User-Level Runtime Security Auditing for the Cloud

Suryadipta Majumdar, Taous Madi, Yushun Wang,
Yosr Jarraya, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi

***Abstract*—Cloud computing is emerging as a promising IT solution for enabling ubiquitous, convenient, and on-demand accesses to a shared pool of configurable computing resources. However, the widespread adoption of cloud is still being hindered by the lack of transparency and accountability, which has traditionally been ensured through security auditing techniques. Auditing in cloud poses many unique 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 need of runtime verification for the dynamic nature of cloud). To this end, existing runtime auditing techniques do not offer a practical response time to verify a wide-range of user-level security properties for a large cloud. In this paper, we propose a runtime security auditing framework for the cloud with special focus on the user-level including common access control and authentication mechanisms e.g., RBAC, ABAC, SSO, and we implement and evaluate the framework based on OpenStack, a widely deployed cloud management system. The main idea towards reducing the response time to a practical level is to perform the costly operations for only once, which is followed by significantly more efficient incremental runtime verification. Our experimental results show that runtime security auditing in large cloud environment is realistic under our approach (e.g., our solution performs runtime auditing of 100,000 users within 500 milliseconds).**

***Index Terms*—Cloud security, security auditing, compliance verification, runtime verification, user-level security, OpenStack.**

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], [2]. Particularly, the multi-tenancy and ever-changing nature of clouds usually implies significant design and operational complexity, which may prepare the floor for misconfigurations and vulnerabilities leading to violations of security properties. Runtime security auditing may increase cloud tenants' trust in the service providers by providing assurance on the compliance with security properties mainly derived from the applicable laws, regulations, policies, and standards. Evidently, the Cloud Security Alliance has recently introduced the Security, Trust & Assurance Registry (STAR) for security assurance in clouds, which defines three levels of certifications (self-auditing, third-party auditing, and continuous, near real-time verification of security compliance) [3].

However, there are currently many challenges in the area of security auditing for the cloud. Most importantly, the sheer scale of cloud (e.g., a large-size cloud is said to have around 10,000 tenants and 100,000 users [4]), together with its self-provisioning, elastic, and dynamic nature may specially render the overhead of runtime verification process prohibitive. Additionally, there exists a significant gap between the high-level recommendations provided in most cloud-specific standards (e.g., Cloud Control Matrix (CCM) [5] and ISO 27017 [6]) and the low-level logging information currently available in existing cloud infrastructures (e.g., OpenStack [7]). 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.

Existing approaches can be roughly divided into three categories. First, the *retroactive* approaches (e.g., [8], [9]) catch security violations after the fact. Second, the *intercept-and-check* approaches (e.g., [10], [11]) verify security invariants for each user request before granting/denying it. Third, the *proactive* approaches (e.g., [10], [11], [12]) verify user requests in advance. Our work falls into the second category. Therefore, this work potentially prevents the limitation of the retroactive approaches, and also requires no future change plan unlike proactive approaches (e.g., [10], [11]). In comparison with existing intercept-and-check solutions, our approach reduces the response time significantly and supports a wide range of user-level security properties.

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

- Section 13.2.1 of ISO 27017 [6], 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 [7] 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).
- Continuously allocating and deprovisioning of resources and user roles for up to 100,000 users mean any verification results may only be valid for a short time. For instance, a re-verification might be necessary after certain frequently-occurred operations such as: `user create 1c73637`

S. Majumdar, T. Madi, Y. Wang, L. Wang and M. Debbabi are with the CIISE, Concordia University, Montreal, Canada e-mail: (su_majum, t_madi, yus_wang, wang and debbabi@encs.concordia.ca).

Y. Jarraya and M. Pourzandi are with the Ericsson Security Research, Montreal, QC, Canada e-mail: (yosr.jarraya and makan.pourzandi@ericsson.com).

(meaning the 1c73637 user is created), and `role grant member 1c73637` (meaning the member role is granted to the 1c73637 user).

Clearly, during the runtime security auditing, collecting and processing all the data again after each operation can be very costly and may represent a bottleneck for achieving the desired response time due to the performance overhead involved with data collection and processing operations (as reported in Section V). In addition to data collection and processing, runtime verification of ever-changing clouds within a practical response time is essential and non-trivial. In this specific case, no automated tool exists yet in OpenStack for these purposes.

Objective and Contributions. In this paper, we propose a user-level runtime security auditing framework 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 perform costly auditing operations (e.g., data collection and processing, and initial verification on whole cloud) only once during the initialization phase so that later runtime operations can be performed in an incremental manner to reduce the cost of runtime verification significantly with a negligible delay. We rely on formal verification methods to enable automated reasoning and provide formal proofs or counter examples of compliance. We implement and integrate the proposed runtime auditing framework into OpenStack, and report real-life experiences and challenges. Our framework supports several popular cloud access control and authentication mechanisms (e.g., role-based access control (RBAC) [13], attribute-based access control (ABAC) [14] and single sign-on (SSO)) with the provision of adding such more extensions. Our experimental results confirm the scalability and efficiency of our approach.

The main contributions of this paper are as follows:

- We propose an efficient user-level runtime security auditing framework in a multi-domain cloud environment.
- The study on security properties provides a bridge between the cloud security standards and the literature on multi-domain access control and authentication.
- Our prototype system can be a part of the auditing system for OpenStack-based cloud infrastructure management systems providing a practical auditing solution with the support of common access control and authentication mechanisms (e.g., RBAC, ABAC and SSO).
- The experimental results show that our proposed system is realistic for large-scale cloud environments (e.g., the response time is less than 500 ms for a large cloud with 100,000 users).
- In contrast to our previous work [9], which adopts a retroactive approach that can only catch a security compliance violation after the fact, we are proposing a different, runtime approach in this paper. The major extensions include: i) a new auditing framework for catching security violations at runtime (Section III); ii) the support of not only RBAC but also attribute-based access control (ABAC) and single sign-on (SSO) (Sections II-A, II-B and III-D); iii) new algorithms and implementation for supporting incremental verification (Sections IV-B and IV-C); and iv) new evaluation results on runtime auditing (Section V).

The rest of this paper is organized as follows. Section II presents attack scenarios and security properties. Section III describes our methodology. Section IV discusses system architecture and implementation details. Section V gives experimental results. Section VI discusses several aspects of our approach. Section VII reviews the related work. Section VIII concludes our paper discussing future directions.

II. User-Level Security Properties

We first show different attack scenarios based on authorization and authentication models. Then we formulate user-level threats as a list of security properties mostly derived from the cloud-specific standards, and finally discuss our threat model.

A. Models

We now describe RBAC, ABAC and SSO models.

RBAC Model. We focus on verifying multi-domain role-based access control (RBAC), which is adopted in real world cloud platforms (as shown in Table I). In particular, we assume the extended RBAC model as in [15], which adds multi-tenancy support in the cloud. The definitions of different components of this model can be found in [15].

Plugins	Cloud Platforms				
	OpenStack [7]	Amazon EC2 [16]	Microsoft Azure [17]	Google GCP [18]	VMware [19]
RBAC	•	•	•	•	•
ABAC	Blueprint [20]	•	Azure AD	Firebase	•
SSO	Federation	AWS Directory	Microsoft account	G Suite	myOneLogin

TABLE I
USAGE OF RBAC, ABAC AND SSO IN MAJOR CLOUD PLATFORMS

Example 1 Figure 1 depicts our running example, which is an instance of the access control model presented in [15]. 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 tenants[1] respectively owned by the two domains. In such a 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. Therein, Mallory belonging to domain Da is assigned a tenant-role pair (Pb, Member), which is from domain Db. This violates the security requirement of these domains as they do not trust each other.

ABAC Model. ABAC [14], is considered as a strong candidate to replace RBAC in Sandhu [21], which identifies several limitations of RBAC and thus emphasizes the importance of ABAC specially for large infrastructures (e.g., cloud). In fact, major cloud platforms have started supporting ABAC (as shown in Table I). The definitions of different components of this model can be found in [14]. We mainly use two ABAC functions, i.e., user attribute (*UATT*) and object attribute (*OATT*).

[1] We interchangeably use the terms, tenant and project in Figures 1 and 2

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

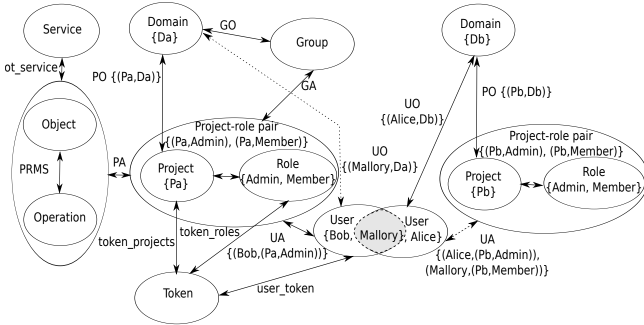


Fig. 1. Two domain instances of the access control model of [15] depicting the resultant state of the access control system after the exploit of the vulnerability, OSSN-0010. The shaded region and dotted arrows show an instance of the exploit described in Example 1.

