Security Compliance Auditing of Identity and Access Management in the Cloud: Application to OpenStack

Suryadipta Majumdar*, Taous Madi*, Yushun Wang*, Yosr Jarraya†,
Makan Pourzandi†, Lingyu Wang* and Mourad Debbabi*
*Concordia Institute for Information Systems Engineering, Concordia University, Montreal, QC, Canada
Email: {su_majum,t_madi,yus_wang,wang,debbabi}@encs.concordia.ca
†Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada
Email: {yosr.jarraya,makan.pourzandi}@ericsson.com

***Abstract*—Cloud computing has seen a lot of interests and adoption lately. Nonetheless, the widespread adoption of cloud is still being hindered by the lack of transparency and accountability, which has traditionally been ensured through security compliance auditing techniques. Auditing in cloud, however, presents many new challenges in data collection and processing (e.g., data format inconsistency and lack of correlation due to the heterogeneity of cloud infrastructures) and in verification (e.g., prohibitive performance overhead due to the sheer scale of cloud infrastructures and their self-provisioning, elastic, and dynamic nature). In this paper, we propose a security compliance auditing framework for cloud, with special focus on identity and access management, and we implement and evaluate the framework based on OpenStack, one of the most popular cloud management systems. Our experimental results show that auditing with formal methods in large cloud environment is realistic (e.g., our auditing solution can handle 60 thousand users in less than one minute).**

I. Introduction

While cloud computing has seen increasing interests and adoption lately, the fear of losing control and governance still persists due to the lack of transparency and trust [1]. Security auditing and compliance validation may increase cloud tenants' trust in the service providers by providing assurance on the compliance with the applicable laws, regulations, policies, and standards. However, there are currently many challenges in the area of cloud auditing and compliance validation. There exists a significant gap between the high-level recommendations provided in most cloud-specific standards (e.g., Cloud Control Matrix (CCM) [2] and ISO 27017 [3]) and the low-level logging information currently available in existing cloud infrastructures (e.g., OpenStack [4]). Furthermore, the unique characteristics of cloud computing may introduce additional complexity to the task, e.g., the use of heterogeneous solutions for deploying cloud systems may complicate data collection and processing and the sheer scale of cloud, together with its self-provisioning, elastic, and dynamic nature, may render the overhead of many verification techniques prohibitive. In practice, limited forms of auditing may be performed by cloud subscriber administrators [5], and there does not exist any automated compliance tool to the best of our knowledge.

One of the key security aspects to put under the scope of auditing and compliance is *identity and access management* [2]. It is essential for the cloud management system to enable auditing of all activities related to the authentication of user identities and the management and control of user accesses and actions inside a cloud. Although there already exist various efforts on cloud auditing (a detailed review of related works will be given in Section II), to the best of our knowledge, none has facilitated automated auditing of identity and access management. On the other hand, existing approaches to the verification of traditional access control policies are generally insufficient for auditing the cloud since they cannot directly deal with multi-domain cloud environments (see [6], [7]).

Motivating example. Here we provide a sketch of the gap between high-level standards and low-level input data.

- Section 13.2.1 of ISO 27017 [3], which provides security guidelines for the use of cloud computing, recommends *"checking that the user has authorization from the owner of the information system or service for the use of the information system or service..."*.
- The corresponding logging information is available in OpenStack [4] from at least three different sources:
  - Logs of user events (e.g., `router.create.end 1c73637 94305b c7e62 2899` meaning user 1c73637 from domain 94305b is creating a router).
  - Authorization policy files (e.g., `"create_router": "rule:regular_user"` meaning a user needs to be a regular user to create a router).
  - Database record (e.g., `1c73637 Member` meaning user 1c73637 holds the *Member* role).

Clearly, to audit the security compliance, an automated system must locate and collect all such relevant information from different sources inside a cloud, convert the collected data into a consistent format, while reconstructing any missing correlation between the data, before it can feed the data into a verification tool to check against the security property suggested in the high level standard. In this specific case, no automated tool exists yet in OpenStack for those purposes.

Objective and Contributions. In this paper, we propose a cloud auditing framework, in which we focus on identity and access management in a multi-domain cloud environment. We compile a set of security properties from both the existing literature on authorization and authentication and common cloud security standards. We rely on formal methods to enable automated reasoning and allow providing formal proofs or counterexamples of compliance. We implement and integrate the proposed auditing framework into OpenStack, and report real-life experiences and challenges. Our experimental results confirm the scalability and efficiency of our approach. The main contributions of this paper are as follows:

- To the best of our knowledge, this is the first effort on auditing identity and access management in a multi-domain cloud environment based on formal methods.
- The list of security properties provides a bridge between the cloud security standards and the literature on multi-domain access control and token-based authentication.
- Our prototype system may potentially become an integral part of OpenStack providing a practical auditing solution.
- The experimental results show that security auditing using formal methods in large scale cloud environments is realistic, which we believe may generate more interest in applying formal methods to cloud.

The remainder of this paper is organized as follows. Section II reviews the related work. Section III describes our methodologies. Section IV presents the necessary formalization for the access control model and security properties. Section V details the integration of our auditing framework into OpenStack. Section VI experimentally evaluates the performance of our approach. Finally, we conclude our paper discussing future directions in Section VII.

II. Related work

Several existing efforts consider the verification of access control policies at the design time expressed in the standard eXtensible Access Control Markup Language (XACML) using formal reasoning. Among them, Fisler et al. [8] propose Binary Decision Diagrams (BDD) and custom algorithms to verify access-control policies. Ahn et al. [9] use answer set programming (ASP) and leverage existing ASP reasoning models to conduct policy verification. Arkoudas et al. [10] propose a Satisfiability Modulo Theory (SMT) policy analysis framework. In most of those work, multi-domains access control models are not considered.

