ProSAS: Proactive Security Auditing System for Clouds

Suryadipta Majumdar, Gagandeep Singh Chawla, Amir Alimohammadifar, Taous Madi, Yosr Jarraya, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi

Abstract—The multi-tenancy in a cloud along with its dynamic and self-service nature could cause severe security concerns, such as isolation breaches among cloud tenants. To mitigate such concerns and ensure the accountability and transparency of the cloud providers towards their tenants, verifying cloud states against a list of security policies, a.k.a. *security auditing*, is a promising solution. However, the existing security auditing solutions for clouds suffer from several limitations. First, the traditional auditing approach, which is retroactive in nature, can only detect violations after the fact and hence, often becomes ineffective while dealing with the dynamic nature of a cloud. Second, the existing runtime approaches can cause significant delay in the response time while dealing with the sheer size of a cloud. Finally, the current proactive approaches typically rely on prior knowledge about future changes in a cloud and also require significant manual efforts, and thus become less practical for a dynamic environment like cloud. To address those limitations, we present a novel proactive security auditing system, namely, *ProSAS*, which can prevent violations to security policies at runtime with a practical response time, and yet does not require prior knowledge about future changes. More specifically, *ProSAS* first establishes its models (e.g., dependency relationships between cloud events, and critical events) through learning from historical data (e.g., logs); it then predicts future critical events which would likely follow a received event by leveraging the dependency relationships; afterwards, it proactively verifies the impacts of those future events, and prevents those events which can cause violations of security policies. ProSAS is integrated into OpenStack, a popular cloud management platform, and we provide a concrete guideline to port ProSAS to other popular cloud platforms, such as Google Cloud Platform, and Amazon EC2. Our experiment results using both real and synthetic data demonstrate the improvement of efficiency (i.e., reducing response time to 1,450 nanoseconds at best and 8.5 milliseconds on average for a large-scale cloud with 10,000 tenants) and level of automation (i.e., learning more than 20 new critical events spanning 100 days) in proactive security auditing by ProSAS.

Index Terms—Security Auditing, Runtime Enforcement, Cloud Security, Proactive Auditing, Continuous Auditing, OpenStack.

1 Introduction

Multi-tenancy in cloud is a double-edged sword that may lead to various security concerns in spite of its transformative contribution to resource optimization [1], [2]. Such security concerns are evident from a wide range of attacks reported in both the literature (e.g., [3], [4], [5]) and the industry (e.g., [6], [7]). As a result, the accountability and transparency of cloud service providers often become questionable to cloud tenants [1]. To defend against security threats and build the trust of users, verifying security policies using formal verification methods, a.k.a. *security auditing*, has been a standard practice for years in the industry (e.g., Delloite [8] and KPMG [9]) and is a desirable solution for clouds.

However, security auditing in clouds presents several unique challenges. First, the dynamic and self-service nature of clouds means any auditing result may become obsolete very quickly and therefore, a runtime security auditing process is desirable for ensuring continuous protection against security threats. Second, the sheer size and high operational complexity of clouds means a runtime solution must be highly efficient and scalable in order to ensure a practical response time to users. Finally, the co-existence of a large number of tenants and users with different needs implies that an auditing solution cannot assume users' behaviors to follow any fixed or previously known patterns.

The existing security auditing approaches still fall short to overcome such challenges. First, the *retroactive* approaches (e.g., [10], [11], [12]) catch violations of security policies after the fact by verifying cloud states (e.g., configurations and logs). As a result, 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., [13], [14], [15], [16]) audit the impacts of each change request to the cloud before granting them, which leads to a substantial delay to users' requests. Third, the *proactive* approaches in [13], [14] verify potential user requests in advance, by assuming a known sequence of user requests, namely, the change plan; however, such an assumption about fixed change plans might not always be realistic, especially considering the diverse and fast evolving needs of cloud tenants and users. Through the following example, we further motivate towards our solution.

Motivating Example. Fig. 1 depicts three timelines (showing different steps of typical retroactive and intercept-and-check approaches, and our proactive solution, respectively) with a sequence of three cloud events. Among those events, the *update port* is a *critical event*, which can potentially breach a security policy. In this example, we consider a security policy that audits anti-spoofing mechanisms in the cloud; which can be violated by real world vulnerabilities (e.g., OpenStack [17] vulnerability, OSSA-2015-018, [18]). We highlight the major limitations of the existing

- *S. Majumdar, G.S. Chawla, A. Alimohammadifar, L. Wang and M. Debbabi are with The Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada. E-mail: majumdar, g_chawla, ami_alim, wang and debbabi@encs.concordia.ca*
- *T. Madi, Y. Jarraya and M. Pourzandi are with The Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada. E-mail: taous.madi, yosr.jarraya and makan.pourzandi@ericsson.com*

Manuscript received April 06, 2020.

approaches and position our solution as follows.

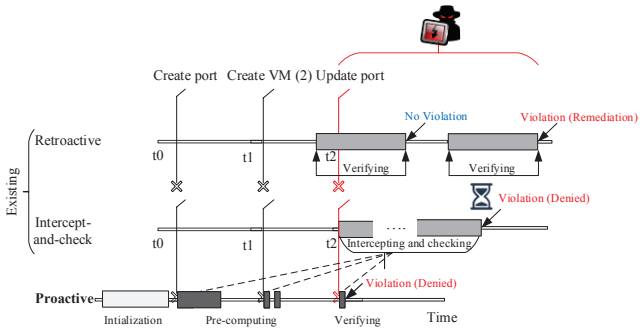

Fig. 1: Comparison of the execution time of our solution with the typical intercept-and-check and retroactive approaches

- A typical retroactive auditing (as shown in the first timeline) is conducted periodically (e.g., at time t_2 and t_3) within a certain interval. Due to its after-the-fact nature, it allows attackers to exploit the vulnerable systems for a considerable amount of time (e.g., seconds to minutes for a medium to large scale cloud [10], [11]) with potentially irreversible damages (e.g., information leakage, data corruption, and DoS).
- A typical intercept-and-check approach (as shown in the second timeline) overcomes the above-mentioned limitation of retroactive auditing. However, as it starts the verification only after the *update port* event occurs, this approach can result in a significant delay (e.g., four minutes as reported in [13]).
- Our approach (as shown in the last timeline) performs security auditing in a proactive manner, i.e., it prepares for the critical event (*update port*) ahead of the actual occurrence of that event, and consequently it can ensure a much shorter response time. Moreover, unlike existing proactive approaches (not shown in the timeline), we do not assume events will follow a fixed future change plan (e.g., the fixed sequence *create port, create VM and update port*), as will be detailed in later sections.

More specifically, we present a proactive security auditing system for clouds, namely, *ProSAS*. First, *ProSAS* learns a list of cloud events (namely, *critical events*) that may violate a security policy by utilizing a formal verification method on the cloud state (e.g., logs and configurations). Second, it learns various (e.g., structural, probabilistic and temporal) dependency relationships between cloud events from historical data (e.g., logs). Third, it proactively verifies the future critical events, which are predicted based on their dependency relationships with the current event, against security policies, and prepares a list of allowed parameters (namely, *watchlist*) for those events. Finally, when a critical event actually occurs, *ProSAS* utilizes and recycles those verification results to efficiently enforce the security policies. Thus, our approach can be regarded as a "synthesis" of the three existing types of approaches (retroactive, intercept-and- check, and proactive) so that it can utilize the bests of each world (e.g., high accuracy of retroactive, runtime enforcement capabilities of intercept-and-check, and high efficiency of proactive) while avoiding their major limitations (e.g., after-the-fact nature of retroactive, significant delay of intercept-and-check, and lower accuracy of proactive). Specifically, ProSAS ensures that: (i) proactive becomes more accurate through learning new critical events using a retroactive method, (ii) retroactive is only used in an offline learning so that its after-the-fact nature does not affect the runtime enforcement capability of ProSAS, and (iii) intercept-and-check can respond in a few milliseconds with the help of a proactive approach. We implement and integrate *ProSAS* into OpenStack [17]. We also provide a concrete guideline outlining the major steps to adapt our system to other popular cloud platforms (e.g., Amazon EC2 [19], Google GCP [20], and Microsoft Azure [21]). Our experimental results using both real and synthetic data confirm the efficiency, accuracy, and scalability of our approach.

The main contributions of our paper are as follows.

- To the best of our knowledge, *ProSAS* is the first efficient and scalable proactive security auditing system for clouds that can reduce the response time to a practical level for a large-scale cloud (e.g., 8.5 milliseconds to audit 10,000 tenants).
- We are the first to propose an approach to learn critical events from historical data. Our method can assist human analysts to more effectively identify critical events, which could in turn improve the accuracy of proactive security auditing. Our experimental results show the learning of more than 20 new critical events spanning 100 days.
- The practicality of the proposed system is confirmed through its integration into the popular cloud platform OpenStack, the experiments using real cloud data, and the concrete guidelines for adapting the system to other major cloud platforms.

The preliminary version of this paper has appeared in [22], which proposes the basic approach of proactive security auditing. In this work, we significantly extend this approach in the proposed ProSAS system, while focusing more on the system aspects, such as improving the level of automation for various components, and improving both the accuracy and efficiency of the overall system. Specifically, our major extensions are as follows: i) we design and implement a new system component for automatically learning the inputs (e.g., critical events and dependency models) to reduce the required manual efforts and improve the accuracy (Section 3.2); ii) we design and implement a new system component to significantly improve the response time through preserving recent actions and results (Section 3.4); iii) we design and implement a new system architecture to integrate those new components into the ProSAS system based on OpenStack [17] (Section 4); iv) we provide a concrete guideline to port ProSAS to other major cloud platforms (e.g., Amazon EC2, Google GCP, and Microsoft Azure) (Section 4); and v) finally, we evaluate both efficiency and accuracy of ProSAS through a new set of experiments (Section 5); which demonstrates significantly improved performance over our previous solution [22].

The paper is organized as follows. Section 2 describes the threat model and the dependency models. Section 3 details our methodology. Section 4 provides the implementation details, and Section 5 presents the experimental results. Section 6 discusses different aspects of our approach. Section 7 summarizes related works and compares them with our approach. Section 8 concludes the paper providing future research directions.

2 Models

This section defines our threat model and dependency models.

2.1 Threat Model

Even though the high-level idea of proactive auditing can potentially be applied to various IT infrastructures, the design and implementation required for realizing such an idea would heavily depend on various aspects of the targeted infrastructures such as the logging systems, interception methods, event types, event

dependencies, etc. In this paper, we focus on the specific context of cloud management operations (e.g., *create VM*, *create tenant*, etc.) in an Infrastructure as a Service (IaaS) cloud management platform. We assume that the cloud management platforms: (a) may be trusted for the integrity of the management operations, their notifications, and database records (existing techniques on trusted computing and remote attestation may be applied to establish a chain of trust from TPM chips embedded inside the cloud hardware, e.g., [23], [24], [25], [26]), and (b) may have implementation flaws, misconfigurations and vulnerabilities that can be potentially exploited by malicious entities to violate security policies specified by cloud tenants. The cloud users including cloud operators and agents (on behalf of a human) may be malicious. Any threats directing from the cloud management operations is within the scope of this work. Therefore, any violation bypassing the cloud management interface is beyond the scope of this work. Also, our focus is not to detect specific attacks or intrusions, even though our framework may catch violations of specified security policies due to either misconfigurations or vulnerabilities. In this work, we mainly support structural policies that involve cloud management operations (e.g., creating a tenant, creating a VM, granting a role, and assigning VMs to physical hosts) and as long as those operations are covered in our dependency models in Section 3.2. We rely on security experts to define the policies based on literature, security standards, and other requirements of a tenant. We assume that before our runtime approach, 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 is required. Moreover, we rely on the selected offline auditing tools (e.g., formal methods) and the accuracy of our learning method can only be as accurate as those tools. To make our discussions more concrete, the following shows an example of in-scope threats based on a real vulnerability.