Example 2 Figure 2 depicts our running example for ABAC, and shows a similar attack scenario as Example 1. The model in the figure is an instance of the access control model presented in [22]), and shows the resultant state of the access control system after this attack.

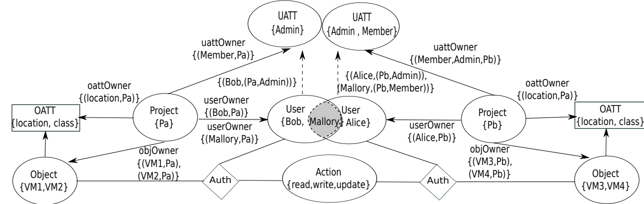


Fig. 2. Two tenant instances of the access control model of [22] depicting the resultant state of the access control system after the exploit of the vulnerability, OSSN-0010. The shaded region and dotted arrows show an instance of the exploit described in Example 2.

SSO Mechanism. SSO, which is a popular cloud authentication extension and supported by major cloud platforms (shown in Table I), only requires a single user action to permit a user to access all authorized computers and systems. In this work, we detail two SSO protocols: OpenID [23] and SAML [24] supported by OpenStack and many other cloud platforms.

However, there are several attacks (e.g., [25], [26], [27], [28]) on two above-mentioned SSO protocols. The following describes several security concerns specific to these protocols.

- In SAML, there is no communication between service provider (SP) and identity provider (IdP). Therefore, an SP maintains a list of trusted IdPs, and any ID generated by an IdP that is not in this list must be strictly restricted.
- On the other hand, OpenID accepts any IdP by communicating with the corresponding relying party (RP), which provides the login service as a third party. Therefore, a proper synchronization between IdP and RP is essential for the OpenID protocol. Otherwise, it may result following security critical incidents:
  - Logging out from IdP may not ensure logging out from RP, and thus an unauthorized session is possible.
  - Linking to an existing account with an OpenID without any authentication may result unauthorized access.

To address such security concerns and to be compliant with aforementioned cloud-specific security standards, we devise security properties in the next subsection.

B. Security Properties

Table II presents an excerpt of the list of user-level security properties that we identify from the access control and authentication literature, relevant standards (e.g., ISO 27002 [29], NIST SP 800-53 [30], CCM [5] and ISO 27017 [6]), and the real-world cloud implementation (e.g., OpenStack). Even though some properties (e.g., cyclic inheritance) are not directly found in any standard, they are included based on their importance and impact described in the literature (e.g., [31]).

RBAC Security Properties. RBAC-specific security properties are shown in Table II. For our running example, we will focus on following two properties. Common ownership: based on the challenges of multi-domain cloud discussed in [31], [15], 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 [15].

Example 3 The attack scenario in Example 1 violates the common ownership property. According to the property, Mallory must not hold a role member in tenant Pb belonging to domain Db, because Mallory belongs to domain Da and there exists no collaboration between domains Da and Db.

Properties	Standards				RBAC	ABAC	SSO
	ISO27002 [29]	ISO27017 [6]	NIST800 [30]	CCM [5]			
Role activation [32]	*13.2.2b*	*15.2.2b*	*AC-1*	*IAM-09*	•	•	
Permitted action [32]	*11.2.1.b, 1.2.2c*	*13.2.1b, 13.2.2c*	*AC-14*	*IAM-10*	•	•	
Common ownership [31]	*11*	*13*	*AC*	*IAM*	•	•	
Minimum exposure [15]	*11.6.1*	*13.4.1*	*AC-4*	*IAM-04,06*	•	•	
Separation of duties [15]	*11.6.2*	*13.6.2*	*AC-5*	*IAM-02,05*	•		
Cyclic inheritance [31]					•		
Privilege escalation [31]	*11.2.2.b*	*13.2.2b*	*AC-6*	*IAM-08*	•	•	
Cardinality [32]	*11.2.4*	*13.2.4*	*AC-1*		•	•	
Consistent constraints [33]							
add/delete user	*11.5.1*	*13.4.2*	*AC-7,9*	*IAM-02*		•	
modify user attributes	*13.2.2b*	*15.2.2b*	*AC-1*	*IAM-09*		•	
add/delete object	*13.2.2b*	*15.2.2b*	*AC-1*	*IAM-09*		•	
modify object attributes	*13.2.2b*	*15.2.2b*	*AC-1*	*IAM-09*		•	
Session de-activation [30]	*11.5.5*	*13.2.8*	*AC-12*		•	•	
User-access validation [6]	*11.5.2*	*13.4*	*AC-3*	*IAM-10*			•
User-access revocation [29]	*11.2.1h*	*13.2.1h*	*AC-2*	*IAM-11*			•
No duplicate ID [5]	*11.5.2*	*13.5.2*	*AC-2*	*IAM-12*			
Secure remote access [6]	*11.4.2*	*13.4.2*	*AC-17*	*IAM-02,07*			
Secure log-on [6]	*11.5.1*	*13.4.2*	*AC-7,9*	*IAM-02*			
Session time-out [30]	*11.5.5*	*13.2.8*	*AC-12*				•
Concurrent session [30]		*13.5.4*	*AC-10*				
Brute force protection	*11.2.2.b*	*13.2.2b*	*AC-1*	*IAM-09*			•
No caching	*11.2.1.b, 1.2.2c*	*13.2.1b, 13.2.2c*	*AC-14*	*IAM-10*			•

TABLE II
AN EXCERPT OF USER-LEVEL SECURITY PROPERTIES

ABAC Security Properties. Table II provides an excerpt of ABAC-related security properties supported by our auditing system. Some properties are specific to ABAC, and rests are adopted from RBAC. We only discuss the following properties, which are extended or added for ABAC.

- Consistent constraints: Jin et al. [33] define constraints for different basic changes in ABAC elements e.g., adding/deleting users/objects. After each operation, certain changes are necessary to be properly applied. This property verifies whether all constraints have been performed.
- Common ownership: For ABAC, the common ownership property also includes objects and their attributes so that an object owner and the owner of the allowed user performing certain actions on that object must be the same.

Authentication-related Security Properties. Table II shows an excerpt of security properties related to generic authentication mechanisms and extensions (e.g., SSO). We discuss the SSO-related properties as follows. Brute force protection:

account lockout, CAPTCHA or any such brute force protection must be applied in SSO. No caching: SSO and associated applications should set no-cache and no-store cache directives. User access revocation: logout from one application must end sessions of other applications. User access validation: only a valid authentication token must pass the authentication step.

C. Threat Model

Our threat model is based on two facts. First, our solution focuses on verifying the security properties specified by cloud tenants, instead of detecting specific attacks or vulnerabilities (which is the responsibility of IDSes or vulnerability scanners). Second, the correctness of our verification results depends on the correct input data extracted from logs and databases. Since an attack may or may not violate the security properties specified by the tenant, and logs or databases may potentially be tampered with by attackers, our results can only signal an attack in some cases. Specifically, the in-scope threats of our solution are attacks that violate the specified security properties and at the same time lead to logged events. The out-of-scope threats include attacks that do not violate the specified security properties, attacks not captured in the logs or databases, and attacks through which the attackers may remove or tamper with their own logged events. More specifically, in this work we focus on user-level security threats and rely on existing solutions (e.g., [11], [12]) to identify virtual infrastructure level threats. We assume that, before our runtime approach is launched, an initial verification is performed and potential violations are resolved. However, if our solution is added from the commencement of a cloud, obviously no prior security verification (including the initial phase) is required. We also assume that tenant-defined policies are properly reflected in the policy files of the cloud platforms.

III. Runtime Security Auditing

This section presents our runtime security auditing framework for the user-level in the cloud.

A. Overview

Figure 3 shows an overview of our runtime auditing approach. This approach contains two major phases: i) initialization, where we conduct a full verification on the collected and processed cloud data, and ii) runtime, where we incrementally verify the compliance upon dynamic changes in the cloud. The initialization phase is performed only once through an offline verification. This phase performs all costly operations such as data collection and processing, and an initial full compliance verification for a list of security properties. The initial verification result is stored in the result repository. For the latter, we devise an incremental verification approach to minimize the workload at the runtime. During the runtime phase, each management operation (e.g., create/delete a user/tenant) is intercepted, its parameters are processed with the previous result, and finally the verification engine evaluates the compliance and provides the latest verification result. We elaborate major phases of our system as follows.

