe the current configuration of the system. Note that predicates that do not appear as relationships in Figure 1 are inferred by correlating other available relations.

No common ownership. We check that a tenant specific virtual resource belongs to a unique tenant.

$$\forall \mathtt{r} \in \mathtt{Resource}, \forall \mathtt{t1}, \mathtt{t2} \in \mathtt{TENANT} \quad (1)$$
$$\mathtt{BelongsTo}(\mathtt{r}, \mathtt{t1}) \wedge \mathtt{BelongsTo}(\mathtt{r}, \mathtt{t2}) \quad \rightarrow (\mathtt{t1} = \mathtt{t2})$$

No co-residence. Based on the collected data, we check that the tenant's instances are not co-located in the same compute node with adversaries' instances.

$$\forall \mathtt{t1}, \mathtt{t2} \in \mathtt{TENANT}, \forall \mathtt{vm1}, \mathtt{vm2} \in \mathtt{INSTANCE}, \quad (2)$$
$$\forall \mathtt{cn1}, \mathtt{cn2} \in \mathtt{COMPUTEN}:$$
$$\mathtt{HasRunningVM}(\mathtt{vm1}, \mathtt{t1}) \wedge \mathtt{HasRunningVM}(\mathtt{vm2}, \mathtt{t2}) \wedge$$
$$\mathtt{DoesNotTrust}(\mathtt{t1}, \mathtt{t2}) \wedge \mathtt{IsLocatedAt}(\mathtt{vm1}, \mathtt{cn1}) \wedge$$
$$\mathtt{IsLocatedAt}(\mathtt{vm2}, \mathtt{cn2}) \rightarrow \mathtt{cn1} \neq \mathtt{cn2}$$

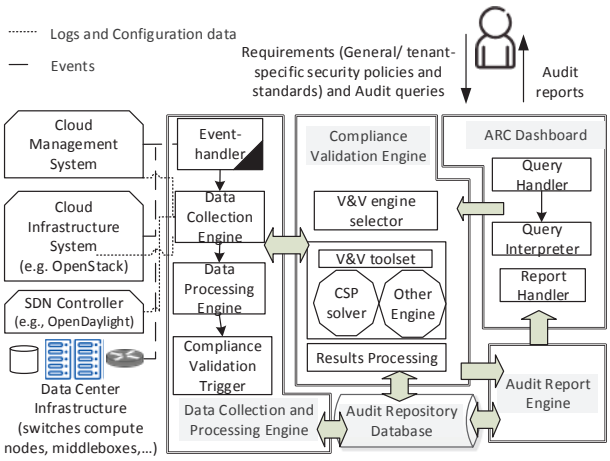


Figure 5: A high-level architecture of our cloud auditing framework

Topology consistency. We check that mappings between virtual resources over different layers are properly maintained and that the current view of the cloud infrastructure management system on the topology, matches the actual topology of the virtual layer. In the following, we consider port consistency as a specific case of topology consistency. We check that the set of virtual ports assigned to a given tenant's VLAN by the provider correspond exactly to the set of ports inferred from data collected from the actual infrastructure's configuration for the same tenant's VLAN.

$$\forall \texttt{vs} \in \texttt{vSWITCH}, \quad \forall \texttt{p} \in \texttt{Port} \quad \forall \texttt{t} \in \texttt{TENANT} \quad \forall \texttt{v} \in \texttt{VLAN} \quad (3)$$
$$\texttt{HasPortVlan}(\texttt{vs}, \texttt{p}, \texttt{v}) \Leftrightarrow \texttt{IsAssignedPortVLAN}(\texttt{p}, \texttt{v}, \texttt{t})$$

EXAMPLE 5. *Listing 1 presented in Appendix B is the CSP code to verify the no common ownership, no co-residence and port consistency properties for our running example. Variables along with their respective domains are first declared (see Listing 1 lines 2-10). Based on the properties of interest, a set of relations are defined and populated with their supporting tuples, where the support is generated from actual data in the cloud (see Listing 1 lines 12-17). Then, the properties are declared as predicates over these relations (see Listing 1 lines 19-24). Finally, the disjunction of the predicates is instantiated for verification (see Listing 1 line 26). As we are formalizing the negation of the properties, we are expecting the UNSAT result, which means that none of the properties holds (i.e., no violation of the properties). We present the verification outputs in Section 6.*

6. APPLICATION TO THE OPENSTACK

This section describes how we integrate our audit and compliance framework into OpenStack. First, we briefly present the OpenStack networking service (Neutron), the compute service (Nova) and Open vSwitch [1], the most popular virtual switch implementation. We then detail our auditing framework implementation and its integration in OpenStack along with the challenges that we faced and overcame.