To accommodate the need of secure collaborative environments such as cloud computing, there have been some efforts towards proposing multi-domain/multi-tenant access control models (e.g., [11], [6], [7]). Gouglidis and Mavridis [7] leverage graph theory algorithms to verify a subset of the access control security properties. Gouglidis et al. [12] utilize model-checking to verify custom extensions of RBAC with multi-domains [7] against security properties. Lu et al. [13] use set theory to formalize policy conflicts in the context of interoperation in the multi-domain environment. However, auditing encompasses more than a verification approach. In contrast to these works, we are dealing with the verification of not only the policies but also their implementations, which involve efficient techniques to collect, process, and verify large amount of data.

In the context of cloud auditing, there are several work that target auditing data location and storage in the cloud (e.g., [14]) and others target infrastructure change auditing (e.g., [15]). Particularly, Ullah et al. [16] propose an architecture to build automated security compliance tool for cloud computing platforms focusing on auditing clock synchronization and remote administrative & diagnostic port protection. Doelitzscher [17] proposes on-demand audit architecture for IaaS clouds and an implementation based on software agents to enable anomaly detection system to identify anomalies in IaaS clouds for the purpose of auditing. The works in [16], [17] have the same general objective, which is cloud auditing, as ours, but they use empirical techniques to perform auditing whereas we use formal techniques to model and solve the auditing problem. Tang et al. [6] formalize the core OpenStack access control (OSAC) and propose a domain trust extension for OSAC to facilitate secure cross-domain authorization. We adapt this model in our work. To the best of our knowledge, none of the aforementioned works support auditing multi-domain identity and access management in the cloud.

Several industrial efforts include solutions to support cloud auditing in specific cloud environments. For instance, Microsoft proposes SecGuru [18] to audit Azure datacenter network policy using the SMT solver Z3. IBM also provides a set of monitoring tool integrated with QRadar [19], which is their security information and event management system, to collect and analyze events in the cloud. Amazon is offering web API logs and metric data to their AWS clients by AWS CloudWatch & CloudTrail [20] that could be used for the auditing purpose. However, none of them apprently audits identity and access management compliance based on cloud standards.

III. Methodology

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

A. Preliminaries

In the following, we describe the multi-domain RBAC model [6] and threat model, followed by a list of security properties for auditing.

Access control model. We focus on auditing multi-domain role-based access control (RBAC), which is known to be scalable and suitable for cloud [21] and being adopted in real world platforms (e.g., OpenStack [4] and Microsoft Azure [22]). In particular, we assume the extended RBAC model as proposed in [6], which adds support for multi-tenancy in the cloud. Beyond the basic components of the standard RBAC model, e.g., users and roles, whose definitions can be found in [6], we only review here the additional entities specific to the cloud.

- Tenant[1]: A tenant is a collection of users who share a common access with specific privileges to the cloud instances.
- Domain: A domain is a collection of tenants, which draws an administrative boundary within a cloud.
- Token: Token is a package of necessary information used to authenticate prior to avail any operation.
- Group: Groups are formed for better user management.
- Service: A service means a distributed cloud service.
- Object and Operation: An object represents a cloud resource e.g., VM. An operation is an access method to an object. Object and operation together represent permissions.

Example 1 *Figure 1 depicts our running example, which is an instance of the access control model presented in [6]. In this scenario, Alice and Bob are the admins of domains, Da and Db, respectively, with no collaboration (trust) between the two domains; Pa and Pb are two projects respectively owned by the two domains. To show what may go wrong in such a*

[1] We interchangeably use the terms, tenant and project, described in Figure 1

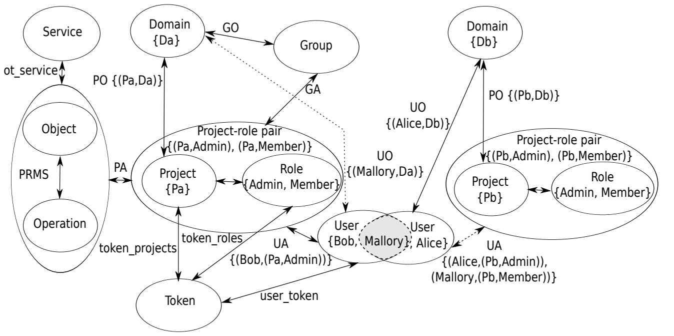


Fig. 1. Two example instances of the access control model of [6] depicting the resultant state of the access control system after the exploit of the vulnerability, OSSN-0010. The shaded region and dotted arrows show an instance of the exploit described in Example 1.

scenario, we consider a real world vulnerability, OSSN-0010[2]*, found in OpenStack, which allows a tenant admin to become a cloud admin and acquire privileges to bypass the boundary protection between tenants, and illicitly utilize resources from other tenants while evading the billing. Suppose Bob has exploited this vulnerability to become a cloud admin. Figure 1 depicts the resultant state of the access control system after this attack. As a result, Mallory belonging to domain Da is assigned a role (Pb,Member), which is from domain Db. This violates the security requirement of these domains as they do not trust each other.*

Threat model. We assume the cloud system may have implementation flaws, mis-configurations and vulnerabilities that can be potentially exploited by malicious entities to violate security properties specified by the cloud tenants. We also assume any cloud users, including cloud operators, except those who initiate the auditing process, may be malicious. However, we assume the cloud infrastructure management systems may be trusted for the integrity of the audit input data (e.g., logs, configurations, etc.) collected through API calls, and 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 the auditing components, e.g., [23]). Although our auditing framework may catch violations of specified security properties due to either misconfigurations or exploits of vulnerabilities, our focus is not to detect specific attacks or intrusions.