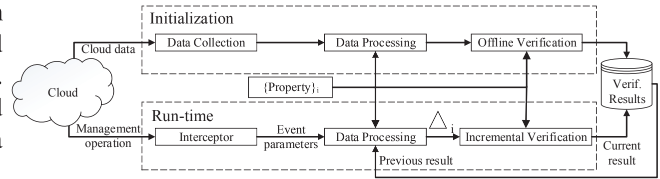


Fig. 3. An overview of our runtime security auditing approach

B. Initialization Phase

Our runtime auditing approach at first requires one-time data collection and processing, and full verification of the cloud, namely the initialization phase, against a list of security properties. Initially, we collect all necessary data from the cloud, which are in different formats and in different levels of abstractions. Therefore, we further process these data to convert the format and correlate them to identify required relationships for considered security properties. Then, we generate inputs for the verification engine incorporating the processed data in the previous step. Finally, the verification engine checks the compliance for a list of security properties, and provides an initial verification result.

The collection engine is responsible for collecting the required data in a batch mode from the cloud management system. The role of the processing engine is to filter, format, aggregate, and correlate this data. The required data is 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 final processing step is to generate and store the inputs to be used by the compliance verification engine. Note that the format of the inputs depends on the selected back-end verification engine.

The compliance verification engine is to perform the actual verification of the security properties. We use formal methods to capture the system model and to verify properties, which facilitates automated reasoning and is generally more practical and effective than manual inspection. If a security property is violated, evidences can be obtained from the output of the verification back-end, e.g., a set of real data in the cloud for which all conditions of a security property are not satisfied are provided as a feedback. Once the outcome of the initial verification is ready, results and evidences are stored in the result repository and made accessible to the runtime engine.

C. Runtime Phase

The initialization phase conducts an offline verification, where we verify security properties on the whole cloud. However, verifying the whole cloud after each configuration change is very expensive. Alternatively, we intercept each event and verify the impact of the events in an incremental manner, where we perform runtime verification on a minimal dataset based on the current change to provide a better response time, to catch a security violation.

We update the verification results continuously by verifying the cloud at runtime. The runtime verification is event-driven and property-specific. Table III shows the events that may affect the result of verification for certain properties. The bottom part of Figure 3 depicts the steps of this phase. We

intercept each operation request generated from the cloud management interface. We further retrieve the parameters of the request and pass them to the data processing module. The data processing module performs similarly as described in Section III-B. However, during the runtime phase, mostly partial data are sent for the incremental verification for each security property. Thus the incremental verification is only conducted on the impact of the current change. Then, the final verification result is inferred from the result of current incremental verification and the previous result. Incremental verification is performed using any of the two methods: i) *deltaVerify*, where compliance verification mechanism discussed in the initialization phase is applied on the delta data, and ii) *customAlgo*, where security property specific customized algorithms are performed. We discuss our runtime verification algorithm in more details in Section IV-B. In the following examples, we assume that the previous verification result for a specific property is stored in $Result_{t0}$, the parameters of the intercepted event is in Δ_i and the updated result will be stored in $Result_t$. For example, all user-role pairs violating the common ownership property at time $t0$ are stored in $Result_{t0}$ as $\{Mallory, Da, (Pb, Member), Db\}$.

Example 4 Table III shows that the verification result for the common ownership property may change for following events: grant role, delete role, delete user, delete tenant and delete domain. Upon intercepting any of these events, we conduct incremental verification as follows:

- Grant a role: Each role assignment alone may affect the common ownership property, and hence, it does not depend on the previous assignments. Therefore, upon a grant role request, we only verify the parameters of the intercepted event using the *deltaVerify* method, and combine the obtained result with the previous result ($Result_{t0}$) to infer the updated result ($Result_t$).
- Delete a role: If the deleted role (Δ_i) is present in the previous result (i.e., $\Delta_i \in Result_{t0}$), then we update the current result by removing that role (i.e., $Result_t = Result_{t0} - \Delta_i$). Otherwise, the result remains unchanged (i.e., $Result_t = Result_{t0}$). Deleting a user/tenant/domain can be similarly handled.

Example 5 Similarly, upon intercepting any of the events marked for the permitted action property in Table III, we conduct incremental verification as follows:

- Grant a role: If the granted role (Δ_i) is present in the previous result (i.e., $\Delta_i \in Result_{t0}$), then we update the current result by removing that role (i.e., $Result_t = Result_{t0} - \Delta_i$). Otherwise, the result remains unchanged (i.e., $Result_t = Result_{t0}$).
- Delete a role: If the deleted role (Δ_i) is present in the previous result (i.e., $\Delta_i \in Result_{t0}$), then we update the current result by removing that role (i.e., $Result_t = Result_{t0} - \Delta_i$). Otherwise, the result remains unchanged (i.e., $Result_t = Result_{t0}$). Deleting a user/tenant/domain can be similarly handled.

Identifying the Impacts of Events. By observing the impacts of cloud events, Table III lists all events that may change the verification result of certain security properties. However, identifying impacts of events in cloud can be challenging. Also, the completeness of the method of identifying the impacts, relies on the specifications of APIs by the cloud platforms. In this work, we mainly follow two methods (i.e., API documentation and infrastructure change inspection) proposed by Bleikertz et al. [11]. Firstly, we go through the API documentation provided by cloud platforms to obtain API specifications including their functionality, parameters and impacts on the infrastructure. Secondly, we perform different events and observe the infrastructure configuration change to capture the impact of those events. Finally, we combine this knowledge with the definition of security properties to populate Table III.

Properties	Events																					
	create user	create role	create tenant	create domain	create operation	create object	delete user	delete role	delete tenant	delete domain	delete operation	delete object	grant role	revoke token	enable user	enable role	enable tenant	enable domain	disable user	disable role	disable tenant	disable domain
Common Ownership							•	•	•	•			•									
Permitted Action							•	•	•	•			•									
Minimum Exposure					•	•				•	•	•										
Role Activation							•	•	•	•												
Separation of Duties							•	•	•	•			•									
Privilege Escalation		•					•	•	•	•			•									
Cardinality		•					•	•	•	•			•			•	•	•	•	•	•	•
Cyclic Inheritance		•						•														
No Duplicate ID	•	•	•	•	•	•	•	•	•	•	•	•										
User Access Validation								•						•						•		
User Access Revocation							•	•	•	•				•		•	•	•	•	•	•	•
Secure Remote Access	•												•	•	•							
Secure Log-on	•												•	•	•							
Session Time-out							•	•	•	•				•		•	•	•	•	•	•	•
Concurrent Session							•		•	•									•		•	•

TABLE III
EVENTS THAT INFLUENCE VERIFICATION RESULTS FOR CERTAIN PROPERTIES

Provision of Enriching the Security Property List. Beside the security properties in Section II-B, tenants might intend to add new security properties over time. Our framework provides the provision of adding new security properties through following simple steps. First, the new security property is defined in the cloud system context, which can be simply performed by following our existing techniques discussed in Section II-B to apply high-level standard terminologies to cloud specific resources. Next, the property is translated to the first order logic and then to Constraint Satisfaction Problems (CSP) constraints, and in many cases the existing relations discussed in Section III-D can be re-used as they include basic relations such as *belongs to, owner of, authorized for,* etc. Our data collection engine already collects data from all relevant sources of data of a cloud platform regardless of security properties. Therefore, no extra effort is needed in the data collection phase, unless the new property requires data from a different layer in the cloud (e.g., SDN). Then, the data processing effort for a new property mainly involves building correlation between data from different sources, because other processing steps are mostly property-independent. The remaining initial verification step is only to add constraints of the new property to the verification list. Finally, we identify the events that may alter the verification result of the new property by re-utilizing the knowledge of impacts of events, and perform the runtime verification through incremental steps either using the deltaVerify method or by a customized algorithm (as in Section IV-B). Additionally, whenever there is any change in the event specification for a cloud system, we capture the update on impacts (if any) of events on the security properties.

D. Formalization of Security Properties

As a back-end verification mechanism, we formalize verification data and properties as Constraint Satisfaction Problems

(CSP) and use a constraint solver, namely Sugar [34], 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. In our case, we formalize each security property in CSP to verify in Sugar. After verification, Sugar provides proper evidence (a.k.a counter examples) of a violation (if any) of a security property. In the following, we first provide a generic description of model formalization, then illustrate examples of property formalization, and finally show some counter examples for those security properties.

Model Formalization. Referring to Figures 1 and 2, 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 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 verification 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.

Properties Formalization for RBAC. Security properties are presented as predicates over relation constraints and predicates. We detail two representative properties in this paper. We first express these properties in first order logic [35] and then in their CSP formalization (using Sugar syntax). Table IV summarizes the relations that we use in these properties.

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

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

The corresponding CSP constraint is

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

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.

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

The CSP constraint for this property is:

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

Properties Formalization for ABAC. For space limitation, we show formalization of one security property for ABAC.