6.1 Background

OpenStack [27] is an open-source cloud infrastructure management platform that is being used almost in half of private clouds and significant portions of the public clouds (see [15] for detailed statistics). The major components of OpenStack to control large collections of computing, storage and networking resources are respectively Nova, Swift and Neutron along with Keystone. Following is the brief description of Nova and Neutron:

Nova [27] This is the OpenStack project designed to provide massively scalable, on demand, self service access to compute resources. It is considered as the main part of an Infrastructure as a Service model.

Neutron [27] This OpenStack system provides tenants with capabilities to build rich networking topologies through the exposed API, relying on three object abstractions, namely, networks, subnets and routers. When leveraged with the Modular Layer 2 plug-in (ML2), Neutron enables supporting various layer 2 networking technologies. For our testbed we consider Open vSwitch as a network access mechanism and we maintain two types of network segments, namely, VLAN for communication inside of the same compute node, and VXLAN for inter compute nodes communications.

Open vSwitch [1]. Open vSwitch is an open source software switch designed to be used as a vSwitch in virtualized server environments. It forwards traffic between different virtual machines (VMs) on the same physical host and also forwards traffic between VMs and the physical network.

6.2 Integration to OpenStack

We focus mainly on three components in our implementation: the data collection engine, the data processing engine, and the compliance validation engine. The data collection engine involves several components of OpenStack e.g., Nova and Neutron for collecting audit data from databases and log files, different policy files and configuration files from the OpenStack ecosystem, and log files from various virtual networking components such as Open vSwitch to fully capture the configuration. The data is then converted into a consistent format and missing correlation is reconstructed. The results are used to generate the code for the validation engine based on Sugar input language. The compliance validation engine performs the verification of the properties by feeding the generated code to Sugar. Finally, Sugar provides the results on whether the properties hold or not. Figure 6 illustrates the steps of our auditing process. In the following, we describe our implementation details along with the related challenges.

Data collection engine. We present hereafter different sources of data in OpenStack along with the current support for auditing offered by OpenStack and the virtual networking components. The main sources of audit data in OpenStack are logs, configuration files, and databases. Table 3 shows some sample data sources. The involved sources for auditing depend on the objective of the auditing task and the tackled properties. We use three different sources to audit configuration correctness of virtualized infrastructures:

- *OpenStack.* We rely on a collection of OpenStack databases, hosted in a MySQL server, that can be read using component-specific APIs such as Neutron APIs. For instance, in Nova database, table *Compute-node*

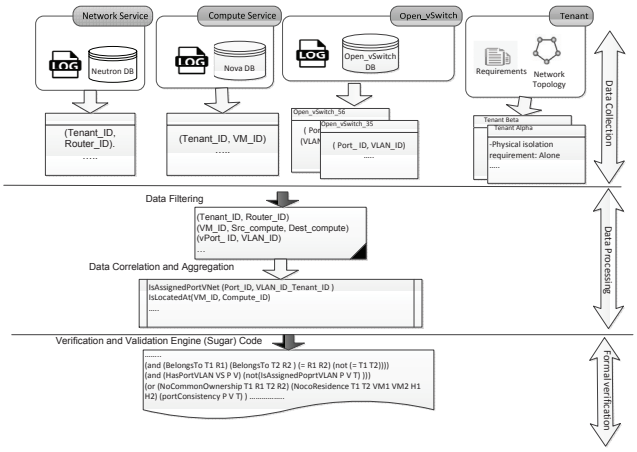


Figure 6: An instance of our OpenStack-based auditing solution with the example of data collection, formatting, correlation building and Sugar source generation

Relations	Sources of Data
$BelongsTo$	Table *Instances* in Nova database and *Routers, Subnets* and *Ports* in Neutron database, Neutron logs
$DoesnotTrust$	The tenant physical isolation requirement input
$IsLocatedAt$	Tables *Instances* in Nova database
$IsAssignedPortVLAN$	*Networks* in Nova database and *Ports* in Neutron database
$HasPortVLAN$	Open vSwitch instances located at various compute nodes
$HasRunningVM$	Table *Instances* in Nova database

Table 3: Sample Data Sources in OpenStack, Open vSwitch and Tenants' requirements

contains information about the hosting machines such as the hypervisor's type and version, table *Instance* contains information about the project (tenant) and the hosting machine, table *Migration* contains migration events' related information such as the source-compute and the destination-compute. The Neutron database includes various information such as security groups and port mappings for different virtualization mechanisms.

