Proactive Verification of Security Compliance for Clouds through Pre-Computation: Application to OpenStack

Suryadipta Majumdar[1], Yosr Jarraya[2], Taous Madi[1], Amir Alimohammadifar[1], Makan Pourzandi[2], Lingyu Wang[1], and Mourad Debbabi[1]

[1] CIISE, Concordia University, Montreal, QC, Canada
{su_majum,t_madi,ami_alim,wang,debbabi}@encs.concordia.ca
[2] Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada
{yosr.jarraya,makan.pourzandi}@ericsson.com

Abstract. The verification of security compliance with respect to security standards and policies is desirable to both cloud providers and users. However, the sheer size of a cloud implies a major challenge to be scalability and in particular response time. Most existing approaches are either after the fact or incur prohibitive delay in processing user requests. In this paper, we propose a scalable approach that can reduce the response time of online security compliance verification in large clouds to a practical level. The main idea is to start preparing for the costly verification proactively, as soon as the system is a few steps ahead of potential operations causing violations. We present detailed models and algorithms, and report real-life experiences and challenges faced while implementing our solution in OpenStack. We also conduct experiments whose results confirm the efficiency and scalability of our approach.

Keywords: Proactive compliance verification, cloud security, auditing, OpenStack.

1 Introduction

The widespread adoption of cloud computing as the replacement of traditional IT solutions is still being hindered by various security concerns [5]. In particular, the multi-tenant and self-service nature of clouds usually implies significant operational complexity, which may prepare the floor for misconfigurations and vulnerabilities leading to violations of security compliance. Therefore, the security compliance verification w.r.t. security standards, policies, and properties, is desirable to both cloud providers and users. Evidently, the Cloud Security Alliance (CSA) 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) [7].

However, the sheer size of clouds (e.g., a *decent-size* cloud is said to have around 1,000 tenants and 100,000 users [25]) implies one of the main challenges in verifying security compliance, specifically the scalability and response time. To that end, existing approaches can be roughly divided into three categories (a more detailed review of related work will be given in Section 6). First, the *retroactive approaches* (e.g., [18,

19]) catch compliance violations after the fact, which means they cannot prevent security breaches from propagating or causing potentially irreversible damages (e.g., leaks of confidential information or denial of service). Second, the *intercept-and-check* approaches (e.g., [3, 23]) verify the compliance of each user request before either granting or denying it, which may lead to a substantial delay to users' requests, as will be further illustrated later in this section. Third, the *proactive approaches* in [3, 23] verify user requests in advance, which, however, assume the sequence of such requests is known beforehand.

In this paper, we propose a scalable approach for proactive verification of security compliance in large clouds, by avoiding the limitations of the last two approaches mentioned above (i.e., *intercept-and-check* and *proactive*). Specifically, unlike both existing approaches, we start to prepare for the costly verification proactively, as soon as the system is N-step (N is an integer) ahead of the operations causing compliance violations (namely, *critical operations*), such that the actual verification of critical operations is reduced to a simple search in a pre-computed table (namely, *watchlist*), causing negligible delay.

To illustrate the idea, Fig. 1 compares how user requests are processed under a typical *intercept-and-check* approach and under our solution, respectively. In the upper timeline, an intercept-and-check approach intercepts and then verifies the *update port* user request against the desired security property[1]. The state of the art, as reported in [3], would take over four minutes for checking the current cloud (medium-size) state to determine whether the request should be granted or denied. Extrapolating such a result to a larger cloud would result in hours of delay, which is clearly infeasible. On the other hand, as depicted in the lower timeline, our approach works very differently. It proactively conducts a set of pre-computations distributed among N-steps ahead of the actual occurrence of the critical operation (*update port*). These pre-computations incrementally prepare the needed information for efficiently verifying the critical operation later on, and consequently the actual verification only takes six milliseconds, instead of four minutes [3], as shown in the timeline.

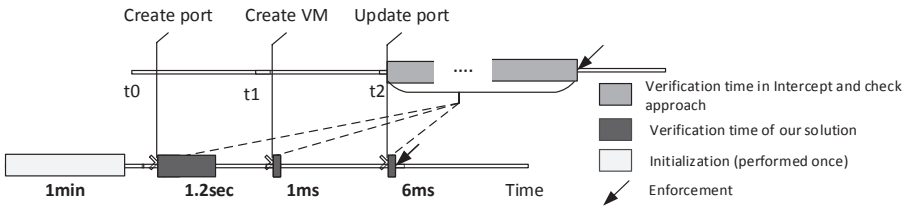

Fig. 1: Comparison of the execution time of our solution with the *intercept-and-check*.

The main contributions of our paper are as follows:

– To the best of our knowledge, the proposed proactive verification approach is the first solution that can reduce the response-time of online security compliance ver-

[1] Here we consider the "no bypass" security property for the anti-spoofing mechanisms in the cloud, which can be violated by real world vulnerabilities (e.g., OpenStack vulnerability [22]).

ification in large clouds to a practical level (e.g., the response time is about 8.5ms for a large cloud with 100,000 virtual ports).
- We devise dependency models to capture the relationships between various management operations and related security properties, for both the identity and access management service and the virtualized infrastructure in clouds, which serve as the foundation of our approach.
- We provide detailed methodology and algorithms. We also report real-life experiences and challenges faced while implementing our solution in OpenStack [24].
- We conduct experiments to evaluate the performance of our solution, and the results confirm the efficiency and scalability to be practical for large clouds.
- Finally, our solution goes inline with the continuous monitoring-based certification, which is the most demanding level specified by CSA [7].

The paper is organized as follows. Section 2 describes the threat model followed by our running example and the dependency models for the virtualized infrastructure, and for the access control management service. Section 3 details our methodology. Section 4 provides the implementation details and experimental results. Section 5 discusses different aspects of our approach. Section 6 summarizes related works and compares them with our approach. Section 7 concludes the paper providing future research directions.

2 Models

Here, we give the threat model and present the dependency models.

2.1 Threat Model

We assume that the cloud infrastructure management systems a) may be trusted for the integrity of the API calls, event notifications, and database records (existing techniques on trusted computing may be applied to establish a chain of trust from TPM chips embedded inside the cloud hardware, e.g., [1]), and b) may have implementation flaws, misconfigurations and vulnerabilities that can be potentially exploited by malicious entities to violate security properties specified by the cloud tenants. The cloud users including cloud operators and agents (on behalf of a human) may be malicious.

Though our 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. We assume that, before our proactive approach is launched, an initial auditing is performed and potential violations are resolved. However, if our solution is added from the commencement of a cloud, obviously no prior security verification is required. This work focuses on attacks directed through the cloud management interfaces and more specifically cloud management operations (e.g., create/delete/update tenant, user, VM, etc.). Any violation bypassing the cloud management interface is beyond the scope of this work. To make our discussions more concrete, the following shows an example of in-scope threats based on a real vulnerability.