Common ownership: Formally the common ownership property is violated in the following conditions: $userOwner(u) \neq uattOwner(userRole_i(u))$ OR $objOwner(o) \neq oattOwner(objRole_{i,j}(o))$. Through this extension, we complement the previous definition, and the property is now more general in the sense that we can identify the misconfiguration in defining policies for an object. Following example further explains this benefit. Alice is a user (from the user set, U) owned by the domain, $d1$. Alice holds a member role in the domain, $d2$, expressed as $userRole_2(Alice)$. The owner of this role is the domain, d2 (inferred from the $uattOwner(userRole2(Alice))$ relationship). This situation violates the common ownership property, as the first part of the condition (i.e., $userOwner(u) \neq uattOwner(userRole_i(u))$) is true. Additionally, there is an object i.e., VM1 (from the object set O) owned by the domain, $d2$. The policy related to $VM1$ states that a user with the $member$ role of the $d2$ domain can $read$ from $VM1$ (as described in $objRole1, 2(VM1)$). To verify the owner of the role that policy allows certain action on the object using the $oattOwner(objRole1, 2(VM1))$ relation. In this case, $objOwner(o) \neq oattOwner(objRole_{i,j}(o))$ is false; hence, the property is preserved.

Since ABAC is more expressive, there might be a larger set of properties for ABAC (as shown in Table II). However, the verification complexity depends more on the security properties, and less on the model. For example, the common ownership, permitted action and minimum exposure properties show different level of complexities, as shown through their formal representation and as supported by the experiment results in Section V.

Properties Formalization for SSO. Due to space limitation, we present formalization steps of one SSO related security property (i.e., user access revocation). The user access revocation property is for the token-based user access. At a given time, for active tokens, we check that none of the situations

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, tenant-role pair, PA, PRMS
$OwnerD(od,t,o)$	Domain od is the owner of the object o in tenant t	PO, tenant-role pair, PA
$AuthorizedR(u,t,r)$	User u belonging to tenant t is authorized for the role r	UA, tenant-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, tenant-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
$ActiveToken(tok,d,t,u,r,time)$	Token tok is active at time $time$ and in use by user u having role r in tenant t and domain d	UA, token_tenants, token_roles, PO, tenant-role pair

TABLE IV

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).

leading to their revocation has been occurred. Function *TimeStamp(tok)* returns the token expiration time.

$$\begin{aligned} &\forall \mathtt{tk} \in \mathtt{Token}, \forall \mathtt{r} \in \mathtt{Role}, \forall \mathtt{t} \in \mathtt{Tenant}, \\ &\quad \forall \mathtt{u} \in \mathtt{User}, \forall \mathtt{d} \in \mathtt{Domain} \\ &\mathtt{ActiveToken(tk,d,t,u,r,Time)} \longrightarrow \\ &\mathtt{AuthorizedR(u,t,r)} \wedge \quad \mathtt{IsActiveR(r,t,u)} \wedge \\ &\mathtt{BelongsToD(u,d)} \wedge \mathtt{IsValidU(u)} \wedge \mathtt{IsValidD(d)} \\ &\wedge \mathtt{IsvalidT(t)} \wedge \mathtt{TimeStamp(tk)} > \mathtt{Time} \end{aligned} \tag{5}$$

Thus, the corresponding CSP constraint is:

$$\begin{aligned} &\mathtt{(and\ \ ActiveToken(tk,d,t,u,r,time)(or\ \ (not} \\ &\mathtt{(not\ \ AuthorizedR(u,t,r))(not\ \ IsActiveR(r,t,u))} \\ &\mathtt{(IsValidU(u))(notIsvalidT(t))(notBelongsToD(u,d))} \\ &\mathtt{(notIsValidD(d))(not(>TimeStamp(tk)Time))))} \end{aligned} \tag{6}$$

Evidences of Violations. Our auditing system using the formal verification tool, Sugar, individually identifies the causes (a.k.a. counter examples) for each security property violated in the cloud. With the following examples, we show how our system can locate the cause of the violations.

Example 6 The CSP predicate for the common ownership property is as follows: $(and\ BelongsToD(u,d)\ AuthorizedR(u,t,r)\ (not\ Tenant\text{-}RoleDom(t,r,d)))$. The property is violated when a user from one domain holds a tenant-role pairs from another domain. In other words, in case of a violation there exists at lease a set of predicates as follows: $(and\ BelongsToD(u1,d1)\ AuthorizedR(u1,t2,r2)\ (not\ Tenant\text{-}RoleDom(t2,r2,d1)))$; meaning that the user $u1$ from domain $d1$ holds a role pair $t2$-$r2$, which is not from domain $d1$. In such cases, our auditing system using Sugar identifies that the (u1, d1, t2, r2) tuple is the cause for a violation of the common ownership property. In Section IV, Example 7 further extends this example to show concrete examples of evidences provided by our auditing system.

IV. IMPLEMENTATION

In this section, we first illustrate the architecture of our system. We then detail our auditing framework implementation and its integration into OpenStack along with the challenges that we face and overcome.

A. Architecture

Figure 4 shows a high-level architecture of our runtime verification framework. It has three main components: data collection and processing module, compliance verification module, and dashboard & reporting module. In the following, we describe different engines inside the data collection and processing module. The security property extractor identifies the sources of required data for a list of security properties. The event intercepter intercepts each management operations requested by the user in the cloud infrastructure system. The data collection engine 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. Then the data processing engine aids to build the correlation and to uniform the collected data. Our compliance verification module is responsible for the offline and runtime verification using the formal verification and validation (V&V) tools and our custom algorithms. Finally, the dashboard & reporting module interacts with the cloud tenant through the dashboard to obtain the tenant requirements and to provide the tenant with the verification results in a report. Tenant requirements encompass both general and tenant-specific security policies, applicable standards, as well as verification queries.

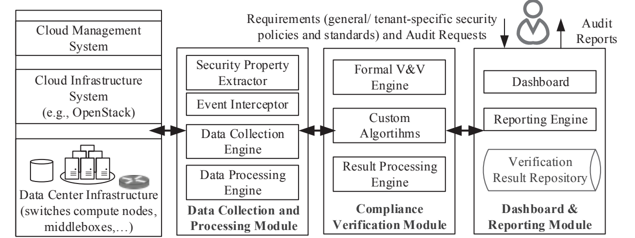


Fig. 4. A high-level architecture of our runtime verification framework

B. Integration into OpenStack

We focus mainly on three components in our implementation: the data collection and processing module, the compliance verification module and dashboard & reporting module. In the following, we first provide background on OpenStack, and then describe our implementation details.

Background. OpenStack [7] is an open-source cloud infrastructure management platform in which Keystone is its identity service, Neutron is its network component, Nova is its compute component, and Ceilometer is its telemetry.

Data Collection Engine. The collection engine involves several components of OpenStack e.g., Keystone and Neutron for collecting data from log files, policy files, different OpenStack databases and configuration files from the OpenStack ecosystem to fully capture the configuration. We present hereafter different sources of data in OpenStack along with the current support for auditing offered by OpenStack. The main sources of data in OpenStack are logs, configuration files, and databases. Table V shows some sample data sources. The OpenStack logs are maintained separately for each service, e.g., Neutron, Keystone, in a directory named $var/log/component_name$, e.g., $keystone.log$ and $keystone_access.log$ are stored in the

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, tenant, assignment in Keystone database
$BelongsToD$	user, domain tables in Keystone database
$TenantRoleDom$	tenant, assignment, domain tables in Keystone database
$LoggedEntry$	$keystone_access.log$ and Ceilometer database
$ActiveToken$	Keystone database and $keystone_access.log$

TABLE V
SAMPLE DATA SOURCES IN OPENSTACK FOR RELATIONS IN TABLE IV

$var/log/keystone$ directory. 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 the $keystone/etc/$ 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. With the proper configuration of the OpenStack middleware, notifications for specific events in Keystone, Neutron and Nova can be gathered from the Ceilometer database.

The effectiveness of a verification solution critically depends on properly collected evidences. 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 verification, presenting data in a uniform manner is very important. Therefore, we facilitate proper formatting within our data processing engine.

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 tool fetches the necessary data fields from the collected data, e.g., identifiers, API calls, timestamps. Similarly, it fetches access control rules, which contain API and role names, from $policy.json$ and $policy.v3cloudsample.json$ files. In the next step, our processing engine formats each group of data as an n-tuple, i.e., (user, tenant, role, etc.). To facilitate verification, we additionally correlate different data fields. In the final step, the n-tuples are used to generate the portion of the Sugar's source code, and the relationships for security properties (discussed in Section III-D) are also appended with the code. Different scripts are needed to generate the Sugar source codes for the verification of different properties, since relationships are usually property-specific.

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. Therefore, we build the data correlation support within the processing engine. 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.

Initial Compliance Verification. The compliance verification module contains two major modules responsible for the initial verification and runtime verification respectively. The prerequisite formalization steps of the initial verification are already discussed in Section III-D. Here, we explain different parts of a Sugar source code through a simple example and verification algorithm (as in Algorithm 1) in the following.