Cloud auditing properties. As a major goal of this work is to establish a bridge between high-level guidelines provided in 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 I presents an excerpt of the list of auditing properties that we identify from the access control literature, relevant standards (e.g., ISO 27002, CCM) and the real-world cloud implementation (e.g., OpenStack). The list covers relevant sections related to identity and access management in several common standards, namely ISO 27002 [24], NIST SP 800-53 [25], CCM [2] and ISO 27017 [3], in which ISO 27002 and NIST SP 800-53 provide recommendations for the information security management, and CCM and ISO 27017 recommend security controls specific to cloud. For our running example, we will focus on following two properties. *Common ownership:* based on the challenges of multi-domain cloud discussed in [12], [6], users must not hold any role from another domain. *Minimum exposure:* each domain in a cloud must limit the exposure of its information to other domains [6].

Example 2 *The exploitation of the vulnerability mentioned in Example 1 violates the* common ownership *property. According to the property, Mallory must not hold a role member in project Pb belonging to domain Db, because Mallory belongs to domain Da and there exists no collaboration between domains Da and Db. We will show how we can formalize and verify this property in the coming sections.*

Properties	Standards			
	ISO27002 [24]	ISO27017 [3]	NIST800 [25]	CCM [2]
Role activation [26]	*11.2.2.b*	*13.2.2b*	*AC-1*	*IAM-09*
Permitted action [26]	*11.2.1.b, 1.2.2c*	*13.2.1b, 13.2.2c*	*AC-14*	*IAM-10*
Minimum exposure [6]	*11.6.1*	*13.4.1*	*AC-4*	*IAM-04,06*
Separation of duties [6]	*11.6.2*	*13.6.2*	*AC-5*	*IAM-02,05*
Common ownership [12]	*11*	*13*	*AC*	*IAM*
Privilege escalation [12]	*11.2.2.b*	*13.2.2b*	*AC-6*	*IAM-08*
Cardinality [26]	*11.2.4*	*13.2.4*	*AC-1*	
Cyclic inheritance [12]				
User-access validation [3]	*11.5.2*	*13.4*	*AC-3*	*IAM-10*
User-access revocation [24]	*11.2.1h*	*13.2.1h*	*AC-2*	*IAM-11*
No duplicate ID [2]	*11.5.2*	*13.5.2*	*AC-2*	*IAM-12*
Remote access [3]	*11.4.2*	*13.4.2*	*AC-17*	*IAM-02,07*
Secure log-on [3]	*11.5.1*	*13.4.2*	*AC-7,9*	*IAM-02*
Session time-out [25]	*11.5.5*	*13.2.8*	*AC-12*	
Concurrent session [25]		*13.5.4*	*AC-10*	

TABLE I. An Excerpt of our Security Properties

B. Audit ready cloud framework

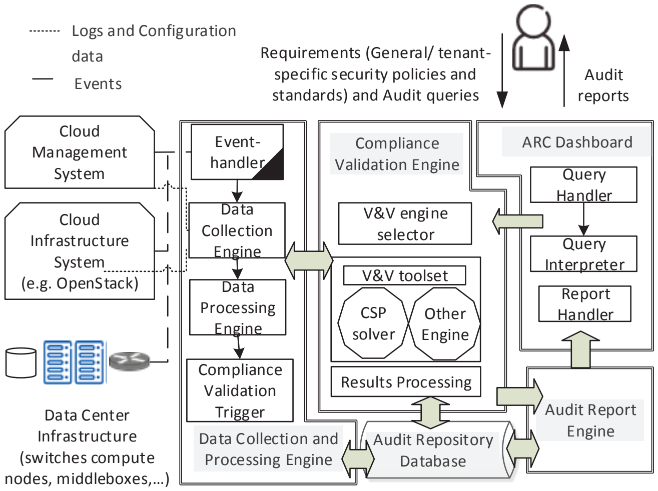

Logs and Configuration data
Events
Requirements (General/ tenant-specific security policies and standards) and Audit queries
Audit reports
Cloud Management System
Cloud Infrastructure System (e.g. OpenStack)
Data Center Infrastructure (switches compute nodes, middleboxes,...)
Event-handler
Data Collection Engine
Data Processing Engine
Compliance Validation Trigger
Data Collection and Processing Engine
Compliance Validation Engine
V&V engine selector
V&V toolset
CSP solver
Other Engine
Results Processing
Audit Repository Database
ARC Dashboard
Query Handler
Query Interpreter
Report Handler
Audit Report Engine

Fig. 2. A high-level architecture of our auditing framework

Figure 2 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, ARC 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

[2] Sample Keystone v3 policy exposes privilege escalation vulnerability, available at: https://wiki.openstack.org/wiki/OSSN/OSSN-0010

the audit result in a report. 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 main components in the following.

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 (e.g., files and databases). The processing engine must pre-process the data in order to provide specific information needed to verify given properties. A 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, evidences can be obtained from the output of the verification back-end, e.g., a valid assignment of the variables that does not satisfy all constraints from which a mapping to real data in the cloud can be inferred to provide a meaningful feedback. 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 (e.g., SAT-based engines) can serve our needs. This may depend on the property and/or the model being verified.

IV. Formalization for Auditing Identity and Access Management

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 [27], 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.

A. Model Formalization

Referring to Figure 1, entities are encoded as CSP variables with their domains definitions (over integer), where instances are values within the corresponding domain. For example, $User$ is defined as a finite domain ranging over integer such that (domain $User$ 0 max_user) is a declaration of a domain of users, where the values are between 0 and max_user. Relationships and their instances are encoded as relation constraints and their supports, respectively. For example, $AuthorizedR$ is encoded as a relation, with a support as follows: (relation $AuthorizedR$ 3 (supports (r1,u1,t1) (r2,u2,t2))). The support of this relation (e.g., (r1,u1,t1)) 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, u and op are entities (or variables) defined respectively over the domains $User$ and $Operation$, 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.