Running Example. Real world vulnerabilities such as the one in OpenStack [22][1], can be exploited to bypass anti-spoofing mechanisms. These mechanisms are implemented

[1] OpenStack [24] is an open-source cloud infrastructure management platform.

in OpenStack using firewall rules enforcing tenants' layer 3 network isolation. Fig. 2 shows the attack scenario to exploit this vulnerability. The exploit consists in changing the device owner (step 3 in Fig. 2) of an instance's port to a string starting with the word `network`, right after the instance is created (steps 1 & 2) and just before security group gets attached to it (race condition). As a result, the firewall rules of the compute node are not applied to that port, since it is treated as a network owned port. Consequently, a malicious tenant can launch IP, MAC, and DHCP spoofing attacks (step 4).

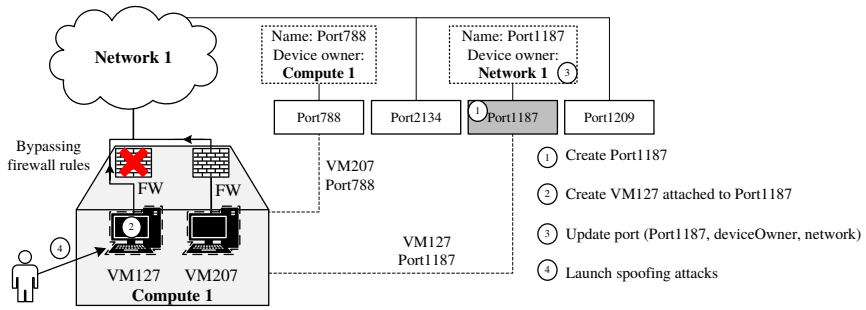

Fig. 2: An exploit of a vulnerability in OpenStack [22], leading to bypassing the anti-spoofing mechanism.

2.2 Dependency Models

Fig. 3 illustrates the two dependency models that we derive for an OpenStack-managed cloud covering virtual infrastructure (Fig. 3(a)) and user access control (Fig. 3(b)). Each dependency model can be used for proactively auditing multiple security properties. We validate these dependency models based on extensive study of OpenStack APIs from different related OpenStack services (e.g., Neutron, Nova, and Keystone) and Open vSwitch. For the user access control model, we are inspired by the OSAC model by Tang et al. [34]. To better explain the usefulness of these models, we start by providing an example on how the cloud infrastructure dependency model (see Fig. 3(a)) allows us to relate actual management operations/events happening in the cloud to the "no bypass" security property presented in Section 2.1.

Example 1 *According to the attack scenario presented in Fig. 2, the critical management operation that leads to the violation of the "no bypass" security property is* `update port`*. The model in Fig. 3(a) includes* `port` *(*`vertex 15`*) and* `VM` *(*`vertex 17`*). The* `vertex 16` *is a specific vertex grouping a port and a subnet pair. The* `update port` *operation is related to the entity* `port` *(*`vertex 15` *in Fig. 3(a)). As it can be seen in Fig. 3(a),* `update port` *depends on other operations such as* `create port` *(*`edge (12,15)`*) and* `create VM` *(*`edge 16, 17`*). More precisely, create VM attaches a port (*`vertex 15`*) on a subnet (*`vertex 14`*) to a VM (*`vertex 17`*).*

As the `create port` *and* `create VM` *operations are closely related to the actual critical operation (*`update port`*), our model captures this dependency relationship and aids to avoid the security violation by starting preparation from the* `create port` *operation. Furthermore, these operations in turn depend on the existence/creation of a*

subnet, a network and a tenant. This induces a chain of dependencies between a set of events that could be related to this security property.

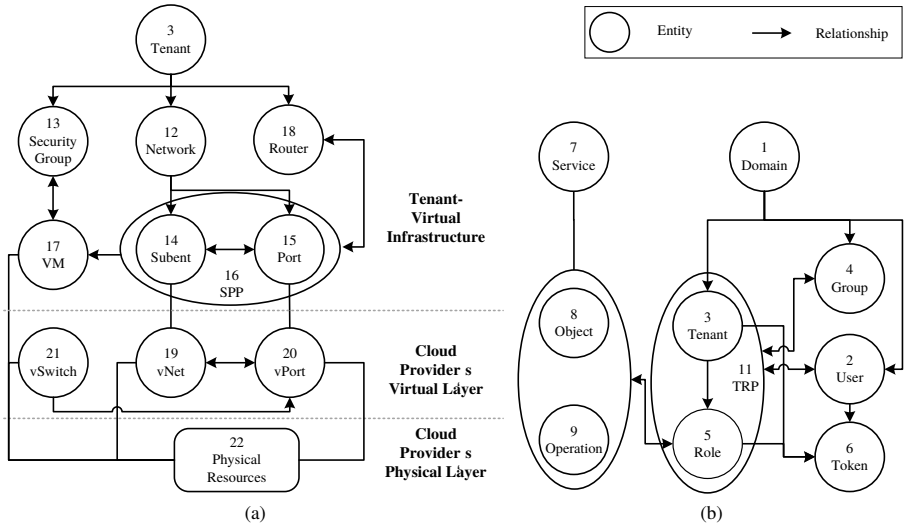

Fig. 3: Dependency models: (a) cloud infrastructure and (b) access control management.

Formally, the dependency model is a graph $G = (V, E)$, where vertices V_i are individual cloud entities (e.g., user, role, tenant, port, VM, etc.) or groups of entities (e.g., (port, subnet) pair) and edges E_{ij} are dependency relationships between connected vertices. These relationships are activated by events/operations in the cloud (e.g., create/delete port, attach VM to a (port, subnet) pair, etc.). We use edges' attributes to store information on which security property are associated with the events that are related to the edge and on the type of this event per property. We define four kinds of relationships. We use different types of edges (unidirectional, bidirectional, or non-directional) to differentiate the following relationships based on their semantics.

- *Precedence* relation, represented by a unidirectional edge, such that $E_{ij} = (V_i, V_j)$ denotes that the entity V_i must exist before being able to create entity V_j within V_i.
- *Association* relation, represented by a bidirectional edge, such that $E_{ij} = (V_i, V_j)$ denotes that entities V_i and V_j should both exist (i.e., created) to be able to make any association between them.
- *Mapping* relation, represented by a non-directional edge, such that $E_{ij} = \{V_i, V_j\}$ denotes a correspondence relationship between entities V_i and V_j existing in different layers in the cloud.
- *Reflexive* relation (not represented in the graph), representing a relation from a node to itself such as updating attributes of the node.