Listing 1. Sugar source code for the common ownership property

```
// Declaration
(domain Domain 0 500) (domain Tenant 0 1000)
(domain Role 0 1000) (domain User 0 60000)
( int D Domain) (int R Role)
( int P Tenant) ( int U User)
// Relations Declarations and Audit Data as their Support
( relation BelongsToD 2 (supports (100 401) (40569 123)
(102 452) (145 404) (156 487) (128 463)))
( relation AuthorizedR 3 (supports (100 301 225)
(40569 1233 9) (102 399 230) (101 399 231)))
( relation TenantRoleDom 3 (supports (301 225 401)
(1233 9 335) (399 230 452) (399 231 452)))
// Security Property : Common Ownership
( predicate (ownership D R U P)
(and (AuthorizedR U P R ) (BelongsToD U D)
(not(TenantRoleDom P R D)) ))
(ownership D R U P)
```

Example 7 Listing 1 is the CSP code to verify the common ownership property. Each domain and variable are first declared (lines 2-5). Then, the set of involved relations, namely $BelongsToD$, $AuthorizedR$, and $TenantRoleDom$ are defined and populated with their supporting tuples (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 (lines 14-16). Finally, the predicate is instantiated (line 17) 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 is unfolded internally by the Sugar engine for all possible values of the variables, which allows to verify each instance of the problem among possible values of domains, users and roles.

In this example, we also describe how a violation of the common ownership property may be caught by our verification process. 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 (as Listing 1) using processed data and translated property. Finally, the Sugar engine is used to verify the security properties. The CSP predicate for the common ownership property is as follows: $(and\ BelongsToD(u,d)\ AuthorizedR(u,t,r)\ (not\ Tenant-RoleDom(t,r,d))$. As Mallory belongs to domain Da, $BelongsToD(Mallory, Da)$ evaluates to true. Mallory has been authorized a tenant-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 $ownership$ predicate unfolded for this case is evaluated to true. In this case, the output of sugar is SAT, which confirms that Mallory violates the common ownership property and further presents the cause of the violation, i.e., $(d = 123, r = 9, t = 1233, u = 40569)$.

Algorithm 1: Runtime Compliance Verification
procedure INITIALIZE(Properties,CloudOS)
  rawData = collectData(CloudOS)
  verData = processData(rawData)
  for each property $p_i \in Properties$ **do**
    $Result_{t0,pi}$= Verify (p_i,verData)
procedure RUNTIME(Event,$Result_{t0}$,Properties)
  for each property $p_i \in Properties$ **do**
    Δ_i = processData(event.parameters)
    if incremental-method(p_i) = $custom$ **then**
      custom-algo($event$,p_i,$Result_{t0,pi}$,Δ_i)
    else
      deltaVerify($event$,p_i,$Result_{t0,pi}$,Δ_i)
  return $Result_t$
procedure DELTAVERIFY($event$,p_i,$Result_{t0,pi}$,Δ_i)
  $Result_{t,pi}$ = verify(p_i,Δ_i)

Runtime Verification. Our runtime verification engine implements Algorithm 1. Firstly, the interceptor module intercepts each management operations based on the existing intercepting methods (e.g., audit middleware [36]) supported in OpenStack. Events are primarily created via the notification system in OpenStack; Nova, Neutron, etc. emit notifications in a JSON format. Here, we leverage the audit middleware in Keystone to intercept Keystone, Neutron and Nova events by enabling the audit middleware and configuring filters. Secondly, the data processing engine handles the intercepted parameters to perform similar data processing operations as discussed previously. The processed data is denoted as Δ_i. Finally, the runtime verification engine performs incremental steps either using the deltaVerify method, which involves Sugar, or custom algorithms. Figures 5 and 6 show the incremental steps for the common ownership and permitted action properties respectively.

There exist difficulties in locating relevant information, e.g., the initiator of Keystone API calls is missing, and in obtaining adequate notifications from Ceilometer for Keystone events. Therefore, to obtain sufficient and proper information about user events to conduct the auditing, we collect Neutron notifications from the Ceilometer database.

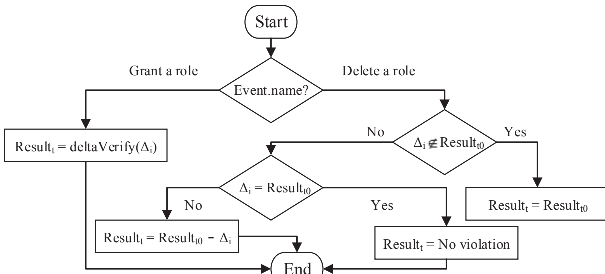


Fig. 5. Showing the runtime steps for the common ownership property

Dashboard & Reporting Module. We further implement the web interface (i.e., dashboard) in PHP to place audit requests and view audit reports. In the dashboard, tenant admins can initially select different standards (e.g., ISO 27017, CCM V3.0.1, NIST 800-53, etc.). Afterwards, security properties under the selected standards can be chosen. Additionally, admins can select any of the following verification options: i) runtime verification, and ii) retroactive verification. Once the verification request is processed, the summarized verification results are shown and continuously updated in the verification report page. The details of any violation with a list of evidences are also provided. Moreover, our reporting engine archives all the verification reports for a certain period.

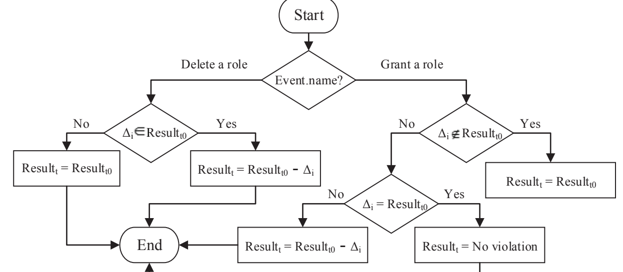


Fig. 6. Showing the runtime steps for the permitted action property

C. Integration to OpenStack Congress

To demonstrate the service agnostic nature of our framework, we further integrate our auditing method with OpenStack Congress [10]. Congress implements policy as a service in OpenStack in order to provide governance and compliance for dynamic infrastructure. Congress can integrate third party verification tools using a data source driver mechanism. Using Congress policy language that is based on Datalog, we define several tenant specific security policies as same as security properties described in Section II-B. We then use our processed data to detect those security properties for multiple tenants. The outputs of the data processing engine in both cases of initialization and runtime are in turn provided as inputs for Congress to be asserted by the policy engine. This integrates compliance status for some policies whose verification is not yet supported by Congress (e.g., permitted action, minimum exposure).

V. EXPERIMENTS

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

A. Experimental Settings

We collect data from the OpenStack setup inside a lab environment. Our OpenStack version is Mitaka (2016.10.15) with Keystone API version v3. There are one controller node and three compute nodes, each having Intel i7 dual core CPU and 2 GB memory with the Ubuntu 16.04 server. To make our experiments more realistic, we follow recently reported statistics (e.g., [4] and [37]) to prepare our largest dataset consisting 100,000 users, 10,000 tenants, and 500 domains. For verification, we use the V&V tool, Sugar V2.2.1 [34]. 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 16 GB memory, and we repeat each experiment 1,000 times.

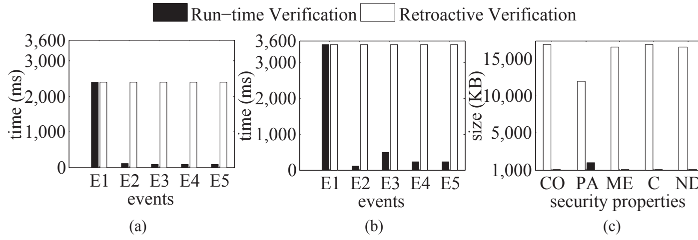
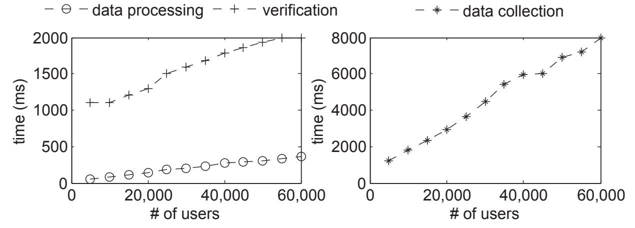
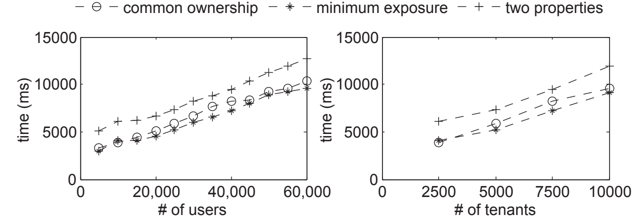


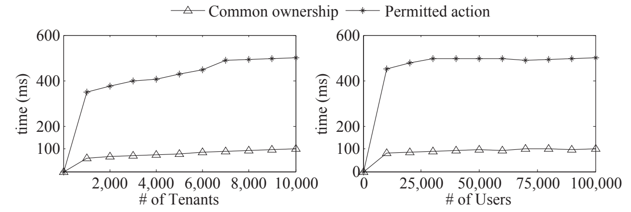
Fig. 7. Comparing the verification time required after each event for our system and the retroactive approach (e.g., [9]) for the (a) common ownership and (b) permitted action properties. Here, E1=initialization, E2=grant a role, E3=delete a role, E4=delete a user and E5=delete a tenant. The chart in (c) shows total size (in Kilo Bytes) of the data to be verified both for our approach and a naive approach for different properties (where CO: common ownership, PA: permitted action, ME: minimum exposure, C: cardinality, ND: no duplicate ID). The results are for our largest dataset.