B. Properties Formalization

Security properties would be expressed as predicates over relation constraints and other predicates. We select two representative properties to detail in this paper: common ownership and minimum exposure. We first express these properties in first order logic [28] and then present their CSP formalization (using Lisp-like Sugar syntax). Table II summarizes the relations that we use in these properties and their correspondent relations in Fig 1.

Common ownership. Users are authorized for the roles that are only defined within their domains.

$$\begin{gathered}\forall \mathtt{u} \in \mathtt{User}, \forall \mathtt{d} \in \mathtt{Domain}, \forall \mathtt{r} \in \mathtt{Role}, \forall \mathtt{t} \in \mathtt{Tenant} \\ \mathtt{BelongsToD}(\mathtt{u},\mathtt{d}) \wedge \mathtt{AuthorizedR}(\mathtt{u},\mathtt{t},\mathtt{r}) \longrightarrow \\ \mathtt{TenantRoleDom}(\mathtt{t},\mathtt{r},\mathtt{d})\end{gathered} \tag{1}$$

The corresponding CSP constraint is

$$\begin{gathered}(\mathtt{and}\quad \mathtt{BelongsToD}(\mathtt{u},\mathtt{d})\mathtt{AuthorizedR}(\mathtt{u},\mathtt{t},\mathtt{r}) \\ (\mathtt{not}\quad \mathtt{TenantRoleDom}(\mathtt{t},\mathtt{r},\mathtt{d})))\end{gathered} \tag{2}$$

Minimum exposure. We assume that the user access is revoked properly and that each domain's administrator may share a set of objects (resources) with other domains. The administrator defines accordingly a policy governing the shared objects, the allowed domains for a given object and the allowed actions for a given domain with respect to a specific object. During data processing, we recover for each domain, the set of foreign objects (belonging to other domains) and the actual operations performed on those objects (from the logs). This property allows checking whether the collected and correlated data complies with the defined policy of each domain.

$$\begin{gathered}\forall \mathtt{d}, \mathtt{od} \in \mathtt{Domain}, \forall \mathtt{o} \in \mathtt{Object}, \forall \mathtt{op} \in \mathtt{Operation}, \\ \forall \mathtt{r} \in \mathtt{Role}, \forall \mathtt{t} \in \mathtt{Tenant}, \forall \mathtt{u} \in \mathtt{User} \\ \mathtt{LogEntry}(\mathtt{d},\mathtt{t},\mathtt{u},\mathtt{r},\mathtt{o},\mathtt{op}) \wedge \mathtt{BelongsTo}(\mathtt{u},\mathtt{d}) \wedge \\ \mathtt{OwnerD}(\mathtt{od},\mathtt{t},\mathtt{o}) \longrightarrow \mathtt{AuthorizedOp}(\mathtt{d},\mathtt{t},\mathtt{u},\mathtt{r},\mathtt{o},\mathtt{op}))\end{gathered} \tag{3}$$

The CSP constraint for this property is:

$$\begin{gathered}(\mathtt{and}(\mathtt{and}\quad \mathtt{LogEntry}(\mathtt{d},\mathtt{t},\mathtt{u},\mathtt{r},\mathtt{o},\mathtt{op}) \\ \mathtt{OwnerD}(\mathtt{od},\mathtt{t},\mathtt{o})\mathtt{BelongsTo}(\mathtt{u},\mathtt{d})) \\ (\mathtt{not}\quad (\mathtt{AuthorizedOp}(\mathtt{d},\mathtt{t},\mathtt{u},\mathtt{r},\mathtt{o},\mathtt{op}))))\end{gathered} \tag{4}$$

Relations in Properties	Evaluate to $True$ if	Corresponding Relations in Fig 1
$AuthorizedOp(d, t, u, r, o, op)$	In domain d, and tenant t, the user u, with the role r is authorized to perform operation op on object o	UA, PO, Project-role pair, PA, PRMS
$OwnerD(od, t, o)$	Domain od is the owner of the object o in tenant t	PO, Project-role pair, PA
$AuthorizedR(u, t, r)$	User u belonging to tenant t is authorized for the role r	UA, Project-role pair
$BelongsToD(u, d)$	User u belongs to the domain d	UO
$TenantRoleDom(t, r, d)$	Role r is defined within the domain d in tenant t	PO, Project-role pair
$LogEntry(d, t, u, r, o, op)$	Operation op on object o is actually performed by user u having role r in tenant t and domain d	ND

TABLE II. CORRESPONDENCE BETWEEN RELATIONS IN OUR FORMALISM AND RELATIONSHIPS/ENTITIES IN FIGURE 1. NOTE THAT ONE OF THE RELATIONS (IN THIRD COLUMN) IS DENOTED BY ND AS IT IS INFERRED FROM DYNAMIC DATA (E.G., LOGS).