We leverage the knowledge captured by these dependencies to appropriately identify the intercepted events, relate them to the security property, identify their roles in

the context of proactive compliance verification, and determine the distance to a critical state. More details are provided in Section 3. It is worth noting that these models are static and do not depend on the execution context of the cloud. They consist of a relatively small set of entities and relationships. For example, for Neutron, Nova, and Keystone services, we enumerated only 86 different entities and about 400 events that are relevant to configuration changes and management.

Table 1 enlists an excerpt of the security properties supported by the cloud infrastructure dependency model. Table 3 (in Appendix A) enlists an excerpt of the security properties supported by the access control dependency model. Here, we categorize events mainly into two types: critical event (CE) and watchlist event (WE). A CE (e.g., `update port`) potentially leads to the violation of the associated property. A WE corresponds to an event that impacts the content of the watchlist associated with the security property (e.g., `create port` and `create VM`). The third type of event is the trigger event (TE), which is neither critical nor watchlist-related, however is useful to determine the distance to a critical state. Note that an event may have multiple types considering different security properties. For example, `create VM` is a WE event for the *no bypass* property, but it is of type CE for the *no co-residency* property.

Property	Critical Event (CE)	Watchlist Event (WE)	Watchlist per tenant
No bypass [6]	update port (15,15)	create VM (16,17) create port (12,15)	Ports except VM ports
Port consistency [13, 6]	create vPort (21,20)	create port (12,15)	ports at tenant layer
No abuse of resources [6]	create VM (16,17), create vNet (14,19)	create VM (16,17), create vNet (14,19)	Counters for VM/vNet
		delete VM (16,17), delete vNet (14,19)	
Common port ownership [6]	attach port to a router (16,18)	create router (3,18)	router-tenant pair
Port isolation [13, 6]	add vPort to vNet (19,20)	create vNet (14,19)	vNets in a subnet
No co-residency[1] [13, 6]	create VM (16,17), migrate VM (17,22)	create VM (16,17)	Hosts with no conflicting VMs
		migrate VM (17,22)	

Table 1: An excerpt of the security properties supported by the cloud infrastructure model with their corresponding critical and watchlist events, and the watchlist contents.

3 Proactive Verification of Security Compliance (PVSC)

In this section, we detail our solution to proactively verify security violation and enforce compliance in the cloud.

3.1 Overview

We devise a novel approach, namely Proactive Verification of Security Compliance (*PVSC*), that proactively conducts a set of pre-computations distributed along *N-step* ahead of the occurrence of a critical operation, where *N* is a parameter tailored for each considered security property and defined as the expected minimal distance to a critical state, which corresponds to the minimal number of operations from the current state. Note that this distance is called minimal as operations related to security properties may be interleaved with unrelated operations. In PVSC, the pre-computations incrementally prepare the needed conditions to preserve security compliance, and are stored in *watchlists*. These *watchlists* are used to detect violations of security properties when critical

operations are about to occur. To measure *N-step* and avoid state explosion, we leverage abstract dependency models described in Section 2.2.

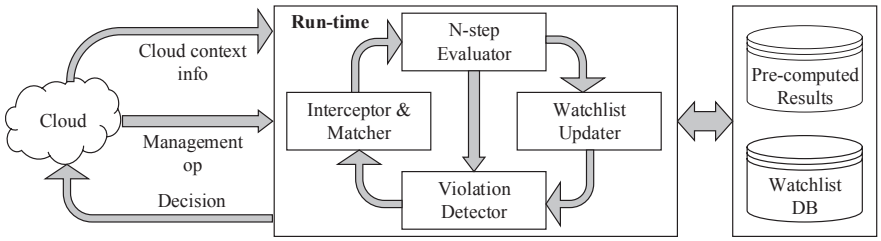

Fig. 4: An overview of our proactive solution.

An overview of this solution is depicted in Fig. 4. Initially, data from the dependency models and the initialized watchlists (generated with the watchlist contents mentioned in Table 1 for each security property) are pre-computed and stored in databases. PVSC uses this data to efficiently intercept and identify relevant operations (*Interceptor & Matcher*) but only blocks the critical ones. This data would be also used together with cloud context information to estimate the minimal distance towards the violation (*N-step Evaluator*). By identifying the type of intercepted operations and their impact on security properties, we elaborate different watchlists that are progressively updated (*Watchlist Updater*). These watchlists are consulted to evaluate the impact of a critical operation (*Violation Detector*). According to the later decision, the critical operation can be realized or blocked to preserve security compliance.

3.2 Methodology

PVSC consists of two phases: an initialization phase that is performed offline only once and a run-time verification phase, which is performed online but proactively.

Figure 5 depicts a detailed overview of PVSC. The initialization phase is mainly meant to pre-process data from different sources. Once the initialization phase is completed, the run-time detection phase serves at intercepting cloud operations, proactively finding out whether considered security properties are about to be compromised, determining the distance N (where N is the number of steps) towards critical states, and acting on re-enforcement towards preserving security compliance. In the following, we describe each phase and other related components of our approach in more details.

Initialization phase. This phase collects initial data from the cloud infrastructure and pre-processes the data in order to prepare the ground for the run-time phase. For each security property, all dependency models are to obtain: i) the involved cloud entities, ii) the related abstract-events with their types, and iii) all possible values of N. Pre-computing all needed information from the dependency models at the initialization phase avoids tracing the models at run-time, which fosters better efficiency. Following we describe different tables we leverage during the initialization phase.

- `Property-WL`: specifying the content of watchlist for each security property to aid watchlist initialization.

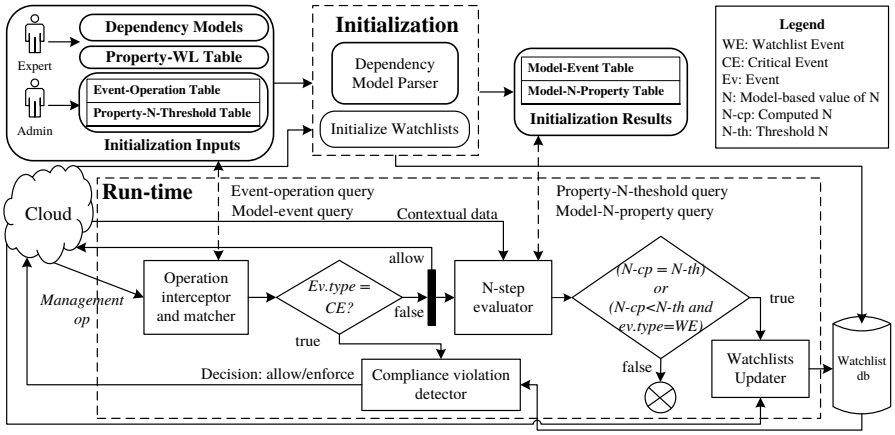

Fig. 5: A detailed overview of PVSC composed of an one-time initialization phase and a run-time verification phase.