Running Example. A real world vulnerability in OpenStack [18] can be exploited to bypass security group rules (which are fine-grained, distributed security mechanisms in several cloud platforms including Amazon EC2 [19], Microsoft Azure [21] and OpenStack [17] to ensure isolation between instances). Fig. 2 shows the attack scenario to exploit this vulnerability. In this example, the owner of the `VM127` is malicious, and `VM207` and `Port788` are legitimate. 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 a security group gets attached to it (race condition) [18]. As a result, the security group 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) [18].

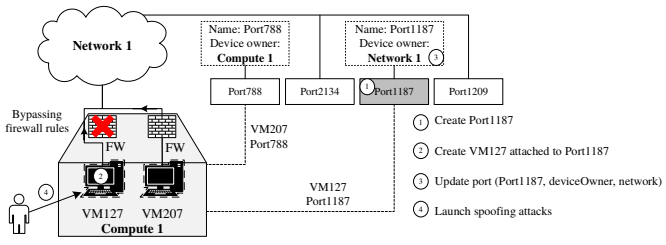

Fig. 2: An exploit of a vulnerability in OpenStack [18], leading to bypassing security group rules.

2.2 Dependency Models

This section first provides background on different dependency relationships (e.g., structural, probabilistic and temporal) among cloud events that our dependency model covers, and then formally defines the model that will later be used in our system.

Structural Dependency. The structural dependency relationships mainly indicate the structure (e.g., nodes and edges) of a dependency model. The structural dependencies are derived mainly from two different sources. (i) From the cloud design: this captures the structure of the dependency that is imposed by the cloud platform (e.g., OpenStack [17]). For example, in OpenStack, a subnet cannot be created before there exists a network; which indicates that the *create subnet* event depends on the *create network* event. (ii) From the cloud behavior: this captures the structure of the dependency that is derived from the behavior of a specific cloud deployment. For example, by analyzing historical data (e.g., logs) of a cloud, we learn all possible transitions which potentially become the nodes and edges of our dependency model.

Probabilistic Dependency. The probabilistic dependency mainly indicates the probabilistic behavior of a cloud. The probabilistic dependencies are derived mainly from the cloud behavior. More specifically, the probabilistic behavior of a cloud is learnt from the historical data (e.g., logs) of a cloud deployment, and expressed in term of probabilities. For example, if the transition from the *create network* event to the *create subnet* event is observed five out of ten times, then the probabilistic dependency is calculated as 0.5.

Temporal Dependency. The temporal dependency mainly indicates the temporal behavior of a cloud. The temporal dependencies are derived mainly from the cloud behavior. More specifically, the temporal behavior of a cloud is learnt from the historical data (e.g., logs) of a cloud deployment, and expressed in term of intervals. For example, if the transition from the *create network* event to the *create subnet* event is observed to have an interval of five minutes on average, then the temporal dependency for this transition can be indicated as $5m$.

The Definition of Dependency Model. For a given list of cloud events *events* and the historical data of those events *hist*, our dependency model is stated as a Bayesian network $BN = (G, P)$, where G is a directed acyclic graph in which each node corresponds to a cloud event in *events* and each directed edge between two nodes indicates that a transition between these two nodes is observed in *hist* where edge is labelled with the list of parameters P. In this paper, G in the dependency model is obtained from the structural dependencies, and P is derived from both probabilistic and temporal dependencies.

3 Proactive Security Auditing System

This section describes our proactive security auditing system.

3.1 Overview

Fig. 3 illustrates the high-level design of our proactive security auditing system (ProSAS). ProSAS performs proactive security auditing for clouds in three major steps: input learning, proactive security verification, and verification result utilization. In Step 1 (detailed in Section 3.2), to establish its models, ProSAS first learns critical events (that may potentially violate a security policy), then learns dependency relationships (that will later be leveraged in our proactive security verification) among cloud

events, and finally initializes other inputs (e.g., identifying event types and defining watchlist content) to ProSAS. In Step 2 (detailed in Section 3.3), to proactively verify security policies, it first intercepts each cloud event, then identifies future events using the dependency relationships, and finally, builds watchlists (that contain allowed parameters) for critical events. In Step 3 (detailed in Section 3.4), to utilize the verification results, ProSAS preserves the recent proactive verification actions (e.g., updating watchlists) that are performed for non-critical events, then preserves recent proactive decisions that are taken for critical events, and finally, utilizes those actions/decisions to quickly respond to a runtime event.

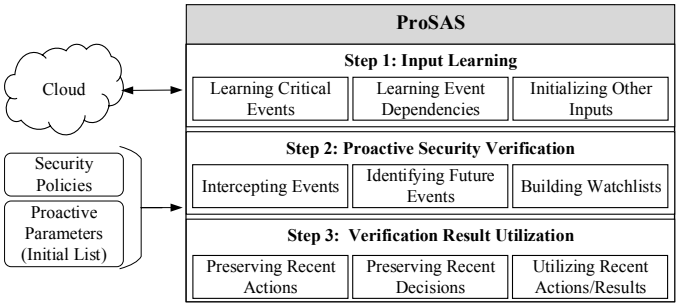

Fig. 3: An overview of ProSAS

3.2 Input Learning

The section elaborates on how ProSAS establishes its different models, such as, critical events, dependency relationships, etc.

3.2.1 Learning Critical Events

ProSAS learns a list of critical events (which may violate a security policy) for different security policies through an offline and iterative process. This list will later be used in our proactive security verification approach in Section 3.3. More specifically, it follows four major steps as shown in Fig. 4 and described in the following). Note that Steps 1-3 are automatically performed by ProSAS and Step 4 requires the intervention of a security expert.

1) The first step is to detect violations of security policies using an offline auditing tools (e.g., [10], [11]). More specifically, ProSAS first collects the state (e.g., configurations) of a cloud from different cloud services (e.g., computing and networking). It then converts those configuration data into the format (i.e., first order logic) of an offline auditing tool (e.g., [10], [11]). Afterwards, it applies that tool to verify the current cloud configurations. Finally, ProSAS identifies any violation of a security policy reported by the tool. Note that these offline tools are chosen for this step, mainly because they consider the entire state of the cloud and hence, are able to detect the violations caused by new critical events that are not yet included in our critical event list. For this work, we mainly rely on a well-known formal method, constraint satisfaction problem (CSP) solver [27], that have passed the proof-of-time as well as provided theoretical accuracy proof (e.g., [28], [29], [30], [31]). Additionally, our previous work [32] shows that various other well-known formal methods (e.g., Datalog, access control model) can also be leveraged for this step depending on the nature of security policies.
2) The next step is to identify log entries for the period during which a violation has occurred. More specifically, ProSAS first finds the cloud service(s) (e.g., network, compute and storage) related to the violated security policy. It then collects event logs for that cloud service. Finally, ProSAS identifies the log entries that occurred during the current iteration of the learning critical event process.
3) The following step is to shortlist the candidate critical events for the violation by eliminating irrelevant log entries from the collected log entries. More specifically, ProSAS first interprets the log entries from the output of the previous step to identify their event types. For instance, the log entry `POST /v2/servers HTTP/1.1` indicates the event type `Create VM`. It then filters out the irrelevant events that are automatically generated by the system and not caused by an event for a cloud management change. For example, the log entry `GET /v2/os-security-groups HTTP/1.1` is generated when a system lists all its security groups on the interface. Thus, ProSAS prepares a shortlist of candidate critical events for the next step.
4) The final step is to identify the responsible critical event for the violation. This step involves a security expert, who identifies the critical event that violates a security policy from the short list (from the previous step) based on his/her discretion. Note that this is the only step in the learning process which requires human interventions.

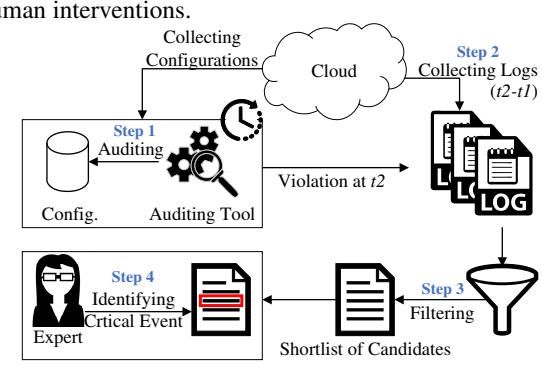

Fig. 4: The steps of learning critical events

Example 1 Fig. 4 depicts the steps of learning critical events. For this example, we consider two iterations of this process at time t_1 and t_2, respectively, where $t_2 > t_1$. First, ProSAS collects the cloud configurations at time t_1, and verifies the *no bypass* policy using our offline auditing tool (e.g., [10], [11]). At time t_1, there is no violation of the policy, and hence, the further steps are not required to perform. Second, ProSAS again collects the cloud configurations at time t_2 and verifies the same policy using our offline auditing tool. At time t_2, there is a violation and therefore, ProSAS performs the following steps. It first collects logs from the network service of the cloud for the period of t_1-t_2; as it is certain that the critical event caused this violation happens within this period. Afterwards, ProSAS filters out all events with the *GET* requests, because those events are interface-generated to show lists of different resources on the interface and not related to cloud management operations. Finally, ProSAS presents a shortlist of events to an expert, who finally identifies *update port* as the responsible critical event for the violation of the *no bypass* policy. Similarly, ProSAS learns critical events for different security policies. The learnt critical events for both virtual-infrastructure-related and access-control-management-related policies are shown in Tables 1 and 2, respectively.

Through this step, ProSAS learns new critical events which eventually contributes to prevent more security violations of poli-

cies by ProSAS and thus, the effectiveness of the framework is enhanced (as shown through the experiment results in Section 5.2).

Policy	Critical Event (CE)	Watchlist Event (WE)	Watchlist per tenant
No bypass [33]	update port (15,15)	create VM (16,17) create port (12,15)	Ports except VM ports
Port consistency [33], [34]	create vPort (21,20)	create port (12,15)	ports at tenant layer
No abuse of resources [33]	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 [33]	attach port to a router (16,18)	create router (3,18)	router-tenant pair
Port isolation [33], [34]	add vPort to vNet (19,20)	create vNet (14,19)	vNets in a subnet
No co-residency [33], [34]	create VM (16,17), migrate VM (17,22)	create VM (16,17)	Hosts with no conflicting VMs
		migrate VM (17,22)	

TABLE 1: Security policies supported by the cloud infrastructure model shown in Fig. 5(a) with their corresponding critical and watchlist events, and the watchlist contents.