- *Open vSwitch.* Flow tables and databases of Open vSwitch instances located in different compute nodes and in the controller node constitute another important source of audit data for checking whether there exist any discrepancies between the actual configuration and the OpenStack view.

- *Tenant policies.* We consider security policies expressed by the customers, such as physical isolation requirements. As expressing tenants' policies is out of the scope of this paper, we assume that they are parsable XML files.

Data processing engine. Our data processing engine, which is implemented in Python, mainly retrieves necessary information from the collected data according to the targeted properties, recovers correlation from various sources, eliminates redundancies, converts it into appropriate formats, and finally generates the source code for Sugar.

- Firstly, for each property, our plug-in identifies the involved relations. The relations' support is either fetched directly from the collected data such as the support of the relation *BelongsTo*, or recovered after correlation, as in the case of the relation *IsAssignedPortVLAN*.

- Secondly, our processing plug-in formats each group of data as an n-tuple, i.e., *(resource, tenant),(port, vlan, tenant)*, etc.

- Finally, our plug-in uses the n-tuples to generate the portions of Sugar's source code, and append the code with the variable declarations, relationships and predicates for each security property (as discussed in Section 5). Different scripts are needed to generate Sugar source code for the verification of different properties.

Compliance Validation. The compliance validation engine is discussed in details in Section 5. In the following example, we discuss how our auditing framework can detect the violation of the no common ownership, no co-residence and port inconsistency security properties caused by the attack scenarios of our running example.

EXAMPLE 6. *In this example, we describe how a violation of no common ownership, no co-residence and port-consistency properties may be caught by auditing.*

Firstly, our program collects data from different tables in the Nova and Neutron databases, and logs from different Open vSwitch instances. Then, the processing engine correlates and converts the collected data and represents it as tuples; for an example: (18038 10) (6100 11000) (512 6020 18033) where Port_84: 18038, Alpha: 10, VM_01: 6100, Open_vSwitch_56: 512, vPort_40: 18033 and VLAN_100: 6020. Additionally, the processing engine interprets each property and generates the associated Sugar source code (see Listing 1 for an excerpt of the code) using processed data and translated properties. Finally, Sugar is used to verify the security properties.

We show for each property how the violation is detected:

a) No common Ownership. The predicate CommonOwnership will evaluate to true if there exists a resource belonging to two different tenants. As Port_84 has been created by Beta, BelongsTo(Port_84, Beta) evaluates to true based on collected data from Neutron logs. Port_84 is defined on Alpha's router, hence, BelongsTo(Port_84, Alpha) evaluates to true based on collected data from Neutron database. Consequently, the predicate CommonOwnership evaluates to true. In this case, the output of sugar (SAT) is the solution of the problem, (r1 = 18038; r2 =18038; t1 =10; T2=11), which is actually the proof that Port_84 violates the no common ownership property.

b) No co-residence. In our example (see Figure 3), the supports HasRunningVM((VM_02, Alpha)(VM_03, Beta)), IsLocatedAt((VM_02, Compute_Node_96)(VM_03,Compute_Node_96) and DoesNotTrust(Alpha, Beta), where VM_02:6101, VM_03:6102, and Compute_Node_96:11100, make the predicate evaluate to true meaning that the no co-residence property has been violated.

c) *Port-consistency. The predicate PortConsistency evaluates to true if there exists a discrepancy between the OpenStack view of the virtualized infrastructure and the actual configuration. The support HasPortVLAN(Open_vSwitch_56, vPort_40, VLAN_100) makes the predicate evaluate to true, as long as there is no tuple such that IsAssignedPortVLAN (Port, VLAN_100, Alpha) where Port maps to vPort_40:18033.*

Challenges. Checking the configuration correctness in virtualized environment requires considering logs generated by virtualization technologies at various levels, and checking that mappings are properly maintained over different layers. Unfortunately, OpenStack does not maintain such overlay details.