Listing 1. Sugar source code for the common ownership property

```
1  // Declaration
2  (domain Domain 0 500) (domain Tenant 0 1000)
3  (domain Role 0 1000) (domain User 0 60000)
4  ( int  D Domain) (int R Role)
5  ( int  P Tenant) ( int  U User)
6  // Relations  Declarations  and Audit Data as  their  Support
7  ( relation  BelongsToD 2 (supports (100 401) (40569 123)
8  (102 452) (145 404) (156 487) (128 463)))
9  ( relation  AuthorizedR 3 (supports  (100 301 225)
10 (40569 1233 9) (102 399 230) (101 399 231)))
11 ( relation  TenantRoleDom 3 (supports (301 225 401)
12 (1233 9 335) (399 230 452) (399 231 452)))
13 // Security  property :  common ownership
14 ( predicate (ownership D R U P)
15 (and (AuthorizedR  U P R ) (BelongsToD U D)
16 (not(TenantRoleDom P R D)) ))
17 (ownership D R U P)
```

Example 3 *Listing 1 is the CSP code to verify the common ownership property. Each domain and variable are first declared (see Listing 1 lines 2-5). Then, the set of involved relations, namely $BelongsToD$, $AuthorizedR$, and $TenantRoleDom$ are defined and populated with their supporting tuples (see Listing 1 lines 7-12), where the support is generated from actual data in the cloud. Then, the common ownership property is declared as a predicate, denoted by $ownership$, over these relations (see Listing 1 lines 14-16). Finally, the predicate should be instantiated (see Listing 1 line 17) to be able to be verified. As we are formalizing the negation of the properties, we are expecting the UNSAT result, which means that all constraints are not satisfied (i.e., no violation of the property). Note that the predicate will be unfolded internally by the Sugar for all possible values of the variables, which allows to verify each instance of the problem among possible values domains, users and roles. We present the verification outputs in Section V.*

V. APPLICATION TO THE OPENSTACK

This section describes how we integrate our audit and compliance framework into OpenStack. First, we briefly present the OpenStack services that are relevant to the scope of this paper, namely the identity management service (Keystone) and the logging service (Ceilometer). We then detail our auditing framework implementation and its integration in OpenStack along with the challenges that we face and overcome.

A. Background

OpenStack [4] 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 [29] 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 Keystone and Ceilometer:

Keystone [4]. This is the identity service in OpenStack for authentication and authorization. Keystone's internal data including domains, projects, users, and role assignments are stored in a database. Keystone implements the RBAC model.

Ceilometer [4]. This is an OpenStack project aiming to facilitate monitoring and metering. Each component of OpenStack generates notifications, which are triggered by predefined activities such as user creation, role assignment, and are sent to Ceilometer for monitoring purposes. Ceilometer extracts the information from the notifications and transforms it to events with the predefined formats and store all events in its database.

B. Audit framework integration into OpenStack

We focus mainly on three components in our implementation: the data collection engine, the data processing engine, and the compliance validation engine. The collection engine involves several components of OpenStack e.g., Keystone and Neutron for collecting audit data from log files, different OpenStack databases along with policy files and configuration files from the OpenStack ecosystem 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 property holds or not. 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. The main sources of audit data in OpenStack are logs, configuration files, and databases. Table III shows some sample data sources. The OpenStack logs are maintained separately for each service, e.g., Neutron, Keystone, and they are stored in a directory named $var/log/component\_name$, e.g., $keystone.log$ and $keystone\_access.log$ are stored in the directory $var/log/keystone$. Two major configuration files, namely $policy.json$ and $policy.v3cloudsample.json$, contain policy rules defined by both the cloud provider and tenant admins, and are stored in $keystone/etc/$ directory. The third source of data is a collection of databases, hosted in a MySQL server, that can be read using component-specific APIs such as Keystone and Neutron APIs. OpenStack further provides log data through the Ceilometer database to leverage monitoring and metering. With the proper configuration of pyCADF library [30] as a middleware, notifications for specific events in Keystone, Neutron and Nova can be gathered from the Ceilometer database.

Relations	Sources of Data
$AuthorizedOp$	user, assignment, role in Keystone database and $policy.json$ and $policy.v3cloudsample.json$
$OwnerD$	user, assignment in Keystone database and $policy.json$
$AuthorizedR$	user, project, assignment in Keystone database
$BelongsToD$	user, domain tables in Keystone database
$TenantRoleDom$	project, assignment, domain tables in Keystone database
$LoggedEntry$	$keystone\_access.log$ and Ceilometer database

TABLE III. SAMPLE DATA SOURCES IN OPENSTACK

The following describes our techniques to collect audit data from these aforementioned sources and provide implementation details. The main task of the collection engine is to retrieve audit data by executing our collection plug-in written in Python. We initially identify all the aforementioned input sources for collecting audit data from the OpenStack realization. Afterwards, our plug-in retrieves $keystone.log$ and $keystone\_access.log$ along with $policy.json$ and $policy.v3cloudsample.json$. Moreover, our plug-in makes API requests, e.g., *keystone user-role-list*, which lists roles granted to a user, to fetch necessary audit data from the Keystone database. Additionally, to collect notifications through Ceilometer, we make the necessary configurations to enable the pyCADF options in the Keystone middleware. Finally, our collection plug-in fetches data from the Ceilometer database. All audit data collected in this step are stored in the audit repository, which is a MySQL database.

Data processing engine. Our data processing engine, which is implemented in Python, mainly retrieves necessary information from the collected data, converts it into appropriate formats, recovers correlation, and finally generates the source code for Sugar. First, our plug-in fetches the necessary data fields, e.g., API calls, timestamps. Similarly, it fetches access control rules, which contain API names and role names, from $policy.json$ and $policy.v3cloudsample.json$ files. In the next step, our processing plug-in formats each group of data as an n-tuple, i.e., (user, tenant, role, etc.). To facilitate auditing, we additionally correlate different data fields. In the final step, our plug-in uses the n-tuples to generate the portion of Sugar's source code, and append the code with the relationships for security properties (discussed in Section IV). Different scripts are needed to generate Sugar source codes for the verification of different properties, since relationships are usually property-specific.