Policy	Critical Event	Watchlist Event	Watchlist per tenant
Common role ownership [33], [34]	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 [35], [36]	grant role (2,11)	delete role (3,5)	counter for each role
		deny role (2,11)	
		grant role (2,11)	
Role activation [33], [34]	create token (3,6)	grant role (2,11)	user-role pair
Permitted action [33], [34]	request an operation (8,9)	create token (3,6)	token-operation pair
		grant role (2,11)	
User-access validation [33], [34]	request an operation (8,9)	create token (3,6)	token-operation pair
		grant role (2,11)	

TABLE 2: Security policies supported by the access control management dependency model shown in Fig. 5(b) with their corresponding critical events, watchlist-related events, and the content of the watchlists.

3.2.2 Learning Dependency Relationships

ProSAS learns several dependency relationships among cloud events. To this end, it first learns the structural dependency between events by analyzing historical cloud data (e.g., logs) and studying API documentations (e.g., [37]) of cloud platforms. It then learns probabilistic and temporal dependencies from the historical cloud data (e.g., logs). Later these dependency relationships are leveraged to conduct the proactive security verification.

Building Structural Dependencies. ProSAS builds structural dependencies among cloud events by following two main steps. First, ProSAS adopts the automated structure learning process (proposed in [38]). To this end, it collects logs from different cloud services (e.g., compute, network and storage), processes them to identify the sequence of different events from the logs, and builds the structure of the dependency model from those sequence of events; which is further described in LeaPS+ [38]. Second, it further learns missing structures by identifying the dependency relationships that are imposed by the cloud design through studying API documentations (e.g., [37]) from different cloud services (e.g., Neutron, Nova, and Keystone) and Open vSwitch [39] (similarly as in [13]).

Example 2 Fig. 5 illustrates the two dependency models that we derive for an OpenStack-managed cloud covering virtual infrastructure (Fig. 5(a)) and user access control (Fig. 5(b)). For the user access control model, we are inspired by the OSAC model by Tang et al. [16]. To build the intuitions of these models, we start by providing an example on how the cloud infrastructure dependency model (see Fig. 5(a)) allows us to relate actual management operations or events happening in the cloud to the "no bypass" security policy presented in Section 2.1. The model in Fig. 5(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. 5(a)). As it can be seen in Fig. 5(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 later aids to avoid the security violation.

Learning Probabilistic and Temporal Dependencies. To learn both probabilistic and temporal dependency relationships, ProSAS utilizes both Bayesian network and time-series model, respectively, (similarly as Proactivizer [32]). To that end, ProSAS first collects logs from different cloud services (e.g., compute, network, and storage). Then, it processes them to identify the transitions of events from those logs. Afterwards, it obtains the frequencies and intervals of those transitions. Next, it utilizes those transitions and their frequencies to obtain a probabilistic model (e.g., Bayesian network). Finally, ProSAS feeds the intervals between transitions and the Bayesian networks for that interval to the time-series predictor (e.g, ARMAX [40]) to build the dependency models, which will be leveraged in Section 3.3 for proactive verification.

3.2.3 Initializing Other Inputs

This section describes the initialization of other inputs of ProSAS. More specifically, ProSAS maintains several tables for its proactive verification step in Section 3.3.

- `Event-operation`: maps event types to operations in different cloud environment to easily integrate different cloud implementations.
- `Model-event`: relates each security policy with the elements of the dependency models and tenant inputs including the types of events.
- `policy-WL`: stores the specification of the contents in a watchlist for each security policy.
- `policy-N-thresholds`: maps security policies and their associated thresholds (denoted as *N-th*), where thresholds are security policy specific and inputs from the administrators. A brief guideline on choosing this threshold is provided in Section 6.
- `Model-N-policy`: stores all possible values of *N* (denoted as *N-cp*) for each policy.

To initiate those tables, it collects necessary data from different cloud services (e.g., compute, network and storage), and to preprocess the data to prepare the conditions for those tables. This step also initializes the watchlist content with the current cloud context.

Example 3 Fig. 6 shows the initialized inputs for the no bypass policy. The `Event-operation` table shows that the `create port` event corresponds to the `neutron port-create` operation in OpenStack. The `Model-event` table stores an entry indicating that the `create port` event is the watchlist event (WE) for the no bypass policy and situates at the edge between nodes 12 and 15 in the dependency model. Also, the critical event `update port` for the policy with its position (i.e., the node 15) in the dependency model is stored in this table. Other events of type TE, such as `create network` and `create subnet`, are not shown for brevity. The minimal distance from the critical event

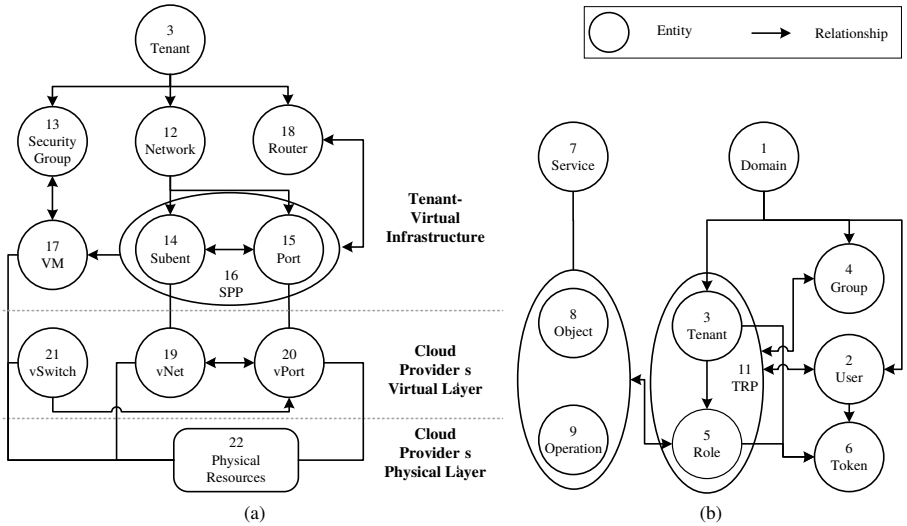

Fig. 5: Structural dependency models: (a) cloud infrastructure and (b) access control management

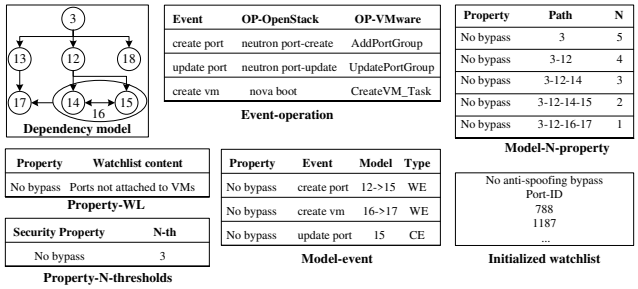

Fig. 6: An excerpt of the initialized inputs for the *no bypass* policy

at which our solution should react is (N-$th = 3$), as shown in the `policy-N-thresholds` table. The `Model-N-policy` table stores all possible computed values of N taking into account the security policy and the dependency model. Finally, the watchlist is initialized for the no bypass policy 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 `policy-WL` table.

3.3 Proactive Security Verification

This section details how ProSAS conducts proactive security verification. To that end, ProSAS first intercepts runtime events, then identifies the most probable future events by leveraging the dependency models, builds watchlists (with the allowed parameters) for those future events, and finally enforces the security decisions (e.g., allow or deny) while checking the requested parameters with the watchlists when a critical event actually occurs. In the following, we elaborate on each of them.

3.3.1 Intercepting Runtime Events

At runtime, ProSAS intercepts all event instances performed in the cloud. Usually, the intercepted event instances provide implementation specific details. Therefore, with the help of the `Event-operation` table, we identify the corresponding event type (so that the remaining steps in ProSAS become cloud-platform-agnostic). Table 3 shows an excerpt of such mapping. We also identify the criticality (i.e., CE, WE or TE) of the intercepted event type from the `Model-event` table. Only if the intercepted event is critical, then we halt the event request till the verification is performed. Otherwise, the event request is immediately processed. Additionally, the position of the event type in the dependency model is identified so that the next step can identify the most probable future critical events.

OpenStack Event Instances	Event Types of ProSAS
`POST /v2/servers HTTP/1.1`	`Create VM`
`POST /v2/os-security-groups HTTP/1.1`	`Create security group`
`GET /v2/os-security-groups HTTP/1.1`	Eliminated

TABLE 3: Examples of OpenStack event instances and converted event types in ProSAS

3.3.2 Identifying the Most Probable Future Event

ProSAS identifies the most probable future events in two steps.

First, it calculates the probabilities of the occurrence of a critical event from the current event using the dependency model. More specifically, ProSAS first traverses the dependency graph for each security policy from the edge corresponding to its critical event backward until reaching the node for the current event in the model. It then finds out all dependent events and entities. Finally, it stores the probabilities for each possible configuration in the `Model-N-policy` table. Note that, a configuration is an abstract state that allows to determine whether the entities that the security policies depend on, actually exist.

Example 4 In the following example, we populate the `Model-N-policy` table for the *no bypass* policy with the probabilities of the occurrence of the critical event `update port` for different cloud configurations (using the dependency model in Fig. 7). First, we consider the cloud configuration with one tenant and without any network, subnet, ports, or VMs, then the cloud configuration is considered to be on the vertex 3. The probability for the vertex 3 in the `Model-N-policy` table indicates the probability of the occurrence of the critical event `update port` from the current event `create tenant`. Similarly, after the `create network` event occurs, the then configuration is indicated as the vertices $(3,12)$ and the probability for this entry in the `Model-N-policy` table indicates the probability of the occurrence of the critical event `update port` from the event `create network`. The similar steps are followed to fully populate the `Model-N-policy` table.

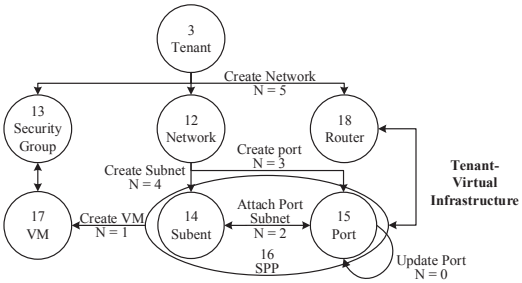

Fig. 7: Part of the cloud infrastructure dependency model annotated with the probabilities of each transition that is relevant to the *no bypass* policy

Second, it identifies the most probable future critical events for different security policies. To this end, ProSAS first finds the event type of the current event (similarly as in LeaPS+ [38]). If the event type is non-critical, it then extracts the related contextual data (e.g., corresponding tenant, network, subnet, etc.) from both event parameters with the help of our interceptor middleware (as detailed in Section 4.2) as well as cloud configurations (that are usually stored in a database) to determine the path to be selected from the `Model-N-policy` table. Finally, it marks those critical events as the most probable future event, for which the probability is bigger than the threshold.