At the OpenStack level, ports are directly mapped to VXLAN IDs, whereas at the Open-vSwitch level, ports are mapped to VLAN tags and mappings between the VLAN tags and VXLAN IDs are maintained. To overcome this limit, we devised a script that generates logs from all the Open vSwitch instances. The script recovers mappings between VLAN tags and the VXLAN IDs from the flow tables using the *ovs-ofctl* command line tool. Then, it recovers mappings between ports and VLAN tags from the Open-vSwitch data base using the *ovs-vsctl* command line utility.

Checking the correct configuration of overlay networks requires correlating information collected both from Open vSwitch instances running on top of various compute nodes and the controller node, and data recovered from OpenStack data bases. To this end, we extended our data processing plug-in to deduce correlation between data. For example, we infer the relation (*portvlantenant*) from the available relations (*vlanvxlan*) recovered from Open vSwitch and (*portvxlantenant*) recovered from the Nova and Neutron databases. In our settings, we consider a ratio of 30 ports per tenant, which leads to 300,000 entries in the relation (*portvxlantenant*) for 10,000 tenants. The number of entries is considerably larger than the number of tenants, because a tenant may have several ports and virtual networks. As a consequence, with the increasing number of tenants, the size of this relation grows and complexity of the correlation step also increases proportionally. Note that, correlation is required for several of our listed properties.

An auditing solution becomes less effective if all needed audit evidences are not collected properly. Therefore, to be comprehensive in our data collection process, we firstly check fields of all varieties of log files available in OpenStack, all configuration files and all Nova and Neutron database tables. Through this process, we identify all possible types of data with their sources.

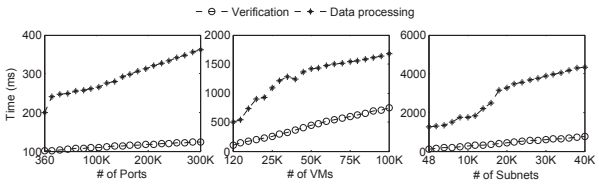
7. EXPERIMENTS

Here we discuss the performance of our work by measuring the execution time, memory, and CPU consumption.

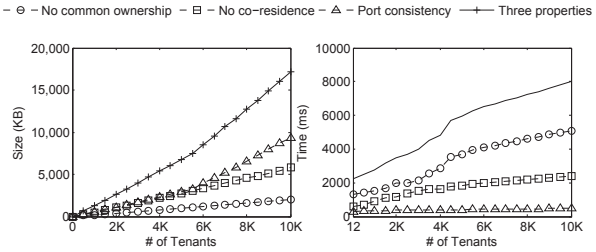
7.1 Experimental setting

We deployed OpenStack with one controller node and three compute nodes, each having Intel i7 dual core CPU and 2GB memory running Ubuntu 14.04 server. Our OpenStack version is DevStack Juno (2014.2.2.dev3). We set up a real test bed environment constituted of 10 tenants, 150 VMs and 17 routers. To stress the verification engine and assess the scalability of our approach as a whole, we furthermore simulated an environment with 10,000 tenants, 100,000 VMs, 40,000 subnets, 20,000 routers and 300,000 ports with a ratio of 10 VMs, 4 subnets, 2 routers and 30 ports per tenant. To comply verification, we use the V&V tool, Sugar V2.2.1 [31]. We conduct the experiment for 20 different audit trail datasets in total.

All data processing and V&V experiments are conducted on a PC with 3.40 GHz Intel Core i7 Quad core CPU and 16 GB memory and we repeat each experiment 1, 000 times.



(a) Time required for data processing and verification for the port consistency (left), no co-residence (middle) and no common ownership (right) by varying number of ports, VMs and subnets respectively.



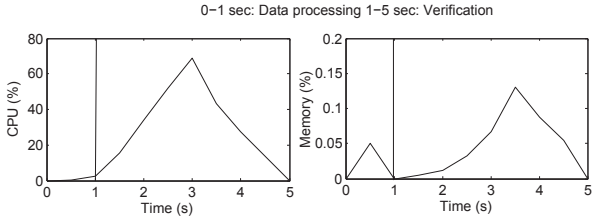
(b) Total size (left) of collected audit data and time required (right) for auditing the port consistency, no co-residence, no common ownership and sequentially auditing three properties (worst case) by varying number of tenants.