- `Event-operation`: mapping events to different operations in different cloud environment to easily integrate different cloud implementations and used as input to the initialization phase.
- `Property-N-thresholds`: mapping the specified security properties and their associated thresholds (denoted as *N-th*), where thresholds are security property specific and inputs from the administrators.
- `Model-event`: relating each security property with the elements of the dependency models including the events with their types.
- `Model-N-property`: storing all possible values of *N* (denoted as *N-cp*) for each property.

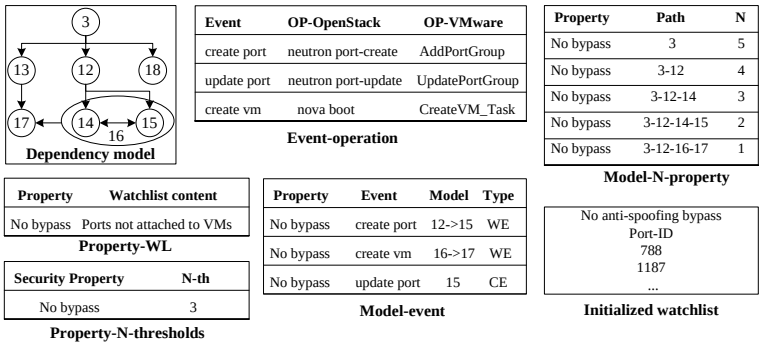

Fig. 6: The result of the initialization phase for the *no bypass* property.

Example 2 *Figure 6 shows the outcome of the initialization phase for the no bypass property. The traversal of the dependency model using the identified property and searching in the attributes of the corresponding edges allow identifying that, for the considered property,* `create port` *and* `create VM` *are of type WE and* `update`

`port` *is of type CE. This information is stored in the* `Model-event` *table. Other events of type TE such as* `create network`, `create subnet` *are not shown for brevity. The* `Event-operation` *table shows that the* `create port` *event corresponds to the* `neutron port-create` *operation in OpenStack. The minimal distance from the critical event at which our solution should react is (N-th* $= 3$*), as shown in the* `Property-N-thresholds` *table. The* `Model-N-property` *table stores all possible computed values of N taking into account the security property and the dependency model. Finally, the watchlist is initialized for the no bypass property based on data collected from the cloud. For each tenant, the watchlist is populated with the list of virtual ports that are not attached to a VM as in the* `Property-WL` *table.*

Precomputation of N consists in traversing the dependency graph for each security property from the edge corresponding to its critical event backward until reaching the root node of the graph, finding out all dependent events and entities and storing precomputed values of *N* for each possible configuration in the `Model-N-property` table. A configuration is an abstract state that allows to determine whether the entities that the security properties depend on, actually exist. The minimal distance to the critical event from the root node is the total number of events that the critical event depends on. This distance represents *N-max*, the maximal value of *N* from which we can apply our proactive approach for this property. The minimum value of *N* is 1 and it corresponds to the configuration where the next event to be observed is possibly the critical event.

Example 3 *For the no bypass property, the* `Model-N-property` *table stores five entries that cover all possible values of N and the associated configuration (See Fig. 7). For instance, if only a tenant already exists (vertex* 3*) without yet any network, subnet, ports, and VMs, we need to observe at least* 5 *events before being able to intercept the critical event* `update port`*. If we observe an event for the creation of network within this tenant (i.e., edge* $(3,12)$*) without yet any subnet, ports, and VMs, the minimal distance to see the* `update port` *event would be* $N = 4$*. The event preceding* `update port` *is the* `create VM` *(i.e., edge* $(16,17)$*) event, and the minimal distance is* 1.

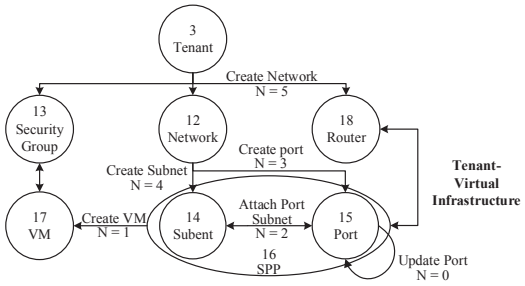

Fig. 7: A part of the cloud infrastructure dependency model annotated with all possible values of *N* that is relevant to the *no bypass* property.

Run-time computation. At run-time, our system intercepts all operation performed in the cloud, but only blocks those which are matched with the critical events. Matching operation with an event consists in querying the `Event-operation` table. For those

critical events, the corresponding operations are halted only during compliance verification, which consists in searching a set of values (e.g., values of the operation's parameters) in the corresponding watchlist. The verification decision is either to allow the operation to continue or apply the planned enforcement approach as specified by the administrator. Being able to intercept all operations, including the non-critical ones as soon as they are executed allows progressively updating watchlists, without impacting the performance of the cloud. At each matched non-critical events (WE or TE), our system updates the estimated current minimal distance to violation, namely $N\text{-}cp$, for the related property using the `N-step evaluator`. The latter collects contextual data from the cloud and uses it to query the table of precomputed values of N, namely `Model-N-property`. When the value of the estimated current distance to violation becomes equal to the threshold value, the watchlist corresponding to the security property is updated using data from the cloud regardless of the event type to ensure that its content is up-to-date, hence allowing appropriate detection. For $N\text{-}cp < N\text{-}th$, whenever a WE-type operation is encountered, the watchlist is directly updated using the values of the parameters of the intercepted operations.

The `N-step evaluator` evaluates at run-time the value of N, whenever a non-critical event concerning a given security property has been matched with the intercepted operation. First, the related context data is then gathered from the cloud to determine whether the status of the dependent entities. The evaluator uses this information and the security property in focus to make a specific query to the `Model-N-property` table, to collect the value of N corresponding to the current context.