Example 5 *To identify the most probable event from the current event (*"`POST /v2.0/ports.json HTTP/1.1`"*), ProSAS first identifies its event type,* `create port`*. Second, as the* `create port` *is non-critical for the* no bypass *policy (according to Table 1), it extracts the current configuration, which is* 3-12-14*, and finds the corresponding probability,* 0.8*, from the* `Model-N-policy` *table. Finally, as the probability is higher than the threshold, it identifies the related critical event,* `update port`*, as one of the potential future events.*

3.3.3 Building Watchlists for Future Events

This step is to build watchlists (which is a list of allowed parameters for critical events) for each security policy that will later be used for runtime security enforcement. To this end, ProSAS first finds the condition for the watchlist of the future critical events from the Tables 1 and 2. Second, it collects the parameters from the cloud for which the condition is satisfied. Finally, it stores those parameters along with their corresponding critical event and security policy. Thus, ProSAS incrementally prepares a watchlist with all allowed parameters for a critical event so that at runtime we can simply check the requested event parameters against this list and quickly take an enforcement decision.

Example 6 *To build the watchlist for the future event* `update port`*, ProSAS first finds that the watchlist of the critical event,* `update port`*, must only include the ports that are not connected to any VM. Second, it collects all such port IDs (*`2134`*,* `1209`*,* `and 1187`*). Third, it stores all those port IDs in the watchlist for the* no bypass *policy. For another consequent event* `create vm` *with the port ID* 1187*, ProSAS removes the port,* 1187*, from the watchlist of this policy, because after the most recent event, this port is now connected to a VM, and hence, ineligible to remain in the watchlist.*

3.4 Verification Result Utilization

ProSAS utilizes the proactive verification results (that are obtained in the previous step) to enforce security policies on the cloud. More specifically, ProSAS first maintains a list of actions that are performed for recent non-critical events, then preserves the verification results for recent critical events, and finally, utilizes those actions/results to respond to the runtime events. Note that our observation (as reported in Section 5) indicates that a cloud deployment often experiences similar event requests and hence, reusing recent actions/results can significantly improve the response time of our solution (as demonstrated in Figure 10).

3.4.1 Preserving Recent Actions on Non-Critical Events

The first step of recycling is to preserve the recent most proactive actions (e.g., finding most likely future events and the needed update on their watchlists) taken by ProSAS for a non-critical event. To this end, ProSAS stores the recently requested non-critical events. It also marks if the event is a potential future event (based on the outcome of the step in Section 3.3.2). Furthermore, it preserves the proactive actions (if any) that have been taken for that event (in the step in Section 3.3.3). For this step, ProSAS maintains a table (which is implemented as a cache and detailed in Section 4) with three attributes: *Recent Events*, $N\text{-}cp < N\text{-}th$, and *Actions Taken*. We further demonstrate its usage through the following example.

Example 7 Table 4 shows an excerpt of the list with the details of three recent non-critical events. The `create VM` event is observed first. Its distance from the critical event `update port` is lower than the threshold, and hence, no proactive action is taken. Second, the `create port` event occurs. Its probability is smaller than the threshold and hence, an insertion into the watchlist for the *no bypass* security policy is performed. Third, similarly for the `delete VM` event, a parameter is removed from the watchlist for another policy, named *no_ downgrade*.

Recent Events	N-cp < N-th	Actions Taken
Create VM	No	-
Create Port	Yes	Insert into no_bypass watchlist
Delete VM	Yes	Remove from no_downgrade watchlist

TABLE 4: An excerpt of preserving recent actions for non-critical events

3.4.2 Preserving Recent Actions on Critical Events

The following step is to preserve the recent actions (e.g., updating watchlists) taken by ProSAS for a critical event. To this end, it stores a list of recently requested critical events. Additionally, it preserves the newly added parameters to a watchlist (reported from the output of the step in Section 3.3.3). Similarly, it also stores recently removed parameters from a watchlist. For this step, ProSAS maintains a table (which is implemented as a cache and detailed in Section 4) with three attributes: *Recent Events*, *Recently Added*, and *Recently Removed*. We further demonstrate its usage through the following example.

Example 8 Table 5 shows an excerpt of the list with the updates on the watchlists for three recent critical events. The `Update Port` event is the most recent critical event, for which, 1257, 1421, and 1109 port IDs are recently added to the watchlist and 2311, 1765, and 1321 port IDs are recently removed from

the watchlist. Similarly, ProSAS stores the recent updates in the watchlists for the `Add Security Group Rule`, and `Delete Security Group Rule` critical events.

Recent Events	Recently Added	Recently Removed
Update Port	Port IDs: 1257, 1421, 1109	Port IDs: 2311, 1765, 1321
Add Security Group Rule	VM IDs: 1788, 2537, 1733	VM IDs: 1921, 2139, 1165
Delete Security Group Rule	VM IDs: 1921, 2139, 1165	VM IDs: 1788, 2537, 1733

TABLE 5: An excerpt of preserving recent actions for critical events

3.4.3 Utilizing Verification Actions/Results

The final step is to utilize these actions/results in taking proactive measures or runtime enforcement decisions, respectively, depending on the type of the current event (e.g., critical or non-critical).

Utilizing Verification Actions. If the current event is a non-critical event, then ProSAS utilizes the recent results for taking proactive actions. To this end, it first searches the name of the current event in the list, (if found) then records the action to be taken for this event, and finally executes that action using the steps described in Section 3.3.3. If the event is not found in the list of recent events, then ProSAS decides about the proactive actions by performing the steps described in Section 3.3.2. This step is further illustrated in Example 9.

Utilizing Verification Results. If the current event is a critical event, then ProSAS utilizes the recent results for taking runtime enforcement decisions. To this end, it first searches the currently requested event parameters in the recent updates of the watchlists for this event, and (if found) takes enforcement decisions (e.g., allow or deny) depending on its presence in the list of "recently added" or "recently removed", respectively. In case the requested parameter is not found in the recent results, then ProSAS checks the entire watchlist.

Example 9 For this example, we consider the recent results in Table 5 and three runtime events: `add security group rule (2537)`, `update port (1321)`, and `delete security group rule (2115)`. First, ProSAS allows the `add security group rule (2537)` event by checking the verification cache, as the VM ID 2537 is present in the recently added attribute. Second, ProSAS denies the `update port (1321)` event, as the `port ID 1321` is found in the recently removed list. Third, ProSAS requires to check further in the entire watchlist to verify the `delete security group rule (2115)` event, as the `VM ID 2115` is not in the recent results.

4 Implementation

This section describes how we integrate ProSAS into OpenStack.

4.1 Architecture

Fig. 8 shows a high-level architecture of ProSAS. ProSAS consists of four major components: dashboard & reporting engine, interceptor, learning engine, and verification engine. The dashboard & reporting engine provides an interface to ProSAS users. The interceptor is placed within the cloud as a middleware, in between the cloud dashboard or command line interface and different services (e.g., Nova, Neutron, Swift, etc. in OpenStack); which intercepts all tenant initiated events and forwards them to ProSAS, and enforces its security decisions (e.g., allow or deny).

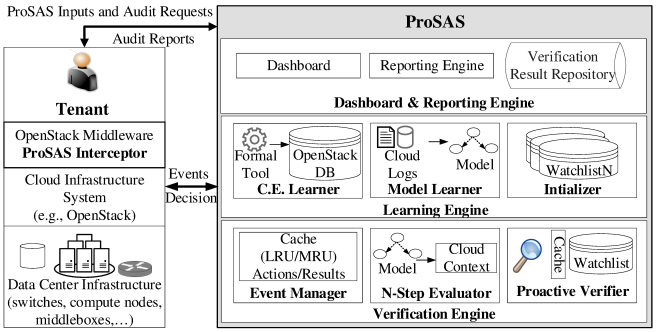

Fig. 8: A high-level architecture of ProSAS

The learning engine first learns critical events using a formal verification tool (e.g., Sugar [27]) on cloud configurations (e.g., OpenStack databases), then learns dependency models from the logs of different cloud services (e.g., Nova, Neutron, Swift, etc. in OpenStack) and Bayesian network and time-series model, and finally initializes all other inputs of ProSAS. In the verification engine, the event manager contains caches for both actions and results, *N*-step evaluator measures *N* for each critical event from the intercepted event using the current cloud context, and the proactive verifier queries the caches or watchlist databases to verify the parameters of the intercepted events.

4.2 Integration to OpenStack

This section discusses the implementation of ProSAS for OpenStack [17], which is an open-source cloud management platform.

Interceptor Middleware. The interceptor module, which is implemented in Python, intercepts operations based on the existing intercepting methods (e.g., audit middleware [41]) supported in OpenStack. We intercept event instances requested to the Nova service, as they are passed through the Nova pipeline, having the ProSAS middleware inserted in the pipeline. The body of requests, contained in the *wsgi.input* attribute of the intercepted requests, is scrutinized to identify the type of requested events. Also, we map all operations in OpenStack API [37] corresponding to the events that are relevant to the monitored security policies. Finally, the interceptor determines the criticality of the current event, and forwards the intercepted event details (e.g., type and parameters) to the following module.

Learning Engine. To learn the critical events (C.E.), the learning engine periodically invokes the formal verification tool (e.g., [10], [11]) to verify OpenStack configurations for the requested security policies. Whenever, the verification tool finds any violation, the C.E. learner collects event logs from the corresponding OpenStack service (e.g., Neutron, Nova and Swift). Finally, this engine filters out all system-initiated events (i.e., GET requests) and identify event type of other requests (i.e., PUT, POST and DELETE) based on its request body. Also, to learn the dependency model, the learning engine first utilizes Logstash [42], a data processing engine, and Python scripts to pre-process raw cloud logs and obtain event sequences and their frequency and intervals. Next, it leverages a Python Bayesian Network toolbox[1] to derive the probabilistic dependencies. Finally, the obtained Bayesian networks are provided to a time-series predictor, ARMAX [40], which is

1. https://pypi.org/project/pgmpy/

a widely used method in prediction of stochastic processes in various fields. Algorithm 1 shows the learning steps as follows.

Algorithm 1: Learning Engine

```
procedure LEARN C.E.(policies, interval, CloudOS)
    while true do
        for each policy p_i ∈ policies do
            results= verifyOffline(CloudOS, p_i)
            if results= "Violated" then
                deltaLog= collectLogs((currentTime-interval), p_i)
                Feedback_i= filterLogs(deltaLog)
                consultExpert(Feedback_i)
        Wait(interval)
procedure LEARN DEP. MODEL(Logs, Interval)
    for each timePeriod t_i ∈ T do
        bayesianNetwork = prepareBN(Logs_{t_i})
        Model = buildModel(bayesianNetwork, Intervals)
```

Event Manager. The event manager, which is mainly implemented as a cache in Python, preserves recent proactive actions/results. We implement two types of caching mechanisms: least recent update (LRU) and most recent update (MRU). Both cache memories are implemented as hash maps, and the management of caches is maintained using doubly linked list. Hash map maintains records of data in the form of key value pairs in which data is stored in value, and key is the hash value. If the value is found in the cache, then we utilize either the results or the actions stored on the cache to skip the step to check the entire watchlist, or figure out what proactive actions to perform, respectively. Furthermore, to handle the concurrent events, ProSAS leverages the Python library *EventQueue*[2] so that events are handled sequentially (in case). Algorithm 2 shows the functionalities of the event manager.