Figure 7: Execution time for each auditing step, total size of the collected audit data and total time for different properties using our framework

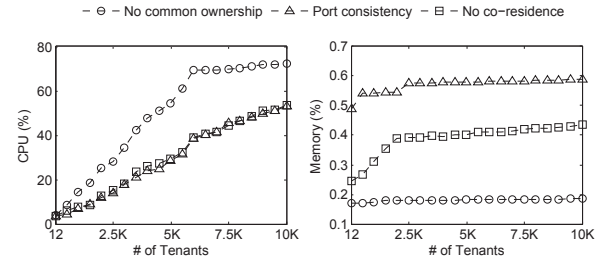
7.2 Results

The first set of our experiment (see Figure 7) demonstrates the time efficiency of our auditing solution. Figure 7(a) illustrates the time in milliseconds required for data processing and compliance verification steps for port consistency, no co-residence and no common-ownership properties. For each of the properties, we vary the most significant parameter (e.g., the number of ports, VMs and subnets for port consistency, no co-residence and no common ownership properties respectively) to assess the scalability of our auditing solution. Figure 7(b) (left) shows the size of the collected data in KB for auditing by varying the number of tenants. The collected data size reaches to around 17MB for our largest dataset. We also measure the time for collecting data as approximately 8 minutes for a fairly large cloud setup (10,000 tenants, 100,000 VMs, 300,000 ports, etc.). Note that data collection time heavily depends on deployment options and complexity of the setup. Moreover, the initial data collection step is performed only once for the auditing process (later on incremental collection will be performed at regular intervals), so the time may be considered reasonable. Figure 7(b) (right) shows the total execution time required for

each property individually and in total. Auditing no common ownership property requires the longest time, because of the highest number of predicates used in the verification step; however, it finishes in less than 4 seconds. In total, the auditing of three properties completes within 8 seconds for the largest dataset, when properties are audited sequentially. However, since there is no interdependency between verifying different security properties, we can easily run parallel verification executions. The parallel execution of the verification step for different properties reduces the execution time to 4 seconds, the maximum verification time required among three security properties. Additionally, we can infer that the execution time is not a linear function of the number of security properties to be verified. Indeed, auditing more security properties would not lead to a significant increase in the execution time.



(a) Peak CPU and memory usage for each step of our auditing solution over time when there are 10,000 tenants, 40,000 subnets, 100,000 VMs and 300,000 ports.



(b) CPU (left) and memory (right) usage for each step of our auditing solution over time when there are 6,000 tenants, 24,000 subnets, 60,000 VMs and 180,000 ports.

Figure 8: CPU and memory usage for each step and for different properties of our auditing solution over time

The objective of our second experiment (Figures 8(a)(left) and 8(b)(left)) is to measure the CPU usage (in %). In Figure 8(a)(left), we measure the peak CPU usage consumed by data processing and verification steps while auditing the no common ownership property. We notice that the average CPU usage is around 35% for the verification, whereas it is fairly negligible for the data processing step. According to Figure 8(b)(left), the CPU usage grows almost linearly with the number of tenants. However, the speed of increase varies depending on the property. It reaches a peak of over 70% for the no common ownership property for 10,000 tenants. This is due to the huge amount of tenant specific resources (e.g. for 10,000 tenants the number of the resources involved may reach $216,000$).

Note that, we conduct our experiments in a single PC; if the security properties can be verified through concurrent and independent Sugar executions, we can easily parallelize this task by running several instances of Sugar on different VMs in the cloud environment. Thus the parallelization in the cloud allows to reduce the overall verification time to the maximum time for any individual security property.

Our final experiment (Figures 8(a)(right) and 8(b)(right)) demonstrates the memory usage of our auditing solution. Figure 8(a)(right) shows that data processing step has a minor memory usage (with a peak of 0.05%), whereas the highest memory usage observed for the verification step for our largest setup is less than 0.19% of 16GB memory. The Figure 8(b) (right) shows that the port consistency property has the lowest memory usage with a percentage of 0.2% whereas no common ownership has the highest memory usage, which is less than 0.6% for 10,000 tenants. Our observation from this experiment is that memory usage is related to the number of relations, variables and constraints involved to verify each property.