Example 4 *Figure 8 illustrates the run-time workflow for the no bypass property assuming that a tenant, a network and a subnet already exist.*

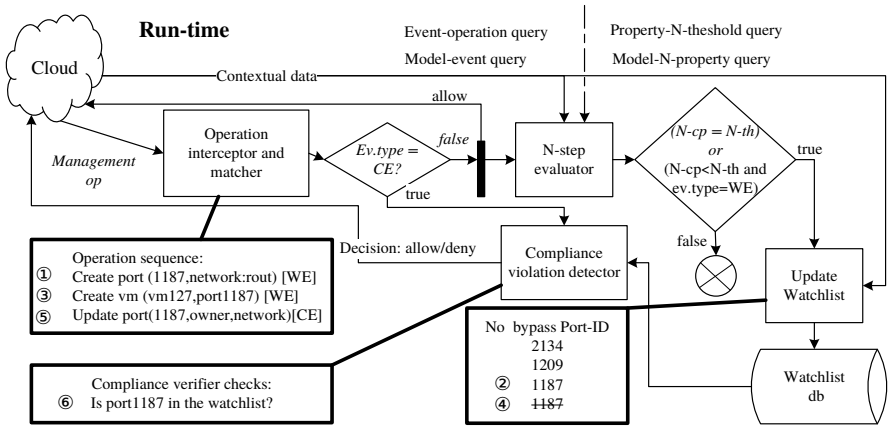

Fig. 8: An excerpt of runtime verification of the *no bypass* property.

To rectify the situation described in the running example, our solution incrementally builds a watchlist with ports that are not attached to VMs, and verifies the `update port` *operation with this watchlist. Firstly, we intercept the* `create port 1187` *operation, identifies the event type (which is WE), and measure the value of N (= 3) respectively from the* `Model-event` *and* `Property-N-threshold` *tables. Since the* `create`

`port` *event is a WE event for the no bypass property and evaluating N results in N-cp=N-th=*3*, we add* `port1187` *to the watchlist without blocking it. Secondly, we intercept* `create VM127 attached to port1187` *operation and measure N similarly. Then,* `port1187` *is removed from the watchlist, as it is now attached to* `VM127`*. Finally, after intercepting the* `update port(port1187, deviceOwner, network)` *operation and measuring N, we identify that this is a CE event. Therefore, we verify with the watchlist with blocking the operation, find that* `port1187` *is not in the watchlist, and hence PVSC recommends denial of this operation to preserve the no bypass property.*

4 Proof of concept in OpenStack

This section describes how we integrate PVSC into OpenStack and presents our experimental results.

4.1 Implementation

We detail the implementations of both the initialization and the runtime detection phases.

Background. OpenStack [24] 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 [8] for detailed statistics). Keystone [24] is the OpenStack identity service for authentication and authorization. Keystone implements the RBAC model [30]. Neutron [24] provides tenants with capabilities to build networking topologies through the exposed APIs. Nova [24] is the OpenStack project designed to provide on-demand access to compute resources, and relies on VMs.

Initialization phase. Firstly, we map all operations in OpenStack API [27] corresponding to the events that are relevant to the monitored security properties. During this phase, we store our pre-computated results and tenant-specific watchlists in a MySQL database; which allows us to efficiently query OpenStack cloud data, which is also stored in databases. Our Python scripts derive the association between the model provided in Fig. 3 and the security properties, and populate our database by adding the dependency information and the values of the precomputed N. Additionally, we capture the current state of the OpenStack cloud by collecting data from the Keystone, Neutron and Nova databases. Algorithm 1 in Appendix B also shows the steps of this phase.

Run-time phase. Firstly, the Interceptor module, which is implemented in Python, intercepts operations based on the existing intercepting methods (e.g., audit middleware [26]) supported in OpenStack. Events are primarily created via the notification system in OpenStack; Nova, Neutron, etc. emits notifications in a JSON format. Here, we leverage the audit middleware in Keystone, which was previously supported by pyCADF [4], to intercept Keystone, Neutron and Nova events by enabling the audit middleware and configuring filters. After intercepting an operation, its details (e.g., name and parameters) are processed by our Matcher module to determine the criticality of the current operation, and later forwarded to our MySQL stored procedures (e.g., N-step evaluator, Watchlist updater and Violation detector). The N-step evaluator measures the distance from any possible violation based on the Algorithm 2 in Appendix B. Based

on the outcome of both the Matcher and N-step evaluator modules, any of the following is processed: i) the Watchlist updater adds the parameter(s) of the current operation to the watchlist database, ii) the Violation detector searches the current values of the parameter(s) in the corresponding watchlist, and iii) forward the decision (e.g., allow or deny) to the cloud based on the enforcement options.

4.2 Experimental Results

In this section, we discuss time and memory requirements of PVSC.

Experiment settings. All experiments are conducted on the OpenStack setup inside a lab environment. Our OpenStack version is Liberty with Keystone API version v3 and Neutron API version v2. 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. Based on a recent survey [25] on OpenStack, we simulated an environment with maximum 100,000 users, 10,000 tenants, 500 domains, 100,000 VMs, 40,000 subnets, 20,000 routers and 100,000 ports. We conduct the experiment for 10 different datasets varying the most important factor and fixing others to the largest value, e.g., for the *no bypass* property, both the number of ports (from 10,000 to 100,000 with the gap of 10,000) and the number of tenants (from 1,000 to 10,000 with the gap of 1,000) are varied, as the content of the watchlist is tenant-specific and a list of ports. For the *common ownership*[1] property, the number of tenants is varied from 1,000 to 10,000 with the gap of 1,000 having 5 roles in each tenant. We repeat each experiment 100 times.

Results. The objective of the first set of experiment is to demonstrate the time efficiency of our proactive solution. Intercepting operations to identify the type of operation, which is the minimum time we need to black for all operations (CE and WE, and all others), is taking constant time (0.266 ms) (INT in Fig. 9(a)). Moreover, calculating N-step (NSE in Fig. 9(a)) completes in constant time (0.133 ms for the largest datasets) for the *no bypass* (NB) property, and in quasi constant time (varying from 0.773 ms to 0.794 ms) for the *common ownership* (CO) property. Violation detector blocks only critical operation for a maximum extra delay of 8.2 ms (VD in Fig. 9(b)) for the largest dataset. Fig. 10(a) shows the required execution time to pre-compute the watchlists for the *no bypass* and *common ownership* properties are 5,000 ms and 5,400 ms respectively, for our largest dataset. As expected, the watchlist pre-computation step,which involves access to the cloud databases, requires comparatively longer time. However, this step is performed only during the initialization phase. Any later update of the watchlist is performed incrementally, and takes few milliseconds. Fig. 10(a) depicts the execution time for the largest dataset (10,000 tenants and 100,000 ports), and shows that preparing watchlist is comparatively time consuming and beneficial to perform proactively, as we spend about 5,400 ms in preparing watchlist during initialization though the subsequent enforcement takes only 8 ms per critical operation call at run-time.

In the second part of the experiment, we measure the memory cost for the watchlists. Fig. 10(b) depicts that the memory requirement increases quasi linearly with the dataset size. We are able to restrict the watchlist size in few MBs by choosing the content of

[1] This property allows users to hold only the roles that are defined within their domains [13, 6].