Algorithm 2: Event Manager

```
procedure BUILDCACHE(cache-type, event, cache-algo)
    if cache(cache-type) is full then
        removeCache(cache-type, cache-algo)
    updateCache(cache-type, event)
procedure SEARCHCACHE(cache-type, event)
    if event-type is "critical" then
        if event.type in cache & event.params in recently added then
            return "allow"
        else if event.type in cache & event.params in recently
removed then
            return "deny"
        else
            proactiveVerify(event, policies)
    else if event-type is "non-critical" then
        if event.type in cache & N-cp > N-th then
            return
        else if event.type in cache & N-cp <= N-th then
            performs actions mentioned in the cache
        else
            update-watchlist(WL, policies, event.params)
```

Proactive Verification Engine. Our proactive verification engine is mainly implemented in Python, and our tenant-specific watchlists are in a MySQL database, which allows us to efficiently query OpenStack cloud data. The initializer module first populates all watchlist tables from Neutron, Nova and Keystone databases; this step allows to capture the initial configurations into the watchlists. Algorithm 3 further details the steps of the verification engine.

2. https://m7i.org/tutorials/python-event-queue-concurrency-modeling/

Algorithm 3: Proactive Verification Engine

```
procedure BUILDWATCHISTS(CloudOS, policy-WL)
    for each policy p_i ∈ Policies do
        WL_i= initializeWatchlist(p_i, policy-WL, CloudOS)
procedure UPDATEWATCHISTS(WL, policy, parameters)
    updateWatchlist(WL, policy, parameters)

procedure EVALUATENSTEP(Event, Policies, Model)
    for each policy p_i ∈ Policies do
        Find N-th for p_i from policy-N-thresholds
        Find entities in Model related to p
        context = CollectCloudData()
        Find N-cp for p_i and context from Model-N-policy
        if N-cp=N-th then
            updateWatchlist(p_i, Event)
        else if N-cp < N-th and Event.type=WE then
            updateWatchlist(p_i, Event)

procedure PROACTIVEVERIFY(Event, policies)
    for each policy p_i ∈ policies do
        if Event.parameters in p_i.watchlist then
            Allow Event in the cloud
        else
            Deny Event in the cloud
```

Dashboard & Reporting Engine. 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 [34], CCM V3.0.1 [33], NIST 800-53 [36], etc.). Afterwards, security policies under the selected standards can be chosen. Apart from the proactive enforcement of compliance through the interceptor, the reporting engine of ProSAS provides a detailed report on recent intercepted events. Also, ProSAS dashboard provides a near real-time monitoring interface showing most recent user-initiated events and their corresponding verification decisions taken by ProSAS. Moreover, our reporting engine archives all the verification reports for a certain period. Fig. 9 shows screenshots of the ProSAS dashboard.

4.3 Adapting to Other Cloud Platforms

We design ProSAS in a platform-agnostic manner so that we can potentially adapt it to other major cloud platforms (e.g., OpenStack [17], Amazon EC2 [19], Google GCP [20] and Microsoft Azure [21]). The main adaption effort includes developing platform-specific interfaces to interact with the cloud platform (e.g., while collecting logs and intercepting runtime events) through two modules: log processor and interceptor. In the following, we elaborate on each of these efforts.

Building Interceptors. The responsibility of the interceptor is to intercept runtime event requests sent to a cloud platform. The interception mechanism may need to be implemented for each cloud platform. In OpenStack, we leverage the WSGI middleware to intercept and enforce the proactive auditing results so that compliance can be preserved. Through our preliminary study, we identified that almost all major platforms provide an option to intercept cloud events. In Amazon using AWS Lambda functions, developers can write their own code to intercept and monitor events. Google GCP introduces GCP Metrics to configure charting or alerting different critical situations. Our understanding is that our framework can be integrated to GCP as one of the metrics similarly as the *dos_intercept_count* metric, which intends to prevent DoS attacks. The Azure Event Grid is an event managing service from Azure to monitor and control event routing which

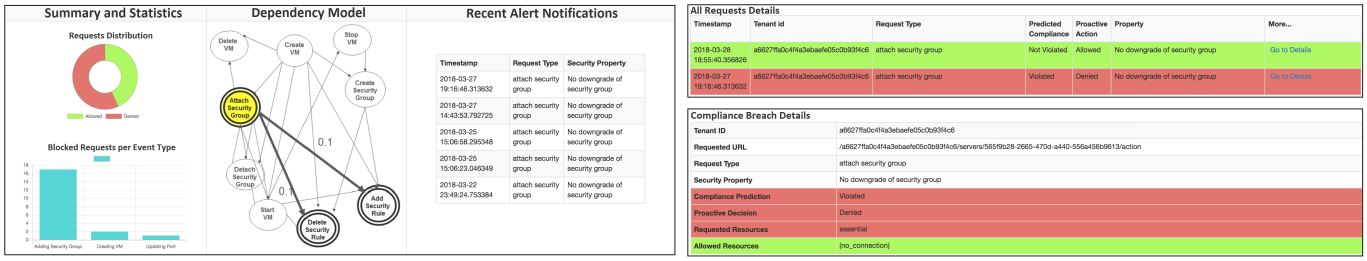

Fig. 9: Screenshots of the ProSAS dashboard

is quite similar as our interception mechanism. Therefore, we believe that our framework can be an extension of the Azure Event Grid to proactively audit cloud events. Table 6 summarizes the interception support in these cloud platforms.

Cloud Platform	Interception Support
OpenStack	*WSGI Middleware* [43]
Amazon EC2-VPC	*AWS Lambda Function* [19]
Google GCP	*GCP Metrics* [20]
Microsoft Azure	*Azure Event Grid* [21]

TABLE 6: Interception supports to adopt our framework in major cloud platforms

Developing Log Processors. The responsibility of the log processor is to interpret platform-specific event instances, and hence, is required to be implemented for each platform. First, to interpret platform-specific event instances to generic event types, we currently maintain a mapping of the APIs from different platforms. Table 7 enlists some examples of such mappings. The rest of the modules deal with the platform-independent data, and hence, the next steps in ProSAS are platform-agnostic.

Generic Event Type	OpenStack [17]	Amazon EC2-VPC [19]	Google GCP [20]	Microsoft Azure [21]
create VM	`POST /servers`	`aws opsworks -region create-instance`	`gcloud compute instances create`	`az vm create 1`
delete VM	`DELETE /servers`	`aws opsworks -region delete-instance -instance-id`	`gcloud compute instances delete`	`az vm delete`
update VM	`PUT /servers`	`aws opsworks -region update-instance -instance-id`	`gcloud compute instances add-tags`	`az vm update`
create security group	`POST /v2.0/security-groups`	`aws ec2 create-security-group`	N/A	`az network nsg create`
delete security group	`DELETE /v2.0/security-groups/{security_group_id}`	`aws ec2 delete-security -group -group-name`	N/A	`az network nsg delete`

TABLE 7: Mapping event APIs from different cloud platforms to generic event types

5 Experimental Results

This section first describes the experiment settings, and then presents the experimental results with both synthetic and real data.

5.1 Experiment Settings

In the following, we describe both testbed and real cloud settings.

Testbed Cloud Settings. Our testbed cloud OpenStack version is Mitaka with Keystone API version v3 and Neutron API version v2. There are one controller node and 80 compute nodes, each having Intel i7 dual core CPU and 2GB memory with the Ubuntu 16.04 server. Based on a recent survey [44] 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 experiments for 10 different datasets varying the most important factors and fixing others to the largest values, e.g., for the *no bypass* policy, 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 watchlist related to our example security policy contains a list of ports belonging to different tenants. For the *common ownership*[3] policy, the number of tenants is varied from 1,000 to 10,000 with the gap of 1,000 having five roles in each tenant. We repeat each experiment 100 times.

Real Cloud Settings. We further test ProSAS using data collected from a real community cloud hosted at one of the largest telecommunications vendors. To this end, we analyze the management logs (size more than 1.6 GB text-based logs) and extract 128,264 relevant log entries for the period of more than 500 days. As Ceilometer (which is the telemetry service of OpenStack) is not configured in this cloud, we utilize Nova and Neutron logs that increases the log processing efforts.

5.2 Experimental Results with Testbed Clouds

The objective of the first set of experiments is to measure the effect of the ProSAS result utilization on the response time. Fig. 10 shows the hit ratio (i.e., $\frac{\text{number of hits}}{\text{total number of tries}}$) and the effects of our result utilization applying two different caching mechanisms, i.e., least recent update (LRU) and most recent update (MRU), by varying the size of the cache. Fig. 10(a) illustrates the hit ratio for both LRU and MRU caches while increasing the size of the cache. Expectedly, the hit ratio increases with the size of the cache and reaches up to 0.93 for the 45,000 cache entries. Fig. 10(b) shows the average response time (in nanoseconds) required when there is a hit (i.e., intercepted event is found in the cache). In such cases, ProSAS responds in maximum 4,000 nanoseconds for the smallest cache size. Even though the response time for the MRU cache drops significantly for two cache sizes (10K and 25K), otherwise the response time for both cache types remains quite similar. Fig. 10(c) illustrates the delay (in nanoseconds) incurred to ProSAS due to a miss (i.e., intercepted event type is not present in the cache). The delay remains quite similar over the different cache sizes, and the maximum delay is 2,000 nanoseconds for the largest cache size. As similar as in Fig. 10(b), the cache with 25K entries results the lowest delay.

The second set of experiments is to compare the time required to process a user request individually by OpenStack and ProSAS. Fig. 11(a) shows the time (in seconds) to process different event

3. This policy allows users to hold only the roles that are defined within their domains [33], [34].

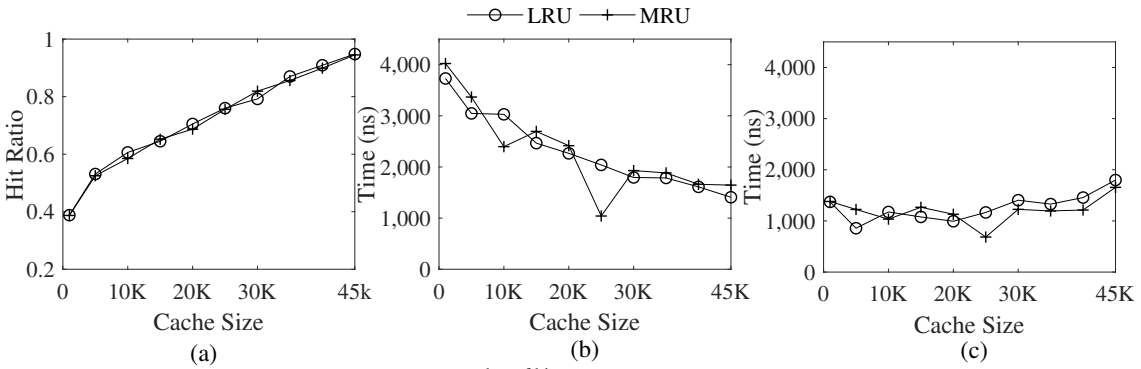

Fig. 10: Evaluation of result utilization: (a) hit ratio ($\frac{\text{number of hits}}{\text{total number of tries}}$) for both least recently used (LRU) and most recently used (MRU) caches; (b) average response time (in nanoseconds) by ProSAS when the intercepted event is found in the cache for both LRU and MRU caches; and (c) delay (in nanoseconds) caused by a miss for both LRU and MRU caches. In all cases, we vary cache size (in the number of entries) from 1,000 to 45,000, and verify the no bypass security policy.

types by OpenStack and ProSAS. Note that the processing time measured for OpenStack remains unaffected with or without ProSAS. The obtained results show that OpenStack requires seven to ten seconds to process the considered event types. In contrast, ProSAS takes maximum 0.0082 second to process the delete security group rule event type. We observe two major findings from this set of experiments. Firstly, Fig. 11(a) shows that ProSAS causes a negligible delay in comparison to the response time of OpenStack. Secondly, the period when OpenStack processes a request may be utilized to handle single-step violation without resulting a significant delay; which is considered as a potential future work.