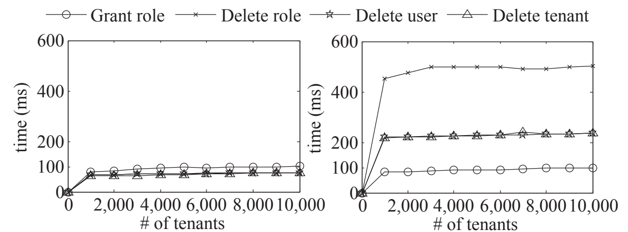
(a) Time required for each step during the initialization phase for the common ownership property while varying the number of users. Time for the data collection (right) is shown separately, as it is a one-time effort. In all cases, number of domains is 500 and number of tenants is 10,000.



(b) Total time required to perform the initialization phase for 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 in curves encompasses all three steps (collection, processing and verification). For the curve of two properties, data collection is performed one time.



(c) Total time required to perform the runtime phase of the common ownership and permitted action properties, by varying the number of tenants with fixed 30,000 users (left) and the number of users with fixed 5,000 tenants (right). In all cases, number of domains is 500.



(d) Time required to perform the runtime phase of the common ownership (left) and permitted action (right) properties for different events, by varying the number of tenants with 10 users per tenant. In all cases, number of domains is 500.

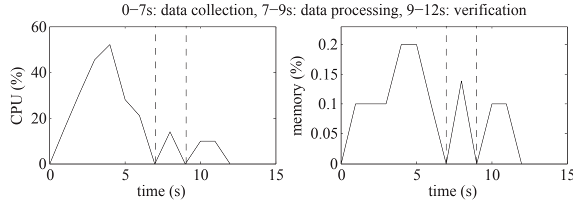
Fig. 8. Execution time for each step during initialization and runtime phases for different properties and different events using our framework

B. Results

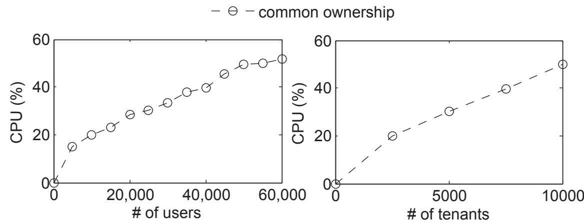
The objective of the first set of our experiments (see Fig. 7) is to demonstrate the time and memory efficiency of our solution, and to compare the performance with a retroactive auditing approach similar as in [9]. Firstly, Fig. 7a shows time in milliseconds required for our runtime verification framework for the common ownership property. Our runtime verification requires a relatively expensive (i.e., about 2.5 seconds) initialization phase, similar to that of the retroactive approach. Afterwards, our runtime approach takes less than 100 ms; whereas, the retroactive approach always takes 2.5 seconds. Secondly, Fig. 7b compares time in milliseconds required for verifying the permitted action property by our framework and a retroactive verification method. For this property, we obtain results of the same nature as the previous one i.e., requiring only a relatively expensive (i.e., about 3.5 seconds) initialization phase followed by runtime verification costing maximum 500 ms. For the permitted action property, after the *delete a role* event, a search for a certain role is performed; hence the verification time reaches the maximum value. Otherwise, verification time is within 100 ms for both properties. Finally, Fig. 7c depicts the comparison between memory requirement for both approaches while verifying different properties. The retroactive approach requires 12 MB to 17 MB space, as each time we have to load the whole verification data. Whereas, in the runtime approach, mostly we perform verification only on the changed data, therefore it takes maximum 1 MB memory.

Our second set of experiments (see Fig. 8) is to demonstrate the time efficiency of individual phases of our solution. Firstly, Fig. 8a shows time in milliseconds required for data collection, data processing and compliance verification during the initialization phase to verify the common ownership property for different cloud sizes (e.g., the number of users). The obtained results show that the verification execution time is less than 2 seconds for fairly large numbers of users. Knowing that this task is performed only once upon each request, we believe that this is an acceptable overhead for verifying a large setup. Fig. 8b shows the total time required for separately performing the initialization phase for 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 verifying more security properties would not lead to a significant increase in the execution time. Fig. 8c shows the total time required for separately performing the runtime phase for common ownership and permitted action properties for different cloud sizes. The obtained results support that the verification time for the permitted action (i.e., up to 500 ms) is more than that of the common ownership (i.e., up to 100 ms). Fig. 8d further depicts the effect of different events on the runtime phase for different security properties, while varying the number of tenants up to 10,000. As our runtime phase is an incremental approach and verifies mainly parameters of the events (as shown in Fig. 7c), the size of the cloud affects the

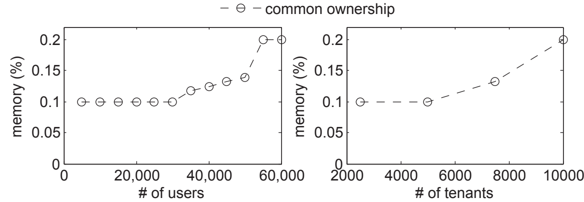
verification time very less.



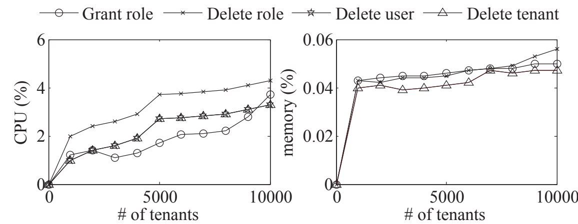
(a) CPU (left) and memory (right) usage for each step during the initialization phase over time with 60,000 users, 10,000 tenants and 500 domains for the common ownership property.



(b) Peak CPU usage to perform the initialization phase for the 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.



(c) Peak memory usage to perform the initialization phase for the 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.



(d) Peak CPU usage (left) and peak memory usage (right) to perform the runtime phase of the common ownership property for different events, by varying the number of tenants with 10 users per tenant. In all cases, number of domains is 500.

Fig. 9. CPU and memory usage for the initialization and runtime phases

Our third experiment (see Figures 9a (left), 9b and 9d (left)) measures the CPU usage (in %) during the initialization and runtime phases. The left chart in Fig. 9a 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 verification 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 verification 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 Fig. 9b, we measure the peak CPU usage (in %) consumed by different steps while verifying 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. Fig. 9d (left) shows that runtime phase expectedly requires negligible CPU (i.e., up to 4.7%) in comparison to the initialization phase.

Our final experiment (Figures 9a (right), 9c and 9d (right)) measures the memory usage during the initialization and runtime phases. The right chart in Fig. 9a 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%. Fig. 9c 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. Fig. 9d (right) depicts the memory usage by our runtime phase and further supports that the runtime phase deals with significantly smaller data set (as also shown in Fig. 7c).

Although we report results for a limited set of security properties, the use of formal methods for verifying these properties shows very promising results. Particularly, we show that the time required for our solution grows very slowly with the number of security properties. As seen in Fig. 8b, an additional security property adds only about 3 seconds to the initial effort. Therefore, we anticipate that verifying a large list of security properties would still be practical.

VI. Discussion

Adapting to Other Cloud Platforms. Our solution is designed to work with most popular cloud platforms (e.g., OpenStack [7], Amazon EC2 [16], Google GCP [18], Microsoft Azure [17]) with a minimal one-time effort. Once a mapping of the APIs from these platforms to the generic event types are provided, rest of the steps in our auditing system are platform-agnostic. Table VI enlists some examples of such mappings.

Generic Event Type	OpenStack [7]	Amazon EC2-VPC [16]	Google GCP [18]	Microsoft Azure [17]
create user	POST /v3/users	aws iam create-user	gcloud beta compute users create	az ad user create
delete user	DELETE /v3/users/{user_id}	aws iam delete-user –user-name	gcloud beta compute users delete	az ad user delete
assign role	/v3/users/{user_id} /roles/{role_id}	aws iam attach-role-policy	gcloud projects add-iam-policy-binding	az role assignment create
create role	POST /v3/roles	aws iam create-role	gcloud beta iam roles create	az role definition create
delete role	DELETE /v3/roles/{role_id}	aws iam delete-role	gcloud beta iam roles delete	az role definition delete

TABLE VI
MAPPING EVENT APIS OF DIFFERENT CLOUD PLATFORMS TO GENERIC EVENT TYPES.