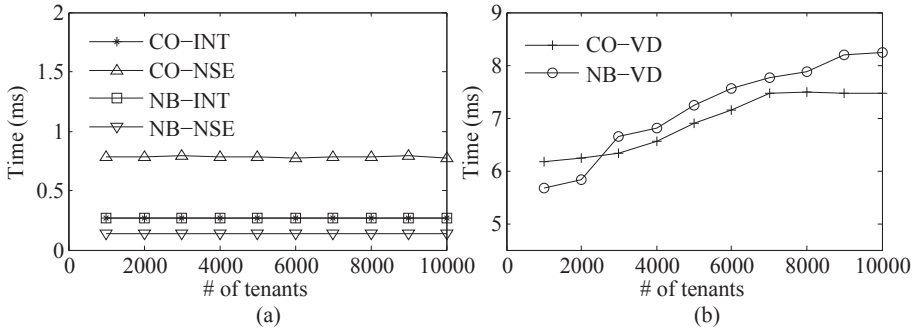

Fig. 9: Time duration (in ms) for different modules (INT: Interceptor, NSE: N-step evaluator, VD: Violation detector) of PVSC for the *common ownership* (CO) and *no bypass* (NB) properties by varying the number of tenants. The number of ports is also varied from 10,000 to 100,000, and each tenant contains 5 roles. Time required for the steps: (a) intercepting operations, evaluating N-step, and (b) detecting violations.

the watchlist carefully. Therefore, we show that our approach improves the execution time without excessive memory costs. We store roles and corresponding tenants for the common ownership property, and only ports for the no bypass property.

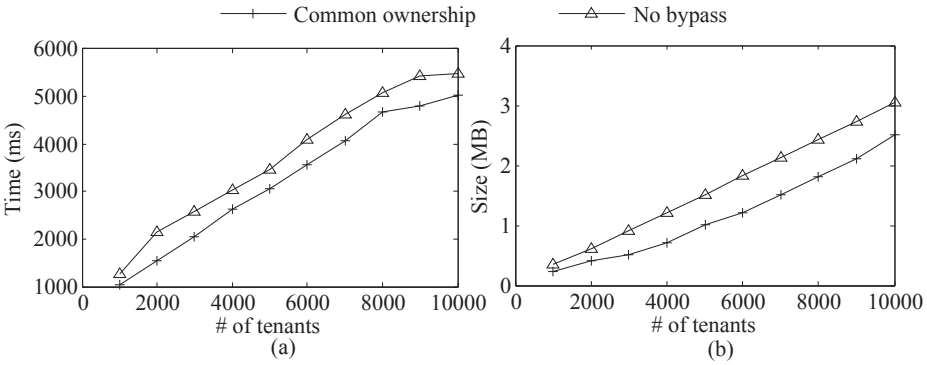

Fig. 10: (a) Time required (in ms) for preparing watchlist for different properties varying the number of tenants at the initialization step. (b) Memory requirement (in MB) for watchlists processing for different properties by varying the number of tenants. Number of ports is also varied from 10,000 to 100,000 , and each tenant contains 5 roles.

Table 2 compares the execution time of PVSC and our alternative implementation of *intercept-and-check*, in which after detecting a critical event we collect data from the cloud and start verifying security properties using a SAT solver (e.g., Sugar [33]). We observe that verifying with the *intercept-and-check* approach including data collection takes 15 seconds (for common ownership) to 8 minutes (for no bypass) for our largest dataset. Therefore, each critical operation would experience long response time. Contrarily, PVSC experiences maximum response time of 8.5 ms. Our solution only permits allowed actions, hence any further accuracy evaluation is irrelevant.

Number of Ports	10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000	90,000	100,000
Intercept-and-check	60,200	107,209	184,230	237,245	317,252	357,261	407,268	437,271	455,276	480,277
PVSC	5.928	6.09	6.916	7.016	7.496	7.815	8.024	8.14	8.453	8.501

Table 2: Comparing execution time (in ms) between PVSC and our alternative implementation of *intercept-and-check* for the *no bypass* property.

5 Discussions

As our experiment results shown in the Section 4.2, PVSC can verify security properties for large size cloud in only few seconds at run time. There could be some cases when the pre-computed information used at run-time needs to be updated. The cases are when a change in the cloud dependency or in the cloud management API specifications occurs, or when extending verification to new security properties. In these cases, the PVSC initialization must be repeated. Even though the initialization can take several minutes, this task can be executed in parallel with run time verification and the pre-computed information updated instantly to minimize the impact on verifications at run time. Note that there are few cases where the pre-computation needs to be repeated and those cases regarding management API changes in the cloud are by nature not frequent.

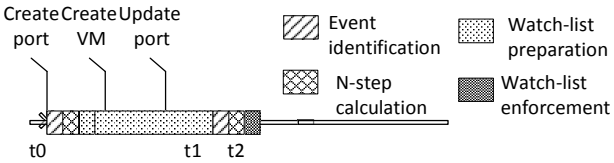

Fig. 11: Our proactive analysis approach with a batch of user requests.

Our PVSC algorithm needs only few seconds for run time verifications, this response time is satisfactory when the management operations are manually done by the administrators. But in the case of batch execution for management operations as described in [3], when these operations are executed in short intervals and if the subsequent operations impact the same watchlists, then it could imply the need for updating the same watchlists between two operations. Fig. 11 depicts how events occur before the watchlist is prepared. This watchlist update takes some time and then increases the response time for PVSC at run time. However, the worst case response time for watchlist preparation is less than 6 seconds for the largest dataset according to our experiments. Comparing this time with the *intercept-and-check* naive approach (requiring few minutes) and Weatherman [3] (requiring few hours), we consider the costs of our approach to be reasonable even for large data centers. 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 prepare concurrently watchlists and therefore reduce the response time in this case.

In this work, we cover structural properties involving cloud management operations (e.g., creating a tenant, granting a role, assignment of instances to physical hosts and the proper configuration of virtualization mechanisms). The properties involving session/context specific data are not yet considered. In our running example, if the malicious tenant can somehow successfully bypass the firewall rules and launch a spoofing attack, our solution cannot yet detect such spoofing attacks. As our solution relies on the

information reported through the management interface, any verification by extracting the information from the actual infrastructure components (e.g., virtual or hardware) is not covered in this paper and considered as a potential future work.

6 Related Work

Auditing security compliance in the cloud has recently been explored. For instance, Solonas et al. [32] detect illegal activities in the cloud only based on collected billing data in order to preserve privacy. In [19, 18], formal auditing approaches are proposed for security compliance checking in the cloud. Unlike our work, those approaches can detect violations only after they occur, which may expose the system to high risks.