The objective of the third set of the experiments is to measure the accuracy gain and time requirement of our input learning engine. Fig. 11(b) shows the number of new violations ProSAS catches over 100 days after introducing the learning engine in it for both *common ownership* and *no bypass* security policies. During the first two months, we observe the highest gain. In the last 20 days, there is no new security violation. Fig. 11(c) measures the time (in seconds) to learn critical events for the *common ownership* and *no bypass* security policies while varying the number of tenants up to 10,000. The critical event learning for the *common ownership* and *no bypass* policies takes maximum 5.55 seconds and 7.88 seconds, respectively, for our largest dataset. Note that the reported time only includes the time to perform automatic steps (Steps 1-3 in Section 3.2.1).

The fourth set of experiments is to demonstrate the time efficiency of our proactive verification steps. Intercepting operations to identify the type of operation, which is the minimum time we need to block for all operations (CE and WE, and all others), is taking constant time (0.266 ms) (INT in Fig. 12(a)). Moreover, calculating N-step (NSE in Fig. 12(a)) completes in constant time (0.133 ms for the largest datasets) for the *no bypass* (NB) policy, and in quasi constant time (varying from 0.773 ms to 0.794 ms) for the *common ownership* (CO) policy. The violation detector blocks only critical operations for a maximum delay of 8.2 ms (VD in Fig. 12(b)) for the largest dataset. Fig. 13(a) shows that pre-computing the watchlists for both *no bypass* and *common ownership* policies take 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. 13(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. On the other hand, the subsequent enforcement takes only 8 ms per critical operation call at run-time.

In the fifth part of the experiments, we measure the memory cost for the watchlists. Fig. 13(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 the watchlist carefully. Therefore, we show that our approach improves the execution time without excessive memory costs. We store role names and corresponding tenants for the common ownership policy, and only port IDs for the no bypass policy.

Finally, Table 8 compares the execution time of ProSAS and our alternative implementation of two existing *intercept-and-check* methods [13], [15] for the common ownership (P1) and no bypass (P2) security policies. Both policies show quite a similar nature. Weatherman [13] causes around five seconds for our largest dataset mainly due to its runtime effort on the entire cloud configuration. Majumdar et al. [15] adopt an incremental approach and take up to 182 milliseconds for the largest dataset. In contrast, ProSAS experiences maximum response time of 8.5 milliseconds. Furthermore, utilizing caching, the response time in ProSAS can be reduced to 3,000 nanoseconds on average (as reported in Fig. 10(b)).

5.3 Experimental Results with Real Clouds

Table 9 summarizes the obtained results for the real cloud dataset, which logs 5,279 event instances for the period of 506 days. For all experiments with real data, the cache size remains 25K, and we utilize the MRU caching technique. In these experiments, we measure the time for different steps of ProSAS and the hit ratio of the cache. Note that the obtained results are shorter due to the smaller size of the community cloud compared to our much larger simulated environment.

6 Discussions

In this section, we discuss different aspects of ProSAS.

Reliance on Cloud Management Platforms.

ProSAS is primarily designed and implemented for cloud management platforms (e.g., OpenStack). In the following, we

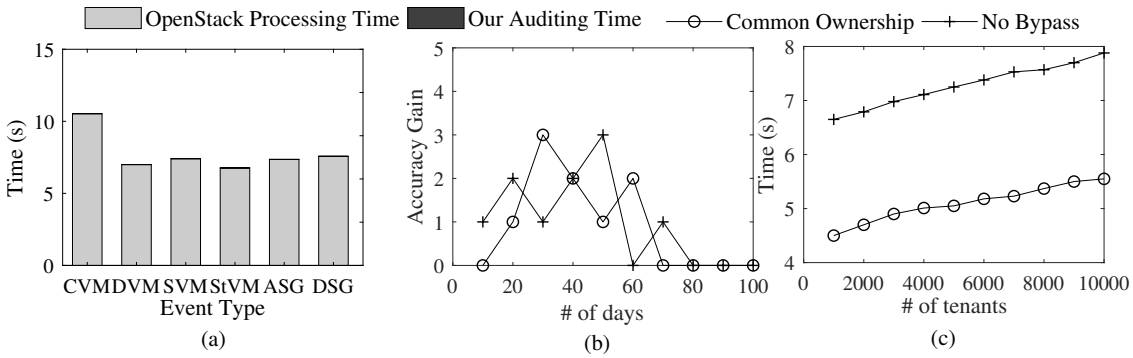

Fig. 11: (a) Time (in seconds) required to process different requests by OpenStack and ProSAS, (b) the accuracy gain (measured in terms of the number of new violations detected) by our learning engine over 100 days with the 10,000 tenants and (c) time (in seconds) required to learn a list of critical events while varying the number of events for the common ownership and no bypass security policies.

Policy	Number of Ports	10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000	90,000	100,000
P1	**Weatherman [13]**	585	817	1,392	1,801	2,415	3,112	3,823	4,231	4,546	5,177
	Majumdar et al. [15]	85	87	92	101	115	112	123	131	146	177
	ProSAS	5.928	6.09	6.916	7.016	7.496	7.815	8.024	8.14	8.453	8.501
P2	**Weatherman [13]**	578	785	1,129	1,711	2,289	3,443	3,837	4,545	4,991	5,555
	Majumdar et al. [15]	66	78	98	107	121	126	138	149	168	182
	ProSAS	5.63	6.23	6.66	6.99	7.11	7.15	7.52	7.64	8.03	8.2

TABLE 8: Comparing the execution time (in ms) between ProSAS and two existing *intercept-and-check* methods [13], [15] for the common ownership (P1) and no bypass (P2) security policies

Policies	Hit Ratio	Pre-Compute	Learning	Verification (W)	Verification (C)	Delay
No bypass	0.71	2,500ms	5,270ms	6.2ms	1,250ns	890ns
Common ownership	0.721	1,700ms	3,150ms	5.5ms	1,000ns	810ns

TABLE 9: Summary of the experimental results with real data. The reported delay is the additional time required in ProSAS verification for a miss in the cache. Note that Verification (W) and Verification (C) indicate the time required for verification through watchlist and verification through cache, respectively.

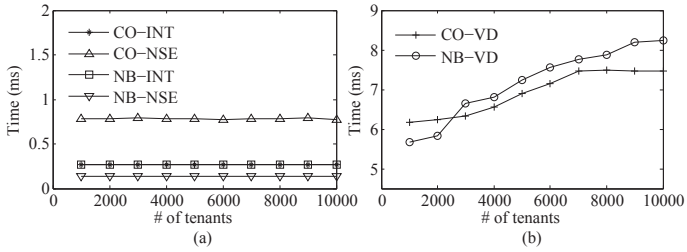

Fig. 12: Time duration (in ms) for different modules (INT: Interceptor, NSE: N-step evaluator, VD: Violation detector) of ProSAS for the *common ownership* (CO) and *no bypass* (NB) security policies by varying the number of tenants. The number of ports is also varied from 10,000 to 100,000, and each tenant contains five roles. Time required for the steps: (a) intercepting operations, evaluating N-step, and (b) detecting violations.

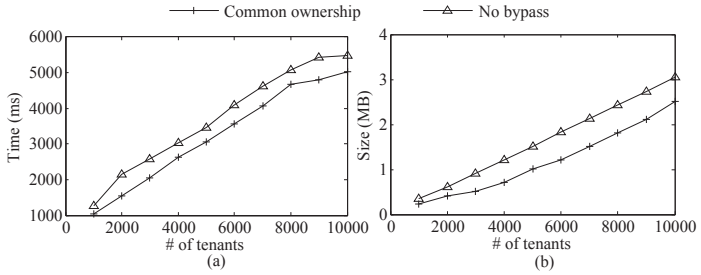

Fig. 13: (a) Time required (in ms) for preparing watchlist for different policies varying the number of tenants at the initialization step. (b) Memory requirement (in MB) for watchlists processing for different security policies by varying the number of tenants. Number of ports is also varied from 10,000 to 100,000 , and each tenant contains five roles.

discuss how different steps of ProSAS are specifically designed for the cloud context. First, all of our steps during learning inputs (in Section 3.2) are based on configurations and logs at the cloud management level; where the contents and formats of both configurations and logs are heavily dependent on cloud management platforms (e.g., OpenStack, Microsoft Azure). Consequently, the obtained list of critical events and dependency models from these steps only include cloud management operations. For instance, cloud configurations for different services (e.g., computing, networking) in OpenStack are stored in different databases (e.g., Nova, Neutron), respectively, and configurations for each cloud element (e.g., tenant, VM, network, subnet, etc.) are stored in various tables. Therefore, ProSAS collects and processes configurations from those various databases and their tables, and converts them into the format of the learning tools (e.g., CSP solver for offline auditing and Bayesian network for learning dependency). Additionally, logs in OpenStack are segregated among different services (e.g., computing, networking) and log entries are stored as a REST APIs (e.g., the log entry `POST /v2/servers HTTP/1.1` indicates the event type *create VM*). As a result, we obtain cloud-specific critical events (as shown in Tables 1 and 2) and dependency models (in Figure 5).

Second, all major steps for our proactive security verification (in Section 3.3) and verification result utilization (in Section 3.4) are based on runtime events at the cloud management level;

where both interception mechanisms and formats (e.g., event types and parameters) of runtime events are solely dependent on cloud management platforms. Consequently, watchlist contents are also in the cloud context where it stores various cloud elements (e.g., VM IDs, port IDs, etc.) depending on the security policies. Therefore, we require to design specific methods that can intercept, interpret, and block (if needed) runtime events to conduct proactive security verification. For instance, in OpenStack, ProSAS intercepts management operations (e.g., create VM, create tenant, add network) requested to different cloud services (e.g., Nova, Neutron), as they are passed through a service pipeline having the ProSAS middleware inserted in the pipeline. Additionally, ProSAS maintains a mapping between operations in OpenStack API (e.g., `POST /v2.0/ports.json HTTP/1.1`) and generic cloud event types (e.g., *create port*) to interpret intercepted events. As a result, our watchlists also contain cloud-specific data, e.g., port IDs except VM ports for the No bypass policy. Moreover, ProSAS enforces verification results (e.g., allow/deny) on OpenStack by leveraging the aforementioned middleware.

Challenges in Adapting to Other IT Systems. Even though the high-level idea of proactive security auditing in ProSAS might be potentially applicable to many other IT management systems, the research challenges involved in its each step are still not explored. For instance, the logging systems (including abstraction level of logging information and log formats) significantly vary depending on targeted systems. Additionally, dependency models and security policies along with their corresponding watchlist contents and critical events require to be identified for each system. Moreover, runtime operation interception and security enforcement depend on the design of a system. Furthermore, there might be additional unknown challenges for each system that need to be addressed to enable proactive security auditing in a system.