Discussion. In our experiments, we audited several security properties e.g., no common ownership and port consistency, for up to $10,000$ tenants with a large set of various resources ($300,000$ ports, $100,000$ VMs, $40,000$ subnets) in less than 8 seconds. The auditing activity occurs upon request from the auditor (or in regular intervals when the auditor sets regular audits). Therefore, we consider the costs of our approach to be reasonable even for large data centers. Although we report results for a limited set of security properties related to virtualized cloud infrastructure, promising results show the potentiality of the use of formal methods for auditing. Particularly, we show that the time required for our auditing solution grows very slowly with the number of security properties. As seen in Fig 7a, we anticipate that auditing a large list of security properties in practice would still be realistic. The cost generally increases almost linearly with the number of tenants.

8. CONCLUSION

In this paper, we elaborated a generic model for virtualized infrastructures in the cloud. We identified a set of relevant structural security properties to audit and mapped them to different standards. Then, we presented a formal approach for auditing cloud virtualized infrastructures from the structural point of view. Particularly, we showed that our approach is able to detect topology inconsistencies that may occur between multiple control layers in the cloud. Our evaluation results show that formal methods can be successfully applied for large data centers with a reasonable overhead. As future directions, we intend to leverage our auditing framework for continuous compliance checking. This will be achieved by monitoring various events, and triggering the verification process whenever a security property is affected by the changes. A second area of investigation is to extend the list of security properties with the operational properties. This would allow to check the compliance of the forwarding network functionality with the access control lists and routing policies.

9. ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable comments. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under CRD Grant N01566.

10. REFERENCES

[1] Open vswitch. Available at: http://openvswitch.org/.

[2] Policy as a service ("congress"). Available at: http://wiki.openstack.org/wiki/Congress.

[3] Federal data protection act. http://www.gesetze-im-internet.de/englisch_bdsg, August 2009.

[4] IBM Corporation. Ibm point of view: Security and cloud computing, 2009.

[5] C. S. Alliance. Security guidance for critical areas of focus in cloud computing v 3.0, 2011.

[6] C. S. Alliance. The notorious nine cloud computing top threats in 2013, February 2013.

[7] M. Bellare and B. Yee. Forward integrity for secure audit logs. Technical report, Citeseer, 1997.

[8] M. Ben-Ari. *Mathematical logic for computer science*. Springer Science & Business Media, 2012.

[9] S. Bleikertz. Automated security analysis of infrastructure clouds. Master's thesis, Technical University of Denmark and Norwegian University of Science and Technology, 2010.

[10] S. Bleikertz, T. Groß, and S. Mödersheim. Automated verification of virtualized infrastructures. In *Proceedings of CCSW*, pages 47–58. ACM, 2011.

[11] S. Bleikertz, C. Vogel, and T. Groß. Cloud radar: near real-time detection of security failures in dynamic virtualized infrastructures. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 26–35. ACM, 2014.

[12] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy. Self-service cloud computing. CCS '12, pages 253–264, New York, NY, USA, 2012. ACM.

[13] Cloud Security Alliance. Top ten big data security and privacy challenges, 2012.

[14] Cloud Security Alliance. Cloud control matrix CCM v3.0.1, 2014. Available at: https://cloudsecurityalliance.org/research/ccm/.

[15] datacenterknowledge.com. Survey: One-third of cloud users' clouds are private, heavily OpenStack, 2015. Available at: http://www.datacenterknowledge.com.

[16] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann. Sphinx: Detecting security attacks in software-defined networks. In *NDSS Symposium*, 2015.

[17] F. Doelitzscher, C. Reich, M. Knahl, A. Passfall, and N. Clarke. An agent based business aware incident detection system for cloud environments. *Journal of Cloud Computing*, 1(1), 2012.

[18] F. H.-U. Doelitzscher. *Security Audit Compliance For Cloud Computing*. PhD thesis, Plymouth University, February 2014.

[19] ISO Std IEC. ISO 27002:2005. *Information Technology-Security Techniques*, 2005.

[20] ISO Std IEC. ISO 27017. *Information technology-Security techniques (DRAFT)*, 2012.

[21] S. Majumdar, T. Madi, Y. Wang, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi. Security compliance auditing of identity and access management in the cloud: Application to openstack. In *IEEE CloudCom*, 2015.

[22] T. E. Network, , and I. S. Agency. Cloud computing benefits, risks and recommendations for information security, December 2012.

[23] NIST, SP. NIST SP 800-53. *Recommended Security Controls for Federal Information Systems*, pages 800–53, 2003.