VeriFlow [15] and NetPlumber [14] monitor network events and check network properties and policies at runtime to capture bugs before or as soon as they occur. They rely on incremental calculations to achieve the runtime verification. These works focus on operational network properties (e.g., black holes and forwarding loops) in traditional networks, whereas our effort is oriented toward preserving compliance with structural security properties that impact isolation in cloud virtualized infrastructures.

Various mechanisms and concepts for designing security service-level-agreement-based cloud monitoring services have been discussed in [29]. CloudSec [11] and CloudMonatt [35] propose VM security monitoring. Our work covers a larger spectrum of properties (beyond the scope of VMs) that require collecting data from various sources. In addition, unlike intercepting security measurements, we intercept multiple kinds of events and assess their impact on the cloud system before applying them. In [28], a host-based secure active monitoring mechanism, where protected hooks into untrusted VMs are installed to intercept malicious events, is proposed. Once a malicious action is intercepted, the control is transferred to security tools running on a trusted VM. They detect unwanted operations initiated by malicious softwares; whereas, our contribution is at a higher level covering events initiated by potentially untrusted users.

Proactive security analysis has been explored for software security enforcement through monitoring programs' behaviors and taking specific actions (e.g., warning) in case security policies are violated. Many state-based formal models are proposed for those program monitors over the last two decades. First, Schneider [31] modelled program monitors using an infinite-state-automata model to enforce safety properties. Those automata recognize invalid behaviors and halt the target application before the violation occurs. Ligatti [16] builds on Schneider's model and defines a more general program monitors model based on the so called edit/security automata. Rather than just recognizing executions, edit automata-based monitors are able to suppress bad and/or insert new actions, transforming hence invalid executions into valid ones. Mandatory Result Automata (MRA) is another model proposed by Ligatti et al. [17, 9] that can transform both actions and results. Narain [20] proactively generates correct network configurations using the model finder Alloy. Our work further expands the proactive monitoring approach into cloud environments differing in scope and approach.

Weatherman [3] is the most closely related work to ours. Aiming at mitigating misconfigurations and enforcing security policies in a virtualized infrastructure, Weatherman has both online and offline approaches. Their online approach intercepts manage-

ment operations for analysis, and relays them to the management hosts only if Weatherman confirms no security violation caused by those operations. Otherwise, they are rejected with an error signal to the requester. The work defines a realization model, that captures the virtualized infrastructure configuration and topology in a graph-based model. The latter is synchronized with the actual infrastructure using the approach in [2]. Two major limitations of this proposition are: i) the model capturing the whole infrastructure causes a scalability issue for the solution, and ii) the time consuming operation-checking that should be performed on the emergence of each event, makes security enforcement not feasible for large size data centers. Our work overcomes these limitations using dependency models, which are not context-dependent, and the pre-computation steps, which considerably reduce the response-time.

Congress [23] is an OpenStack project offering both online and offline policy enforcement approaches. The offline approach requires submitting a future change plan to Congress, so that the changes can be simulated and the impacts of those changes can be verified against specific properties. In the online approach, Congress first applies the operation to the cloud, then checks its impacts. In case of a violation, the operation is reverted. However, the time elapsed before reverting the operation can be critical to perform some illicit actions, for instance, transferring sensitive files before loosing the assigned role. Foley et al. [10] provide an algebra to assess the effect of security policies replacement and composition in OpenStack. Their solution can be considered as a proactive approach for checking operational properties violations, whereas our work targets the runtime verification of structural security property violations.

7 Conclusion

The near-real-time and scalable verification of security compliance with respect to security standards and policies is important to both cloud providers and users. In this paper, we proposed a scalable proactive approach that can significantly reduce the response-time of online security compliance verification in large clouds. To this end, we devised dependency models to capture the relationships between different virtual infrastructure and management operations based on related security properties. We use this dependency model for incrementally pre-compute the needed information for efficiently verify the management operations. We provided a proof of concept in OpenStack, one of the most popular cloud management platforms. Our experiment results show our proactive approach can be used for security compliance verification in large data centers with short response time. We believe this approach based on dependency models usage at verification time can be extended to other security properties and provide basis for new ways of handling proactive security compliance verification.

As future directions, we intend to deal with concurrent critical management operations; which may require a parallel or distributed approach. We will also investigate the feasibility of our solution for all security properties such as those related to network forwarding functionality with access control lists and routing policies.

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

References

1. M. Bellare and B. Yee. Forward integrity for secure audit logs. Technical report, Citeseer, 1997.
2. S. Bleikertz, C. Vogel, and T. Groß. Cloud Radar: Near real-time detection of security failures in dynamic virtualized infrastructures. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, 2014.
3. S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim. Proactive security analysis of changes in virtualized infrastructure. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC '15, 2015.
4. Cloud auditing data federation. pyCADF: A Python-based CADF library, 2015. Available at: `https://pypi.python.org/pypi/pycadf`.
5. Cloud Security Alliance. Security guidance for critical areas of focus in cloud computing v 3.0, 2011.
6. Cloud Security Alliance. Cloud control matrix CCM v3.0.1, 2014. Available at: `https://cloudsecurityalliance.org/research/ccm/`.
7. Cloud Security Alliance. CSA STAR program and open certification framework in 2016 and beyond, 2016. `https://downloads.cloudsecurityalliance.org/star/csa-star-program-cert-prep.pdf`.
8. Data center knowledge. Survey: One-third of cloud users' clouds are private, heavily OpenStack, 2015. Available at: `http://www.datacenterknowledge.com/archives/2015/01/30/survey-half-of-private-clouds-are-openstack-clouds`.
9. E. Dolzhenko, J. Ligatti, and S. Reddy. Modeling runtime enforcement with mandatory results automata. *International Journal of Information Security*, 2014.
10. S. N. Foley and U. Neville. A firewall algebra for openstack. In *IEEE Conference on Communications and Network Security (CNS)*, 2015.
11. A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy. CloudSec: A security monitoring appliance for virtual machines in the iaas cloud model. In *5th International Conference on Network and System Security (NSS)*, 2011.
12. ISO Std IEC. ISO 27002: 2005. *Information Technology-Security Techniques- Code of Practice for Information Security Management. ISO*, 2005.
13. ISO Std IEC. ISO 27017. *Information technology- Security techniques- Code of practice for information security controls based on ISO/IEC 27002 for cloud services (DRAFT), `http://www.iso27001security.com/html/27017.html`*, 2012.
14. P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013.
15. A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013.
16. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information System Security (TISSEC)*, 2009.
17. J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS'10)*, 2010.
18. 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 *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy (CODASPY)*, 2016.
19. S. Majumdar, T. Madi, Y. Wang, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi. Security compliance auditing of identity and access management in the cloud: Application to