Handling Extreme Situations. There might be some extreme cases where our solution may act differently. For instance, if the cloud logging system fails resulting from any disruption or failure in the cloud, then our auditing system will be affected. As in our threat model (in Section II-C), we assume that our solution relies on the correctness of the input data (including the logs) from the cloud. Any other failure or disruption in the cloud must be detected by our system. Also, if our system including the formal verification tool (e.g., Sugar) fails, till now there is no self-healing or self-recovery feature. Therefore, in this extreme case, the efficiency of the system

will be affected and a full (instead of incremental) verification will be required to recover from this failure.

The Rationale behind our Incremental Approach. The incremental verification of a given security property involves instantiating and solving the security property predicates for the affected elements in the supports of the involved relations (as stated in Section III-D). Therefore, any modification to the system data resulted from cloud events (e.g., grant role, delete role, etc.) would not directly change the security property expression itself although the corresponding support may need to be changed. For example, if a role is granted, the only change is that the relationships involving the entity role in the model would include a new element in their supports.

VII. RELATED WORK

Table VII compares existing related works for the cloud. Firstly, the existing approaches are categorized into: retroactive, intercept-and-check and proactive. Secondly, these works mainly cover three major levels: user, network and virtual infrastructure. Thirdly, we identify several features to differentiate our work from others. The *no-future-plan* feature is checked when a proactive or intercept-and-check approach does not require any future change plan; for retroactive approaches this feature is not applicable (N/A). The *first-order-logic* feature is checked when a work can verify any security property that is expressed in first order logic. We also identify the works that support verification on RBAC, ABAC and SSO. Finally, the most works are specifically designed for a particular cloud platform. In summary, our work differs from the existing works as follows. First, this work offers an intercept-and-check approach to audit the user-level at runtime. Second, only our work supports security properties related to RBAC, ABAC and SSO. Thirdly, our approach requires no future change plan for the verification process. Finally, this work explains how it can be adapted to other cloud platforms.

	Approaches			Coverage			Features					Platforms			
Proposals	Retroactive	Intercept-and-check	Proactive	User-level	Network-level	Virtual Inf.	No-future-plan	First-order-logic	Verifying RBAC	Verifying ABAC	Verifying SSO	Supporting OpenStack	Supporting Azure	Supporting VMware	Adaptable to others
CloudRadar [38]	-	●	-	-	-	●	N/A	-	-	-	-	-	-	●	-
Weatherman [11]	-	●	●	-	-	●	-	-	-	-	-	-	-	●	-
Majumdar et al. [9]	●	-	-	●	-	-	N/A	●	●	-	-	●	-	-	-
Madi et al. [8]	●	-	-	-	●	●	N/A	●	-	-	-	●	-	-	-
Majumdar et al. [12]	-	●	●	●	●	●	●	-	●	-	-	●	-	-	-
Doelitzscher et al. [39]	●	-	-	-	-	●	N/A	-	-	-	-	●	-	-	-
Ullah et al. [40]	●	-	-	-	-	●	N/A	-	-	-	-	●	-	-	-
Congress [10]	●	●	●	●	●	●	-	-	-	-	-	●	-	-	-
SecGuru [41]	●	-	-	-	●	-	N/A	●	-	-	-	-	●	-	-
QRadar [42]	●	-	-	●	●	●	N/A	-	-	-	●	-	-	●	-
This work	-	●	-	●	-	-	●	●	●	●	●	●	-	-	●

TABLE VII
COMPARING DIFFERENT EXISTING SOLUTIONS. THE SYMBOL (●) INDICATES THAT THE PROPOSAL OFFERS THE CORRESPONDING FEATURE.

Verifying security compliance in the cloud has recently been explored. For instance, in [8], [9], formal auditing approaches are proposed for retroactive security compliance checking in the cloud. The works in [40], [43] also support retroactive auditing. Unlike our proposal, those approaches can detect violations only after they occur, which may expose the system to high risks. There are several works (e.g., [44], [45], [46]) offering runtime security check in the cloud. VeriFlow [44] and NetPlumber [45] monitor network events and check network policies at runtime to capture bugs before or as soon as they occur. Designing cloud monitoring services based on security service-level agreements have been discussed in [46]. There are several other works that target auditing data location and storage in the cloud (e.g., [47], [48], [49], [50]) and others target infrastructure change auditing (e.g., [40], [39]).

Several existing efforts (e.g., [51], [52], [53], [54]) verify access control policies at the design time. In most of these works, cloud-related user-level security properties are not considered. There are some efforts (e.g., [55], [15], [56], [57]) towards proposing multi-domain/tenant access control models. Gouglidis et al. [31] utilize model-checking to verify custom extensions of RBAC with multi-domain against security properties. Lu et al. [58] use set theory to formalize policy conflicts in the context of inter-operation in the multi-domain environment. In contrast to those 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 at runtime.

There are few other works (e.g., [12], [11], [10]) offering runtime security policy checking in the cloud. Our previous work in [12] proactively verifies security compliance very efficiently through pre-computation by utilizing dependency models. However, there are several properties (e.g., minimum exposure, proper constraint checking, session time-out) which cannot be captured through the dependency models. On the other hand, this paper is capable of verifying a wider range of properties. Weatherman [11] aims at mitigating misconfigurations and enforcing security policies in a virtualized infrastructure. However, expensive computations after each critical event causes significant delay. Our work overcomes this limitation by using incremental verification. Congress [10] is an OpenStack project offering similar features as Weatherman. Several industrial efforts include solutions to support auditing in specific cloud environments. For instance, SecGuru [41] audits Microsoft Azure datacenter using the SMT solver Z3. IBM provides a monitoring tool integrated with QRadar [42], to collect and analyze events in the cloud. Amazon offers web API logs and metric data to their AWS clients by AWS CloudWatch & CloudTrail [59] to facilitate auditing. Although those efforts may assist auditing tasks, we support a wider set of user-level security properties.

VIII. CONCLUSION

Despite existing efforts, runtime security auditing in cloud still faces many challenges. In this paper, we proposed a runtime security auditing framework for the cloud with special focus on the user-level including different access control and authentication mechanisms e.g., RBAC, ABAC, SSO, and we implemented and evaluated the framework based on OpenStack, a popular cloud management system. Our experimental results showed that our incremental approach in runtime verification reduces the response time to a practical level. (e.g., less than 500 milliseconds to verify 100,000 users). This response time is satisfactory when the management operations are manually done by the administrators. The current approach would be insufficient to provide the same response time in

the case of batch execution for management operations, when these operations are executed in short intervals and if the subsequent operations impact the same property. As future work, to address this use case, we consider maintaining a scheduler including an event queue with different threads for different tasks in order to verify properties concurrently and therefore reduce the response time in this case.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their comments. We appreciate Amir Alimohammadifar's suggestions about the SSO properties. This material is based upon work partially supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under CRD Grant XC0907.