Compliance Validation. The compliance validation engine is discussed in details in Section IV. Now we show how the auditing results may help detect violation of security properties. In the following example, we discuss how our auditing framework can detect the violation of the common ownership security property, caused by the attack scenario of our running example in Section III-A.

Example 4 *In this example, we describe how a violation of the common ownership property may be caught by auditing. Firstly, our program collects data from different tables in the Keystone database including* $users$, $assignments$, *and* $roles$*. Then, the processing engine converts the collected data and represents as tuples; for our example: (40569 123) (40569 1233 9) (1233 9 335), where Mallory: 40569, Da: 123, Pb: 1233, member: 9 and Db: 335. Additionally, the processing engine interprets the property and generates the Sugar source code (see Listing 1 for an excerpt of the code) using processed data and translated property. Finally, Sugar is used to verify the security properties. We recall the expression for the ownership property,* $(and\ BelongsToD(u,d)\ AuthorizedR(u,t,r)\ (not$ $TenantRoleDom(t,r,d))$. *As Mallory belongs to domain* Da, $BelongsToD(Mallory, Da)$ *evaluates to true. Mallory has been authorized a project-role pair (Pb,member), thus* $AuthorizedR(Mallory, Pb, member)$ *evaluates to true. However,* $TenantRoleDom(Pb, member, Da)$ *evaluates to false, as the pair* $(Pb, member)$ *does not belong to domain* Da*. Then, the whole predicate* $ownership$ *unfolded for that case, would evaluate to true. In this case, the output of sugar (SAT) is the solution of the problem,* $(d = 335, r = 9, t = 1233, u = 40569)$*, which is actually the proof that Mallory violates the common ownership property.*

Challenges. OpenStack usually provides significant amounts of logs, e.g., we observe logs of size 16 GB in our experimental environment. Given the amount of logs generated, processing large log files is a major challenge. In addition, there exist difficulties in locating relevant information, e.g., the initiator of Keystone API calls is missing. To overcome this limitation, we additionally leverage pyCADF and Ceilometer to gather missing information. Unfortunately, notifications provided by the Ceilometer is also inadequate for Keystone, i.e., initiator of an API call is not available. Therefore, to obtain sufficient information about user events to conduct the auditing, we choose to collect Neutron notifications from the Ceilometer database.

The logs generated by each component of OpenStack usually lack correlation. Even though Keystone processes authentication and authorization steps prior to a service access, Keystone does not reveal any correlated data. For this work, we extend our data processing plug-in to deduce correlation between data. For an example, we infer the relation $(user\ operation)$ from the available relations $(user\ role)$ and $(role\ operation)$. In our settings, we have $61,031$ entries in the $(user\ role)$ relations for $60,000$ users. The number of entries is larger than the number of users, because there are some users with multiple roles. With the increasing number of users having multiple roles, the size of this relation grows, and as a result, it increases the complexity of the correlation step. Note that, correlation is required for several of our listed properties and possibly for new properties not in our list. As an example, for the *cyclic inheritance* property, which verifies any existence of cycle in an inheritance relationship, a recursive inheritance relationship is required based on the available direct inheritance relations.

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 Keystone and more generally in OpenStack, all configuration files and all Keystone database tables (18 tables). Through this process, we identify all possible types of data with their sources. Due to the diverse sources of data, there exist inconsistencies in formats of data. On the other hand, to facilitate auditing, presenting data in a uniform manner is very important. Therefore, we facilitate proper formatting within our data processing plug-in.

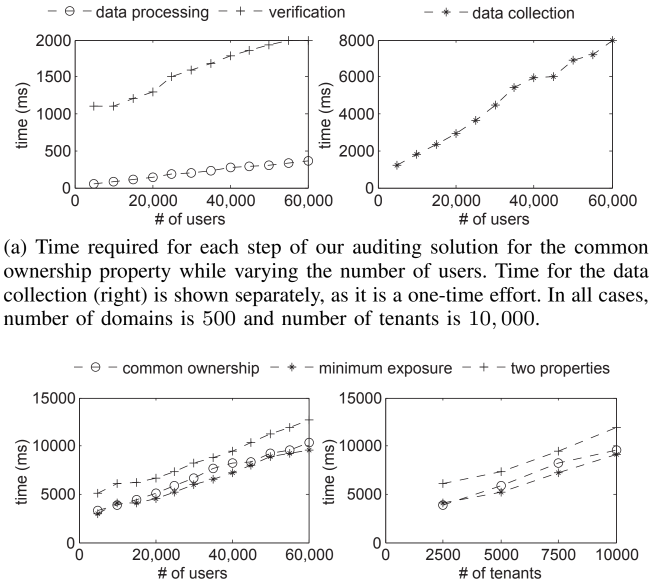
VI. Experiments

This section evaluates the performance of our solution by measuring the execution time, memory, and CPU consumption.

A. Experimental setting

We collect audit data from the OpenStack setup inside a lab environment. Our OpenStack version is DevStack Juno (2014.2.2.dev3) with Keystone API version v3. There are one controller node and three compute nodes, each having Intel i7 dual core CPU and 2GB memory with the Ubuntu 14.04 server. To make our experiments more realistic, we follow statistics in [31], [32] on sizes of real OpenStack environments, which shows 98% of OpenStack clouds (including private, public and hybrid) are having around $50,000$ users. Also, 98% of them contain about $10,000$ tenants. Accordingly, our largest dataset consists of $60,000$ users, $10,000$ tenants, and 500 domains. For verification, we use the V&V tool, Sugar V2.2.1 [27]. We conduct the experiment for 12 different 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 16GB memory and we repeat each experiment $1,000$ times.