OpenStack. In *IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.
20. S. Narain. Network configuration management via model finding. In *Proceedings of the 19th Conference on Large Installation System Administration Conference*, LISA '05, 2005.
21. NIST. SP 800-53. *Recommended Security Controls for Federal Information Systems*, 2003.
22. OpenStack. Neutron firewall rules bypass through port update, 2015. Available at: `https://security.openstack.org/ossa/OSSA-2015-018.html`.
23. OpenStack. OpenStack Congress, 2015. Available at: `https://wiki.openstack.org/wiki/Congress`.
24. OpenStack. OpenStack open source cloud computing software, 2015. Available at: `http://www.openstack.org`.
25. OpenStack. OpenStack user survey, 2015. Available at: `https://www.openstack.org/assets/survey/Public-User-Survey-Report.pdf`.
26. OpenStack. OpenStack audit middleware, 2016. Available at: `http://docs.openstack.org/developer/keystonemiddleware/audit.html`.
27. OpenStack. OpenStack command list, 2016. Available at: `http://docs.openstack.org/developer/python-openstackclient/command-list.html`.
28. B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy (SP'08)*, 2008.
29. D. Petcu and C. Craciun. Towards a security SLA-based cloud monitoring service. In *Proceedings of the 4th International Conference on Cloud Computing and Services Science*, 2014.
30. R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 1996.
31. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 2000.
32. M. Solanas, J. Hernandez-Castro, and D. Dutta. Detecting fraudulent activity in a cloud using privacy-friendly data aggregates. Technical report, arXiv preprint, 2014.
33. N. Tamura and M. Banbara. Sugar: A CSP to SAT translator based on order encoding. In *Proceedings of the Second International CSP Solver Competition*, 2008.
34. B. Tang and R. Sandhu. Extending openstack access control with domain trust. In *Network and System Security*, pages 54–69. Springer, 2014.
35. T. Zhang and R. B. Lee. Cloudmonatt: an architecture for security health monitoring and attestation of virtual machines in cloud computing. In *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

A Security Properties in Access Control Dependency Model

Table 3 illustrates an excerpt list of the covered security properties, the critical and watchlist events related to them, and the content of their associated watchlists. In the following example, we present how the access control dependency model captures common ownership security property and its related events.

Example 5 Access Control Dependency Model (user-level): *The critical management operation that leads to the violation of the "common ownership" security property is* `grant role`. *The model in Fig. 3(b) includes user (vertex 2), tenant (vertex 3) and role (vertex 5). The vertex 11 is a specific vertex grouping a role and a tenant pair. The* `grant role` *operation is related to the entities user and tenant-role pair (edge (2,11) in Fig. 3(b)). As it can be seen in Fig. 3(b), the* `grant role` *operation depends on*

Property	Critical Event	Watchlist Event	Watchlist per tenant
Common role ownership [13, 6]	grant role (2,11)	create role (3,5)	roles in a tenant
		delete role (3,5)	
No cross-tenant token	create token (3,6)	grant role (2,11)	user-tenant-tole tuple
		create user (1,2)	
Cardinality [12, 21]	grant role (2,11)	delete role (3,5)	counter for each role
		deny role (2,11)	
		grant role (2,11)	
Role activation [13, 6]	create token (3,6)	grant role (2,11)	user-role pair
Permitted action [13, 6]	request an operation (8,9)	create token (3,6)	token-operation pair
		grant role (2,11)	
User-access validation [13, 6]	request an operation (8,9)	create token (3,6)	token-operation pair
		grant role (2,11)	

Table 3: An excerpt of the security properties supported by the access control management dependency model shown in Fig. 3(b) with their corresponding critical events, watchlist-related events, and the content of the watchlists.

other operations such as `create user` *(edge (1,2)) and* `create role` *(edge (3,5)). More precisely, the* `grant role` *operation allows a user (vertex 2) to obtain a role (vertex 5) in a tenant (vertex 3). As the* `create role` *and* `create user` *operations are closely related to the actual critical operation (*`grant role`*), our model captures this dependency relationship and aids to avoid the security violation by taking preparation from the* `create role` *operation. Furthermore, these operations in turn depend on the existence/creation of a tenant and a domain. This induces a chain of dependencies between a set of events that could be related to this security property.*

B Algorithms

Algorithms 1 and 2 show two phases of our proactive verification approach.

Algorithm 1: Initialization phase

```
procedure INITIALIZE(DependencyModels,SecPropertyNth,CloudOS)
    Events=GetEvents(DependencyModels)
    Mapping=ReadEventOperationMapping(Events,CloudOS)
    for each event e ∈ Events do
        Populate Event-operation
    for each property p ∈ SecPropertyNth do
        Populate Property-N-thresholds
    GetDependencyGraphInfo(DependencyModels)
    InitializeWatchlist(p)
procedure GETDEPENDENCYGRAPHINFO(DependencyModels)
    for each property p ∈ SecPropertyNth do
        criticalRelation = getCriticalEdge(p,DependencyModels)
        Allpaths = ComputeN-allpaths(DependencyModels,criticalRelation)
        for each path in Allpaths.paths do
            Populate Model-N-property
        Edges = getEdges(DependencyModels)
        for each ed ∈ Edges do
```

(event, model, type) =ReadAtributes(*ed*,DependencyModels)
Populate Model-event from DependencyModels

procedure INITIALIZEWATCHLIST(Property p)
Populate Watchlist(p) from cloud database

Algorithm 2: Run-time phase

procedure RUN−TIME(Operation op)
Find Event e for op from Event-operation
Find e.type from model-event
if e.type=CE **then**
Find security properties p_i in model-event such that e.type=CE
if validateCompliance(p_i, op.params)=allow **then**
Allow op in the cloud
else
Deny op in the cloud
else
Allow op in the cloud
Find all security properties P in model-event
for each property $p \in P$ **do**
evaluateNstep(e, p. op.params)

procedure EVALUATENSTEP(Event e, Property p, Params opparams)
Find N-th for p from Property-N-thresholds
Find entities in the model related to p
context = CollectCloudData(entities)
Find N-cp for p and context from Model-N-property
if N-cp=N-th **then**
updateWatchlist(p, opparams)
else if N-cp $<$ N-th and e.type=WE **then**
updateWatchlist(p, opparams)

procedure VALIDATECOMPLIANCE(Property p, Params opparams)
if opparams is in p.watchlist **then**
Return allow
else
Return enforce

procedure UPDATEWATCHLIST(Property p, Parameters opparams)
Update watchlist based on p and opparams