Effects of Change in Cloud Design. As our experiment results shown in Section 5, ProSAS can verify security policies for large size cloud in only few milliseconds at runtime. There could be certain cases where the pre-computed information used at runtime needs to be updated. For instance, when a change in the cloud dependency or in the cloud management API specifications occurs, or when including a new security policy, the ProSAS initialization must be repeated. Even though this re-initialization may take several minutes, this task can be executed in parallel with runtime verification. Also above-mentioned changes are not frequent by nature.

Correctness of Our Approach. The correctness of our auditing approach can be derived from the underlying constraint satisfaction problems (CSP) solving technique, which is a well-established formal verification technique whose correctness has been extensively discussed in the literature (e.g., [28], [29], [30], [31], [45]). Specifically, a CSP solver is guaranteed to produce sound results in the sense that any violation of a policy can be identified as long as the provided inputs contain supporting data of a violation [28], [45]. ProSAS leverages a CSP solver to learn critical events (in Section 3.2), which are later used in proactive security auditing to prevent future security breaches and any false positive or false negative resulted from those tools would be inherited and led to wrong or missing critical events, respectively. Particularly, using a CSP solver, ProSAS mainly performs incremental verification (which verifies security policies on a system data for a specific time period as described in Section 3.2.1). The incremental verification of a given security policy involves instantiating and solving the policy predicates for the affected elements in the supports of the involved relations. Therefore, any modification to the system data resulted from cloud events (e.g., *create VM*, *create tenant*, *grant role*, *delete role*, etc.) would not directly change the security policy 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. Similarly, if a role is deleted, the relations involving the entity representing the deleted role would have their supports decreased by the tuples including that role. Consequently, any modification to the system data (in the relation supports) only leads to an increase or decrease of the number of predicates instances to be solved to verify the entire system without affecting its soundness.

Supporting Operational Policies. The policies involving session/context specific data are not considered in this work. 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. Additionally, resolving any conflicts or other issues (e.g., ambiguous, incomplete) in a policy description is beyond the scope of this work and considered as a potential future work.

Dealing with One-Step Security Breaches. The proactive auditing mechanisms fundamentally leverage the dependency in a sequence of events. In other words, proactive security auditing is mainly to detect those violations which involve multiple steps. However, there might be violations of the considered security policies with a single step. Such violations cannot be detected by the traditional steps of proactive auditing with the same response time as reported in Figure 12(b), and may be detected by performing all auditing steps at a single point in several seconds (e.g., around six seconds for a decent-sized cloud with 10,000 tenants as shown in Figure 13(a)); which is still faster than any other existing works, e.g., [13], (which responds in minutes). However, this response time might not be very practical. To reduce the response time or at least not to cause any significant delay, we perform a preliminary study as follows. Our initial results conducted in the testbed cloud show that OpenStack takes more than six seconds to perform almost all user requests; which implies the possibility of not resulting in any additional delay by ProSAS even for a single-step violation. Additionally, during our case studies, we observed that OpenStack performs several system events to complete a user request. We may leverage this sequence of system events corresponding to a single user request to proactively perform ProSAS steps. We will elaborate those two ways of tackling single-step violations in our future work.

Choosing the Threshold Value. As presented in Section 3, ProSAS triggers the proactive verification when the probability (*N-cp*) from the current intercepted event to a critical event is greater than a threshold value (*N-th*). The ProSAS users (e.g., security experts and/or tenant admins) choose the value of *N-th*. Within the range of 0 to 1, choosing a lower value may allow ProSAS more time to perform the proactive verification. However, it may cause less accurate predictions (i.e., less predicted events will actually appear in the reality) and hence, less portion of the

precomputed results can be useful. On the other hand, choosing a higher *N-th* may allow a precise precomputed result. However, in that case, ProSAS may not get enough time to perform the proactive security verification step before the critical event occurs.

7 Related Work

In this section, we first compare existing solutions with ProSAS, and then discuss several categories of related works.

7.1 Comparison between Related Works

Table 10 summarizes the comparison between existing works and ProSAS. The first and second columns enlist existing works and their verification methods. The next two columns compare the coverage such as supported environment (cloud or non-cloud) and cloud layers (virtual infrastructure and/or user-level). We mark 'both', if a work supports both virtual infrastructure and user-level cloud layers. The next six columns compare these works according to different features. The *proactive* feature is checked when a solution supports proactive verification. When a solution enforces the verification results on the cloud at runtime, we check the *runtime enforcement* (runtime-enforce.) feature. The *utilizing-results* feature is checked, when a work optimizes its verification computation by storing previous results. The *queuing* feature refers to handling concurrent events for runtime solutions, and we mark this feature as 'N/A' for the works that do not support the runime feature. The *expressive* feature is checked for the works, which utilize well-known expressive policy languages (e.g., first order logic, and declarative logic) to express policies. The *model-learning* feature refers to the works which automatically establish their models. In the last four columns of the table, we compare the works based on their supporting cloud platforms. The *adaptable* field is checked if a work supports multiple cloud platforms or describes how it can be ported to other platforms.

In summary, ProSAS mainly differs from the state-of-the-art works as follows. Firstly, ProSAS is the first proactive auditing approach, which automatically learns its models (e.g., event dependencies and critical events). Secondly, ProSAS is the only proactive auditing solution, which utilizes the recent verification computations and results and reduces the response time to few hundreds nanoseconds. Thirdly, unlike other proactive solutions, ProSAS can handle concurrent events by maintaining an event queue. Finally, the ProSAS methodology is cloud-platform agnostic. However, there are still few limitations in ProSAS. ProSAS is less expressive than other general purpose formal verification approaches. ProSAS partially rely on an initial list of critical events provided by tenant admins or security experts. In the following, we discuss existing works from several related categories.

7.2 Cloud Security Auditing

In the following, we discuss the existing solutions in cloud security auditing under three major categories: retroactive, intercept-and-check, and proactive.

Retroactive Auditing Approach. Auditing security compliance in the cloud has recently been explored. For instance, Solonas et al. [51] detect illegal activities in the cloud only based on collected billing data in order to preserve privacy. In [10], [11], 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 [52] and NetPlumber [53] monitor network events and check network policies 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 policies (e.g., black holes and forwarding loops) in traditional networks, whereas our effort is oriented toward preserving compliance with structural security policies 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 [54]. CloudSec [55] and CloudMonatt [56] propose VM security monitoring. Our work covers a larger spectrum of policies (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 [57], 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 software; whereas, our contribution is at a higher level covering events initiated by potentially untrusted users.

Intercept-and-Check Approach. Existing intercept-and-check approaches (e.g., [13], [14]) perform major verification tasks while holding the event instances blocked, and usually cause significant delay to a user request. There are several other works (e.g., [52], [53]) monitoring network events and checking network policies at runtime. Weatherman [13] and OpenStack Congress [14] offer security verification of virtual infrastructure using the intercept-and-check approach. These works focus on operational network policies (e.g., black holes and forwarding loops) in traditional networks, whereas our effort is oriented toward preserving compliance with structural security policies that impact isolation in a virtualized infrastructure.

Proactive Auditing Approach. Existing proactive auditing approaches (e.g., [13], [14], [49] perform the major steps of an auditing in advance. Weatherman [13] and Congress [14] offer an offline proactive auditing approach, where the auditing process is performed on a future change plan provided by tenant admin. Unlike those works, LeaPS [49] and PVSC [22] automatically predict future critical events to conduct proactive auditing. However, both LeaPS and PVSC rely on manual identification of critical events and redo all proactive and runtime steps even for recurring events. Whereas, ProSAS overcomes all those limitations by learning its inputs (e.g., dependencies and critical events), and utilizing previous results.

7.3 Other Proactive Security Approaches

Proactive security analysis is 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 [58] models program monitors using an infinite-state-automata model to enforce safety policies. Those automata recognize invalid behaviors and halt the target application before the violation occurs.

Proposals	Methods	Coverage		Features						Supporting Platforms			
		Environment	Cloud Layer	Proactive	Runtime-Enforce.	Utilizing-Results	Model-Learning	Queuing	Expressive	OpenStack	Azure	VMware	Adaptable
Doelitzscher et al. [46]	Custom Algorithm	Cloud	Virtual Infr.	-	-	-	●	N/A	-	●	-	-	●
Ullah et al. [47]	Custom Algorithm	Cloud	Virtual Infr.	-	-	-	●	N/A	-	●	-	-	-
Majumdar et al. [11]	CSP Solver	Cloud	User-level	-	-	-	●	N/A	●	●	-	-	-
Madi et al. [10]	CSP Solver	Cloud	Virtual Infr.	-	-	-	●	N/A	●	●	-	-	-
Majumdar et al. [15]	CSP Solver	Cloud	User-level	-	●	-	●	-	●	●	-	-	-
Ligatti et al. [48]	Model Checking	Non-Cloud	N/A	●	●	-	-	●	●	N/A	N/A	N/A	N/A
PVSC [22]	Custom Algorithm	Cloud	Both	●	●	-	-	-	-	●	-	-	-
LeaPS [49]	Custom + Bayesian	Cloud	Both	●	●	-	-	-	-	●	-	-	●
Weatherman [13]	Graph-theoretic	Cloud	Virtual Infr.	●	-	-	●	-	-	-	-	●	-
Congress [14]	Datalog	Cloud	Both	●	-	-	-	●	-	●	-	-	-
Patron [50]	Custom Algorithm	Cloud	User-level	-	●	●	-	●	●	●	-	-	-
ProSAS	Custom Algorithm	Cloud	Both	●	●	●	●	●	○	●	-	-	●

TABLE 10: Comparing existing solutions with ProSAS. The symbols (●), (○), (-) and N/A mean fully supported, partially supported, not supported and not applicable, respectively. The (○) symbol is used when a work supports a feature partially, but less than other works that support the same feature fully.

Ligatti [59] 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 is proposed by Ligatti et al. [48], [60] that can transform both actions and results. Narain [61] proactively generates correct network configurations using the model finder Alloy. Our work further expands the proactive monitoring into cloud environments differing in scope and approach.

8 Conclusion

The continuous auditing with scalability and practical response time is important to both cloud providers and their tenants. In this paper, we proposed a proactive security auditing system, namely, ProSAS, which significantly reduces the response time and enforces the auditing results on the cloud before any violation can take effect. More specifically, ProSAS first established its models (e.g., dependency model and critical events). Second, it conducted proactive security verification by leveraging its models. Finally, it utilized those verification results to enforce security on the cloud at runtime. We integrated ProSAS to OpenStack, one of the most popular cloud management platforms, and provided guidelines to port it to other major cloud platforms. Furthermore, we evaluated the efficiency and accuracy of our method, and showed that the response time is reduced to a practical level. However, there exist several limitations in ProSAS, which we consider as future works. First, the current method of learning critical events needs a manual inspection, which could be further automated by using machine learning techniques to select the final candidate. Second, a single-step violation is not yet efficiently handled in ProSAS. An efficient runtime approach might help to address this concern.

Acknowledgement. We thank the anonymous reviewers for their valuable comments and suggestions. This work was supported partially by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under the Industrial Research Chair (IRC) in SDN/NFV Security. Additionally, the first author was supported by the start-up funds from University at Albany and Concordia University.