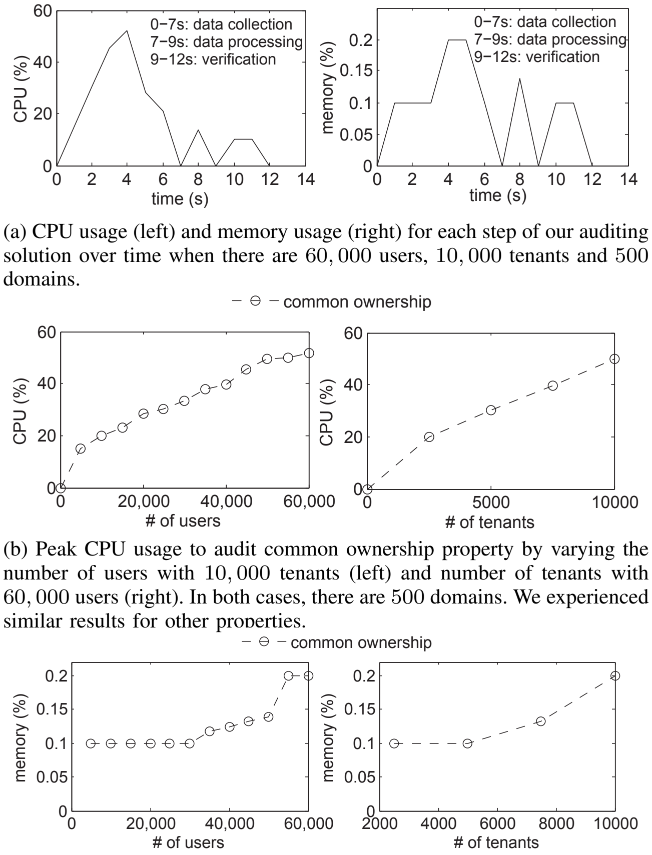
(a) Time required for each step of our auditing solution for the common ownership property while varying the number of users. Time for the data collection (right) is shown separately, as it is a one-time effort. In all cases, number of domains is 500 and number of tenants is $10,000$.

(b) Total time required to audit common ownership, minimum exposure and both properties together, by varying the number of users with fixed $5,000$ tenants (left) and the number of tenants with fixed 30,000 users (right). In all cases, number of domains is 500. Note that time for curves encompass all three steps (collection, processing and verification). For the curve of two properties, data collection is performed one time.

Fig. 3. Execution time for each auditing step and total time for different properties using our framework

B. Results

The objective of the first set of our experiment (see Figure 3) is to demonstrate the time efficiency of our auditing solution. Firstly, Figure 3(a) shows time in milliseconds required for data collection, data processing and compliance validation to verify the *common ownership* property for different cloud sizes (e.g., the number of users). Unlike the processing and verification steps, the data collection step is performed only once for the whole auditing process. Therefore, we show its execution time separately. During this experiment, we change the number of users from $5,000$ to $60,000$ with fixed number of tenants and domains. We can see from the results that the verification execution time is less than 2 seconds for fairly large numbers of users. Knowing that this task happens only once upon each audit request, we believe this is an acceptable overhead for auditing a large setup. Figure 3b shows the total time required for separately auditing *common ownership* and *minimum exposure* properties, and also for both of the properties together. We can easily observe that the execution time is not a linear function of the number of security properties to be verified. In fact, we can see that auditing more security properties would not lead to a significant increase in the execution time.



(a) CPU usage (left) and memory usage (right) for each step of our auditing solution over time when there are $60,000$ users, $10,000$ tenants and 500 domains.

(b) Peak CPU usage to audit common ownership property by varying the number of users with $10,000$ tenants (left) and number of tenants with $60,000$ users (right). In both cases, there are 500 domains. We experienced similar results for other properties.

(c) Peak memory usage to audit common ownership property by varying the number of users with $10,000$ tenants (left) and number of tenants with $60,000$ users (right). In both cases, there are 500 domains. We experienced similar results for other properties.

Fig. 4. CPU and memory usage for each auditing step using our framework

Our second experiment (see Figures 4a(left) and 4b) measures the CPU usage (in %). The left chart in Figure 4a depicts the fact that the data collection step requires significantly higher CPU usage than the other two steps. However, the average CPU usage for data collection is 30%, which is reasonable since the auditing process lasts only a few seconds. Note that, we conduct our experiment in a single PC; if the security properties can be verified through concurrent independent Sugar executions, we can easily parallelize this task by running several instances of Sugar on different VMs in the cloud environment. Thus, performing auditing using the cloud or even with multiple servers possibly reduces the cost significantly. For the other two steps, the CPU cost is around 15%. In Figure 4b, we measure the peak CPU usage (in %) consumed by different steps while auditing the *common ownership* property. Accordingly, the CPU usage grows almost linearly with the number of users and tenants. We observe a significant reduction in the increase rate of CPU usage for datasets with $45,000$ users or more. Note that, other properties show the same trend in CPU consumption, as the CPU cost is mainly influenced by the data collection step, which is property-independent.

Our final experiment (Figures 4a(right) and 4c) measures the memory usage of our auditing solution. The right chart in Figure 4a shows that the data collection step is the most costly in terms of memory usage. However, the highest memory usage observed during this experiment is only 0.2%. Figure 4c shows that the rise in memory consumption is only observed beyond $50,000$ users (left) and $8,000$ tenants (right). We investigated the peak in the memory usage for $50,000$ users and it seems that this is due to the internal memory consumption by Sugar. However, it remains under the boundary of 0.2% throughout our experiment. For the same reason described previously, the observed memory consumption for the other properties shows the same behavior as the reported one.