[24] Opendaylight. The OpenDaylight platform, 2015. Available at: https://www.opendaylight.org/.

[25] OpenStack. Ossa-2014-008: Routers can be cross plugged by other tenants. Available at: https://security.openstack.org/ossa/OSSA-2014-008.html.

[26] OpenStack. Nova network configuration allows guest vms to connect to host services, 2015. Available at: https://wiki.openstack.org/wiki/OSSN/OSSN-0018.

[27] OpenStack. OpenStack open source cloud computing software, 2015. Available at: http://www.openstack.org.

[28] D. Perez-Botero, J. Szefer, and R. B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. Cloud Computing '13, pages 3–10, New York, NY, USA, 2013. ACM.

[29] T. Probst, E. Alata, M. Kaâniche, and V. Nicomette. An approach for the automated analysis of network access controls in cloud computing infrastructures. In *Network and System Security*, pages 1–14. Springer, 2014.

[30] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[31] N. Tamura and M. Banbara. Sugar: A CSP to SAT translator based on order encoding. *Proceedings of the Second International CSP Solver Competition*, pages 65–69, 2008.

[32] TechNet. Nova network configuration allows guest vms to connect to host services cloud services foundation reference architecture - reference model, 2013.

[33] Y. Xu, Y. Liu, R. Singh, and S. Tao. Identifying sdn state inconsistency in openstack. SOSR '15, pages 11:1–11:7, New York, NY, USA, 2015. ACM.

[34] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *(NSDI 14). Seattle, WA: USENIX Association*, pages 87–99, 2014.

[35] S. Zhang and S. Malik. Sat based verification of network data planes. In D. Van Hung and M. Ogawa, editors, *Automated Technology for Verification and Analysis*, volume 8172 of *Lecture Notes in Computer Science*, pages 496–505. Springer International Publishing, 2013.

[36] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. SP '11, pages 313–328, Washington, DC, USA, 2011. IEEE Computer Society.

APPENDIX

A. SECURITY PROPERTIES

In the following, we provide a brief description of the most pertinent properties presented in Table 1.

Data and processing location correctness. One of the main cloud specific security issues is the increased complexity of compliance with laws and regulations [18]. The cloud provider might have data centers spread over different continents and governed by various court jurisdictions. Data and processing can be moved between the cloud provider's data centers without tenants awareness, and fall under conflicting privacy protection laws. The Germany's data protection act [3] stipulates that personal data can only be transferred into countries with the same adequate level of privacy protection laws.

Virtual resources isolation (No common ownership). Resource sharing technology was not designed to offer strong isolation properties for a multi-tenant architecture and thus has been ranked by the CSA among the nine notorious threats related to the cloud [6]. The related risks include the failure of mechanisms separating virtual resources between different tenants of the shared infrastructure, which may lead to situations where one tenant has access to another tenant's resources or data.

No co-residency. Cloud providers consolidate virtual machines, possibly belonging to competing customers, to be run on the same physical machine, which may cause major security concerns as described in [36]. According to the CSA standard [5], customers should be able to express their willingness to share or not the physical hosts with other tenants, and they should be provided with the related evidences.

Redundancy and fault tolerance. Cloud providers have to apply several measures to achieve varying degrees of resiliency following the criticality of the customer's applications. Duplicating facilities in various locations, and replicating storage services are examples of the measures that could be undertaken. Considering additional redundancy of network connectivity and information processing facilities has been mentioned in ISO 27002:2013 [19] as one of best practices.

No abuse of resources. Cloud services can be used by legitimate anonymous customers as a basis to illegitimately lead criminal and suspicious activities. For example, cloud services can be used to stage DDoS attacks [6].

No resource exhaustion. The ease with which virtual resources can be provisioned in the cloud introduces the risk of resource exhaustion [22]. For example, creating a huge amount of VMs within a short time frame drastically increases the odds of misconfiguration which opens up several security breaches [5].

Topology consistency. As stated in [4], it is critical to maintain consistency among cloud layers. The architectural model of the cloud can be described as a stack of layered services: Physical layer, system resources layer, virtualized resources layer, support services layer, and at the top cloud-delivered services. The presence of inconsistencies between these layers may lead to security breaches, which in turn makes the security controls at higher layers inefficient.