References

[1] J. Aikat, A. Akella, J. S. Chase, A. Juels, M. Reiter, T. Ristenpart, V. Sekar, and M. Swift, "Rethinking security in the era of cloud computing," *IEEE Security & Privacy*, vol. 15, no. 3, 2017.

[2] K. Ren, C. Wang, and Q. Wang, "Security challenges for the public cloud," *IEEE Internet Computing*, vol. 16, no. 1, pp. 69–73, 2012.

[3] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *ACM CCS*. ACM, 2009.

[4] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *ACM CCS*, 2014.

[5] Z. Xu, H. Wang, and Z. Wu, "A measurement study on co-residence threat inside the cloud." in *USENIX Security Symposium*, 2015.

[6] OpenStack, "Nova network security group changes are not applied to running instances," 2015, available at: https://security.openstack.org/ossa/OSSA-2015-021.html, last accessed on: February 14, 2018.

[7] ——, "Neutron security groups bypass through invalid CIDR," 2015, available at: https://security.openstack.org/ossa/OSSA-2014-014.html, last accessed on: February 14, 2018.

[8] Deloitte, "Cybersecurity and the role of internal audit," 2019, available at: https://www2.deloitte.com/us/en/pages/risk/articles/cybersecurity-internal-audit-role.html.

[9] KPMG, "Internal audit risk & compliance services," 2019, available at: https://home.kpmg/xx/en/home/services/advisory/risk-consulting/internal-audit-risk.html.

[10] 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 *ACM CODASPY*, 2016.

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

[12] Z. Ismail, C. Kiennert, J. Leneutre, and L. Chen, "Auditing a cloud provider's compliance with data backup requirements: A game theoretical analysis," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 8, pp. 1685–1699, 2016.

[13] S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim, "Proactive security analysis of changes in virtualized infrastructure," in *ACSAC*, 2015.

[14] OpenStack, "OpenStack Congress," 2015, available at: https://wiki.openstack.org/wiki/Congress. [Online]. Available: https://wiki.openstack.org/wiki/Congress

[15] S. Majumdar, T. Madi, Y. Wang, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi, "User-level runtime security auditing for the cloud," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1185–1199, 2018.

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

[17] OpenStack, "OpenStack open source cloud computing software," 2015, available at: http://www.openstack.org.

[18] ——, "Neutron firewall rules bypass through port update," 2015, available at: https://security.openstack.org/ossa/OSSA-2015-018.html.

[19] Amazon, "Amazon virtual private cloud," available at: https://aws.amazon.com/vpc, last accessed on: February 14, 2018.

[20] Google, "Google cloud platform," available at: https://cloud.google.com, last accessed on: February 14, 2018.

[21] Microsoft, "Microsoft Azure virtual network," available at: https://azure.microsoft.com, last accessed on: February 14, 2018.

[22] 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.

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

[24] M. Li, W. Zang, K. Bai, M. Yu, and P. Liu, "MyCloud: supporting user-configured privacy protection in cloud computing," in *ACSAC*, 2013.

[25] N. Schear, P. T. Cable II, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud," in *ACM CCS*, 2016.

[26] J. Schiffman, Y. Sun, H. Vijayakumar, and T. Jaeger, "Cloud verifier: Verifiable auditing service for IaaS clouds," in *IEEE SERVICES*, 2013.

[27] 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.

[28] M. Veksler and O. Strichman, "A proof-producing CSP solver." in *AAAI*, 2010.

[29] L. Zhang and S. Malik, "Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications," in *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2003, pp. 880–885.

[30] B. Beckert, R. Hahnle, and F. Manya, "The 2-SAT problem of regular signed CNF formulas," in *Proceedings 30th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2000)*. IEEE, 2000, pp. 331–336.

[31] B. Beckert, R. Hähnle, and F. Manya, "The SAT problem of signed CNF formulas," in *Labelled Deduction*. Springer, 2000, pp. 59–80.

[32] S. Majumdar, A. Tabiban, M. Mohammady, A. Oqaily, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi, "Proactivizer: Transforming existing verification tools into efficient solutions for runtime security enforcement," in *ESORICS*, 2019.

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

[34] 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.

[35] ——, "ISO 27002: 2005," *Information Technology-Security Techniques-Code of Practice for Information Security Management. ISO*, 2005.

[36] NIST, "SP 800-53," *Recommended Security Controls for Federal Information Systems*, 2003.

[37] OpenStack, "OpenStack command list," 2016, available at: http://docs.openstack.org/developer/python-openstackclient/command-list.html.

[38] S. Majumdar, A. Tabiban, Y. Jarraya, M. Oqaily, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "Learning probabilistic dependencies among events for proactive security auditing in clouds," *Journal of Computer Security*, vol. 27, no. 2, pp. 165–202, 2019.

[39] L. Foundation, "Open vSwitch," 2018, available at: https://www.openvswitch.org.

[40] J. D. Hamilton, *Time series analysis*. Princeton New Jersey, 1994, vol. 2.

[41] OpenStack, "OpenStack audit middleware," 2016, available at: http://docs.openstack.org/developer/keystonemiddleware/audit.html.

[42] Elasticsearch, "Logstash," available at: https://www.elastic.co/products/logstash, last accessed on: February 14, 2018.

[43] WSGI, "Middleware and libraries for WSGI," 2016, available at: http://wsgi.readthedocs.io/en/latest/libraries.html, last accessed on: February 15, 2018.

[44] OpenStack, "OpenStack user survey," 2015, available at: https://www.openstack.org/assets/survey/Public-User-Survey-Report.pdf.

[45] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara, "Compiling finite linear csp into sat," *Constraints*, vol. 14, no. 2, pp. 254–272, 2009.

[46] F. Doelitzscher, C. Fischer, D. Moskal, C. Reich, M. Knahl, and N. Clarke, "Validating cloud infrastructure changes by cloud audits," in *IEEE SERVICES*, 2012.

[47] K. W. Ullah, A. S. Ahmed, and J. Ylitalo, "Towards building an automated security compliance tool for the cloud," in *IEEE TrustCom*, 2013.

[48] J. Ligatti and S. Reddy, "A theory of runtime enforcement, with results," in *ESORICS*, 2010.

[49] S. Majumdar, Y. Jarraya, M. Oqaily, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "LeaPS: Learning-based proactive security auditing for clouds," in *ESORICS*, 2017.

[50] Y. Luo, W. Luo, T. Puyang, Q. Shen, A. Ruan, and Z. Wu, "OpenStack security modules: A least-invasive access control framework for the cloud," in *IEEE CLOUD*, 2016.

[51] M. Solanas, J. Hernandez-Castro, and D. Dutta, "Detecting fraudulent activity in a cloud using privacy-friendly data aggregates," arXiv preprint, Tech. Rep., 2014.

[52] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: verifying network-wide invariants in real time," in *USENIX NSDI*, 2013.

[53] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *USENIX NSDI*, 2013.

[54] 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.

[55] 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 *NSS*, 2011.

[56] 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.

[57] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *IEEE S&P*, 2008.

[58] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 2000.

[59] J. Ligatti, L. Bauer, and D. Walker, "Run-time enforcement of nonsafety policies," *ACM Transactions on Information System Security (TISSEC)*, 2009.

[60] E. Dolzhenko, J. Ligatti, and S. Reddy, "Modeling runtime enforcement with mandatory results automata," *International Journal of Information Security*, 2014.

[61] S. Narain, "Network configuration management via model finding," in *LISA*, 2005.

Suryadipta Majumdar Suryadipta Majumdar is currently an Assistant Professor in Concordia Institute for Information Systems Engineering (CIISE), Concordia University, Montreal, Canada. Previously, Suryadipta was an Assistant Professor in the Information Security and Digital Forensics department at University at Albany - SUNY, USA. He received his Ph.D. on cloud security auditing from Concordia University. His research mainly focuses on cloud security, Software Defined Network (SDN) security and Internet of Things (IoT) security.

Gagandeep Singh Chawla Gagandeep Singh Chawla is currently working as Security Architect with SAP Security, Montreal, Canada. He received his master's degree in information system security from Concordia University, Montreal, Canada. His research interests include security of public/private clouds, containers, container orchestration platforms and Software Defined Networks (SDN).

Amir Alimohammadifar Amir Alimohammadifar completed his BSc in Information Technology in 2010. He received his first master's in Information Technology, communication and computer networks from Sharif University of Technology, 2012, and his second master's in Information Systems Security from Concordia University.

Taous Madi Taous Madi is currently an Experienced Researcher at Ericsson Canada. She holds a Ph.D. in Information Systems Engineering from Concordia University, Montreal. Her research interests include network function virtualization security, software-defined networking security, internet of things security, security metrics, machine learning and formal verification. She has co-authored a book and several conference and journal articles at reputable cybersecurity venues.

Yosr Jarraya Yosr Jarraya is currently a researcher in security at Ericsson focusing on security and privacy in cloud, SDN, and NFV. Previously, she was awarded a postdoctoral fellowship at the same company. Before that, she was a Research Associate at Concordia University, Montreal. She received a Ph.D. in Electrical and Computer Engineering from Concordia University. She has several patents granted or pending. She co-authored two books and more than 40 research papers on topics including cloud security, data anonymization, network and software security, formal verification, and SDN.

Makan Pourzandi Makan Pourzandi is a research leader at Ericsson, Canada. He received his Ph.D. degree in Computer Science from University of Lyon I Claude Bernard, France and M.Sc. in parallel computing from École Normale Supérieure de Lyon, France. He is the co-inventor of 19 granted US patents and more than 65 research papers in peer-reviewed scientific journals and conferences. His current research interests include security, cloud computing, software security engineering, cluster computing, and component-based methods for secure software development.

Lingyu Wang Lingyu Wang is a Professor at the Concordia Institute for Information Systems Engineering (CIISE) at Concordia University, Montreal, Canada. He holds the NSERC/Ericsson Senior Industrial Research Chair in SDN/NFV Security. He received his Ph.D. degree in Information Technology in 2006 from George Mason University. His research interests include cloud computing security, SDN/NFV security, security metrics, software security, and privacy. He has co-authored five books, two patents, and over 120 refereed conference and journal articles at reputable venues including TOPS, TIFS, TDSC, TMC, JCS, S&P, CCS, NDSS, ESORICS, PETS, ICDT, etc.

Mourad Debbabi Mourad Debbabi is a Full Professor at the Concordia Institute for Information Systems Engineering and Interim Dean at the Gina Cody School of Engineering and Computer Science. He holds the NSERC/HydroQuebec 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 a member of CATAAlliance's Cybercrime Advisory Council. He serves/served on the boards of Canadian Police College, PROMPT Québec and Calcul Québec. He is the founder and one of the leaders of the Security Research Centre at Concordia University. Dr. Debbabi holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay, University, France. He published 5 books and more than 300 peer-reviewed research articles in international journals and conferences on cyber security, cyber forensics, smart grid, privacy, cryptographic protocols, threat intelligence generation, malware analysis, reverse engineering, specification and verification of safetycritical systems, programming languages and type theory. He supervised to successful completion 32 Ph.D. students, 76 Master students and 14 Postdoctoral Fellows. 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.