Discussion. In our experiments, we verified a list of security properties, such as common ownership and minimum exposure, for up to $60,000$ users in less than 15 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 identity and access management properties, the use of formal methods for auditing identify and access management shows very promising results. Particularly, we show that the time required for our auditing solution grows very slowly with the number of security properties. As seen in Fig 3b, an additional security property adds only about 3 seconds. Therefore, we anticipate that auditing a large list of security properties in practice would still be practical. In general, the cost increases almost linearly with the number of users and tenants. Data collection is currently the most costly part of our approach. However, note that each log entry or database record only needs to be collected once, and does not need to be repeated for auditing different properties.

VII. Conclusion

Despite existing efforts, auditing in cloud still faces many challenges. In this paper, we have applied formal methods for auditing identity and access management in a cloud environment. We have proposed a general auditing framework and realized it based on OpenStack. Our evaluation results show that formal methods can be used for large data centers with a reasonable overhead. Our future work will extend the list of security properties presented in sec III. Moreover, we will integrate into our system existing techniques on trusted auditing (e.g., [23]) to establish the trust on the cloud infrastructure.

Acknowledgment

The authors thank the anonymous reviewers for their valuable comments. This material is based upon work partially supported by Natural Sciences and Engineering Research Council Project Grant XXX-YYY,... Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

[1] K. Ren, C. Wang, and Q. Wang, “Security challenges for the public cloud,” *IEEE Internet Computing*, no. 1, pp. 69–73, 2012.

[2] Cloud Security Alliance, “Cloud control matrix CCM v3.0.1,” 2014, available at: https://cloudsecurityalliance.org/research/ccm/.

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

[4] OpenStack, “OpenStack open source cloud computing software,” 2015, available at: http://www.openstack.org.

[5] Open Data Center Alliance, “Open data center alliance usage: Cloud based identity governance and auditing rev. 1.0,” Tech. Rep., 2012.

[6] B. Tang and R. Sandhu, “Extending OpenStack access control with domain trust,” in *Network and System Security*, 2014, pp. 54–69.

[7] A. Gouglidis and I. Mavridis, “domRBAC: An access control model for modern collaborative systems,” *computers & security*, 2012, 31(4).

[8] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, “Verification and change-impact analysis of access-control policies,” in *ICSE*, 2005.

[9] G.-J. Ahn, H. Hu, J. Lee, and Y. Meng, “Representing and reasoning about web access control policies,” in *COMPSAC ’10*, 2010.

[10] K. Arkoudas, R. Chadha, and J. Chiang, “Sophisticated access control via SMT and logical frameworks,” *TISSEC*, vol. 16, no. 4, 2014.

[11] N. Ghosh, D. Chatterjee, S. K. Ghosh, and S. K. Das, “Securing loosely-coupled collaboration in cloud environment through dynamic detection and removal of access conflicts,” *IEEE Trans. on Cloud Comp.*, 2014.

[12] A. Gouglidis, I. Mavridis, and V. C. Hu, “Security policy verification for multi-domains in cloud systems,” *Int. Jour. of Info. Sec.*, 2014, 13(2).

[13] Z. Lu, Z. Wen, Z. Tang, and R. Li, “Resolution for conflicts of inter-operation in multi-domain environment,” *Wuhan University Journal of Natural Sciences*, vol. 12, no. 5, 2007.

[14] H. Kai, H. Chuanhe, W. Jinhai, Z. Hao, C. Xi, L. Yilong, Z. Lianzhen, and W. Bin, “An efficient public batch auditing protocol for data security in multi-cloud storage,” in *ChinaGrid’13*, Aug 2013.

[15] F. Doelitzscher, C. Fischer, D. Moskal, C. Reich, M. Knahl, and N. Clarke, “Validating cloud infrastructure changes by cloud audits,” in *SERVICES’12*, June 2012.

[16] K. Ullah, A. Ahmed, and J. Ylitalo, “Towards building an automated security compliance tool for the cloud,” in *TrustCom’13*, July 2013.

[17] F. Doelitzscher, “Security audit compliance for cloud computing,” Ph.D. dissertation, Plymouth University, 2014.

[18] N. Bjørner and K. Jayaraman, “Checking cloud contracts in Microsoft Azure,” in *Distributed Computing and Internet Technology*, 2015.

[19] IBM Corporation, “Safeguarding the cloud with IBM security solutions,” http://www.ibm.com, Tech. Rep., 2013.

[20] Amazon Web Services, “Security at scale: Logging in AWS,” http://aws.amazon.com, Tech. Rep., November 2013.

[21] N. Meghanathan, “Review of access control models for cloud computing,” *Computer Science & Information Science*, pp. 77–85, 2013.

[22] Microsoft Azure, “Role-based access control in the Azure portal,” http://azure.microsoft.com/en-us, last Visited May 2015.

[23] M. Bellare and B. Yee, “Forward integrity for secure audit logs,” Citeseer, Tech. Rep., 1997.

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

[25] NIST, SP, “NIST SP 800-53,” *Recommended Security Controls for Federal Information Systems*, pp. 800–53, 2003.

[26] W. Jansen, “Inheritance properties of role hierarchies,” in *21st National Information Systems Security Conference*, 1998, pp. 6–9.

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

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

[29] datacenterknowledge.com, “Survey: One-third of cloud users’ clouds are private, heavily OpenStack,” 2015, available at: http://www.datacenterknowledge.com.

[30] Cloud auditing data federation (CADF), “pyCADF: A Python-based CADF library,” 2015, available at: https://pypi.python.org/pypi/pycadf.

[31] OpenStack, “OpenStack user survey statistics November 2013,” 2013, available at: http://www.openstack.org.

[32] getcloudify..org, “OpenStack in numbers - the real stats,” 2014, available at: http://getcloudify.org.