B. CONSTRAINTS AND CODE

The CSP constraint to no common ownership property:

$$(\texttt{and}\ \ \texttt{BelongsTo}(\texttt{r1},\texttt{t1})\ \ \texttt{BelongsTo}(\texttt{r2},\texttt{t2})\ \ (\texttt{r1}=\texttt{r2})(\texttt{not}(\texttt{t1}=\texttt{t2})) \tag{4}$$

The CSP constraint for this no co-residence property:

$$(\texttt{and}\ \ \texttt{DoesNotTrust}(\texttt{t1},\texttt{t2})\ \ \texttt{HasRunningVM}(\texttt{vm1},\texttt{t1})\ \ \texttt{HasRunning}(\texttt{vm2},\texttt{t2})\ \ \texttt{IsLocatedAt}(\texttt{vm1},\texttt{h1})\ \ \texttt{IsLocatedAt}(\texttt{vm2},\texttt{h2})\ \ (\texttt{h1}=\texttt{h2})) \tag{5}$$

The CSP constraint corresponding to topology consistency:

$$(\texttt{or}(\texttt{and}\ \ \texttt{HasPortVLAN}(\texttt{vs},\texttt{p},\texttt{v})\ \ (\texttt{not}\ \ \texttt{IsAssignedPortVLAN}(\texttt{p},\texttt{v},\texttt{t}))\ \ (\texttt{and}\ \ \texttt{IsAssignedPortVLAN}(\texttt{p},\texttt{v},\texttt{t})\ \ (\texttt{not}\ \ \texttt{HasPortVLAN}(\texttt{vs},\texttt{p},\texttt{v}))) \tag{6}$$

Listing 1: Sugar source code for common ownership, co-residence and port consistency verification

```
1  //Declaration
2  (domain TENANT 0 60000) (domain RESOURCE 0
       216000)
3  (domain INSTANCE 0 100000) (domain HOST 0 1000)
4  (domain PORT 0 300; 000) (domain VLAN 0 60000)
5  (domain VSWITCH 0 1000)
6  (int T1 TENANT) (int T2 TENANT)
7  (int R1 Resource) (int R2 Resource)
8  (int VM1 INSTANCE) (int VM2 INSTANCE)
9  (int H1 HOST) (int H2 HOST)(int V VLAN)
10 (int T TENANT) (int P PORT) (int vs VSWITCH)
11 //Relations Declarations and Audit data as their support
12 (relation BelongsTo 2 (supports (18037 10)(18038 10) (
       18039 10)(18040 10)(18038 11)(18042 11)(18043
       11)(18044 11)(18045 11)(18046 12)(18047 12)))
13 (relation HasRunningVM 2 (supports (6100 10)(6101
       10)(6102 11)(6103 11)(6104 11)(6105 11)))
14 (relation IsLocatedAt 2 (supports(((6089 11000)(6090
       11000)(6093 11000)(6101 11100)(6102 11100))
15 (relation DoesNotTrust 2 (supports(9 11)(9 13)(9 14)))
16 (relation IsAssignedPortVLAN 3 (supports (18028 6017
       9)(18029 6018 9)(18030 6019 10)(18031 6019
       10)(18032 6020 10) ))
17 (relation HasPortVLAN 3 (supports(511 18030 6019)(511
       18031 6019 10)(512 18032 6020)(512 18033 6021)))
18 //Security properties expressed in terms of predicates
       over relation constraints
19 (predicate (CommonOwnership T1 R1 T2 R2)
20 (and (BelongsTo T1 R1) (BelongsTo T2 R2 ) (= R1 R2)
       (not (= T1 T2))))
21 (predicate (coResidence T1 T2 VM1 VM2 H1 H2) (and
       (DoesNotTrust T1 T2) (HasRunningVM VM1 T1)
22 (HasRunningVM VM2 T2) (IsLocatedAt H1 VM1)
       (IsLocatedAt H2 VM2) (=H1 H2)))
23 (predicate (portConsistency P V T)(or (and
       (IsAssignedPoprtVLAN P V T)(not(HasPortVLAN
       VS P V)))
24 (and (HasPortVLAN VS P V)
       (not(IsAssignedPoprtVLAN P V T)))))
25 \\The Body
26 (or (CommonOwnership T1 R1 T2 R2) (coResidence T1
       T2 VM1 VM2 H1 H2) (portConsistency P V T) )
```