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] E. Aguiar, Y. Zhang, and M. Blanton, "An overview of issues and recent developments in cloud computing and storage security," in *High Performance Cloud Auditing and Applications*, 2014.
[3] Cloud Security Alliance, "CSA STAR program and open certification framework in 2016 and beyond," 2016, available at: https://cloudsecurityalliance.org.
[4] OpenStack, "OpenStack user survey," 2016, available at: https://www.openstack.org.
[5] Cloud Security Alliance, "Cloud control matrix CCM v3.0.1," 2014, available at: https://cloudsecurityalliance.org/research/ccm/.
[6] ISO Std IEC, "ISO 27017," *Information technology- Security techniques (DRAFT)*, 2012.
[7] OpenStack, "OpenStack open source cloud computing software," 2015, available at: http://www.openstack.org.
[8] T. Madi, S. Majumdar, Y. Wang, Y. Jarraya, M. Pourzandi, and L. Wang, "Auditing security compliance of the virtualized infrastructure in the cloud: Application to OpenStack," in *CODASPY*, 2016.
[9] 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 *CloudCom*, 2015.
[10] OpenStack, "OpenStack Congress," 2015, available at: https://wiki.openstack.org/wiki/Congress.
[11] S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim, "Proactive security analysis of changes in virtualized infrastructure," in *ACSAC*, 2015.
[12] S. Majumdar, Y. Jarraya, T. Madi, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "Proactive verification of security compliance for clouds through pre-computation: Application to OpenStack," in *ESORICS*, 2016.
[13] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," *ACM TISSEC*, vol. 4, no. 3, 2001.
[14] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to attribute based access control (ABAC) definition and considerations," *NIST SP*, vol. 800, 2014.
[15] B. Tang and R. Sandhu, "Extending OpenStack access control with domain trust," in *Network and System Security*, 2014, pp. 54–69.
[16] Amazon, "Amazon virtual private cloud," available at: https://aws.amazon.com/vpc.
[17] Microsoft, "Microsoft Azure virtual network," available at: https://azure.microsoft.com.
[18] Google, "Google cloud platform," available at: https://cloud.google.com.
[19] VMware, "VMware vCloud Director," available at: https://www.vmware.com.
[20] X. Jin, "Attribute Based Access Control Model," available at: https://blueprints.launchpad.net/keystone/%2Bspec/attribute-based-access-control.
[21] R. Sandhu, "The authorization leap from rights to attributes: maturation or chaos?" in *SACMAT*, 2012.
[22] N. Pustchi and R. Sandhu, "MT-ABAC: A multi-tenant attribute-based access control model with tenant trust," in *NSS*, 2015.
[23] OpenID Foundation, "OpenID: the internet identity layer," 2016, available at: http://openid.net.
[24] OASIS, "Security assertion markup language (SAML)," 2016, available at: http://www.oasis-open.org/committees/security.
[25] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed single-sign-on web services," in *IEEE S&P*, 2012.
[26] T. Groß, "Security analysis of the SAML single sign-on browser/artifact profile," in *ACSAC*, 2003.
[27] H.-K. Oh and S.-H. Jin, "The security limitations of SSO in OpenID," in *10th International Conference on Advanced Communication Technology*, 2008.
[28] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra, "Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google apps," in *ACM FMSE*, 2008.
[29] ISO Std IEC, "ISO 27002:2005," *Information Technology-Security Techniques*, 2005.
[30] NIST, SP, "NIST SP 800-53," *Recommended Security Controls for Federal Information Systems*, 2003.
[31] 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).
[32] W. Jansen, "Inheritance properties of role hierarchies," in *NISSC*, 1998.
[33] X. Jin, "Attribute based access control and implementation in infrastructure as a service cloud," Ph.D. dissertation, The University of Texas at San Antonio, 2014.
[34] N. Tamura and M. Banbara, "Sugar: A CSP to SAT translator based on order encoding," *Proceedings of the Second International CSP Solver Competition*, 2008.
[35] M. Ben-Ari, *Mathematical logic for computer science*. Springer Science & Business Media, 2012.
[36] OpenStack, "OpenStack audit middleware," 2016, available at: http://docs.openstack.org/developer/keystonemiddleware.
[37] getcloudify..org, "OpenStack in numbers - the real stats," 2014, available at: http://getcloudify.org.
[38] S. Bleikertz, C. Vogel, and T. Groß, "Cloud radar: near real-time detection of security failures in dynamic virtualized infrastructures," in *ACSAC*, 2014.
[39] F. Doelitzscher, C. Fischer, D. Moskal, C. Reich, M. Knahl, and N. Clarke, "Validating cloud infrastructure changes by cloud audits," in *SERVICES*, 2012.
[40] K. Ullah, A. Ahmed, and J. Ylitalo, "Towards building an automated security compliance tool for the cloud," in *TrustCom*, 2013.
[41] N. Bjørner and K. Jayaraman, "Checking cloud contracts in Microsoft Azure," in *Distributed Computing and Internet Technology*, 2015.
[42] IBM, "Safeguarding the cloud with IBM security solutions," http://www.ibm.com, IBM Corporation, Tech. Rep., 2013.
[43] F. Doelitzscher, "Security audit compliance for cloud computing," Ph.D. dissertation, Plymouth University, 2014.
[44] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: verifying network-wide invariants in real time," in *NSDI*, 2013.
[45] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *NSDI*, 2013.
[46] D. Petcu and C. Craciun, "Towards a security SLA-based cloud monitoring service," in *CLOSER*, 2014.
[47] C. Wang, S. S. Chow, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for secure cloud storage," *IEEE TC*, 2013.
[48] 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*, 2013.
[49] Z. Ismail, C. Kiennert, J. Leneutre, and L. Chen, "Auditing a cloud provider's compliance with data backup requirements: A game theoretical analysis," *IEEE TIFS*, 2016.
[50] Y. Wang, Q. Wu, B. Qin, W. Shi, R. H. Deng, and J. Hu, "Identity-based data outsourcing with comprehensive auditing in clouds," *IEEE TIFS*, 2017.
[51] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *ICSE*, 2005.
[52] G.-J. Ahn, H. Hu, J. Lee, and Y. Meng, "Representing and reasoning about web access control policies," in *COMPSAC*, 2010.
[53] K. Arkoudas, R. Chadha, and J. Chiang, "Sophisticated access control via SMT and logical frameworks," *ACM TISSEC*, 2014.
[54] S.-J. Horng, S.-F. Tzeng, Y. Pan, P. Fan, X. Wang, T. Li, and M. K. Khan, "b-SPECS+: Batch verification for secure pseudonymous authentication in VANET," *IEEE TIFS*, 2013.

[55] 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.
[56] Q. Alam, S. U. Malik, A. Akhunzada, K.-K. R. Choo, S. Tabbasum, and M. Alam, "A cross tenant access control (CTAC) model for cloud computing: Formal specification and verification," *IEEE TIFS*, 2016.
[57] A. Gouglidis and I. Mavridis, "domRBAC: An access control model for modern collaborative systems," *Computers & Security*, 2012.
[58] 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.
[59] Amazon Web Services, "Security at scale: Logging in AWS," http://aws.amazon.com, Amazon, Tech. Rep., 2013.

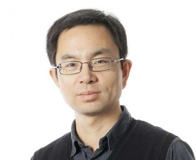
Suryadipta Majumdar Suryadipta Majumdar is a Ph.D. candidate at the Concordia Institute for Information Systems Engineering, Concordia University, Canada. He finished his M.A.Sc. in Information Systems Security from Concordia University and his B.Sc. in Computer Science and Engineering from BUET, Bangladesh. His research interests include cloud computing security, privacy, key management and authentication.

Taous Madi Taous Madi is currently working toward the Ph.D. degree in information and systems engineering at the Concordia Institute for Information Systems Engineering, Concordia University, Montreal, QC, Canada. She is currently a Research Assistant with Concordia University. She received the B.S. degree in engineering and the Magister degree from the Universit des Sciences et de la Technologie Houari Boumediene, Algiers. Her research interests include cloud computing, mobile computing, formal verification, software-defined networking, and security.

Yushun Wang Yushun Wang is an MASc student in Information System Security, Concordia University from 2014. Previously, he worked as a customer network support engineer, Ericsson(China) for 12 years.

Yosr Jarraya Yosr Jarraya is currently a researcher in security at Ericsson. Before that, she had a two-year MITACS postdoctoral fellowship with the company. She was previously Research Associate and Postdoctoral Fellow at Concordia University, Montreal. She received a Ph.D. in Electrical and Computer Engineering from Concordia University. In the past six years she has produced more than 25 research papers on topics including SDN, security, software and the cloud.

Makan Pourzandi Makan Pourzandi is a researcher at Ericsson, Canada. He received his Ph.D. degree in Computer Science from University of Lyon I, France and M.Sc. in parallel computing from cole Normale Suprieure de Lyon, France. He has more than 15 years of experience in security for Telecom systems, cloud computing, distributed systems security and software security. He is the inventor of over 28 patents granted or pending. He has published more than 50 research papers in peer-reviewed scientific journals and conferences.



Lingyu Wang Lingyu Wang is a professor in the Concordia Institute for Information Systems Engineering (CIISE) at Concordia University, Montreal, Quebec, Canada. He received his Ph.D. degree in Information Technology from George Mason University. His research interests include data privacy, network security, security metrics, cloud computing security, and malware analysis. He has co-authored over 100 refereed publications on security and privacy.

Mourad Debbabi Mourad Debbabi is a Full Professor at the Concordia Institute for Information Systems Engineering and Associate Dean Research and Graduate Studies at the Faculty of Engineering and Computer Science. He holds the NSERC/Hydro-Qubec Thales Senior Industrial Research Chair in Smart Grid Security and the Concordia Research Chair Tier I in Information Systems Security. He is also the President of the National Cyber Forensics and Training Alliance (NCFTA) Canada. He is also a member of CATAAlliance's Cybercrime Advisory Council and a member of the Advisory Board of the Canadian Police College. He is the founder and one of the leaders of the Security Research Centre at Concordia University. In the past, he was the Specification Lead of four Standard JAIN (Java Intelligent Networks) Java Specification Requests dedicated to the elaboration of standard specifications for presence and instant messaging. Dr. Debbabi holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay, University, France. He published 3 books and more than 260 peer-reviewed research articles in international journals and conferences on cyber security, cyber forensics, privacy, cryptographic protocols, threat intelligence generation, malware analysis, reverse engineering, specification and verification of safety-critical systems, smart grid, programming languages and type theory. He supervised to successful completion 26 Ph.D. students and more than 65 Master students. He served as a Senior Scientist at the Panasonic Information and Network Technologies Laboratory, Princeton, New Jersey, USA; Associate Professor at the Computer Science Department of Laval University, Canada; Senior Scientist at General Electric Research Center, New York, USA; Research Associate at the Computer Science Department of Stanford University, California, USA; and Permanent Researcher at the Bull Corporate Research Center, Paris, France.