Proactive Security Auditing for Clouds

Suryadipta Majumdar

A thesis
in
The Concordia Institute
for
Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Information and Systems Engineering) at
Concordia University
Montréal, Québec, Canada

May 2018

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Suryadipta Majumdar**
Entitled: **Proactive Security Auditing for Clouds**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Information and Systems Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________ Chair
Dr. Ion Stiharu

______________________ External Examiner
Dr. Xiaodong Lin

______________________ External to Program
Dr. Otmane Ait Mohamed

______________________ Examiner
Dr. Chadi Assi

______________________ Examiner
Dr. Jeremy Clark

______________________ Thesis Supervisor
Dr. Lingyu Wang

Approved by ______________________
Dr. Chadi Assi, Graduate Program Director

July 24, 2018 ______________________
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Proactive Security Auditing for Clouds

Suryadipta Majumdar, Ph.D.
Concordia University, 2018

Cloud computing is emerging as a promising IT solution for enabling ubiquitous, convenient, and on-demand accesses to a shared pool of configurable computing resources. However, the widespread adoption of cloud is still being hindered by the lack of transparency and accountability, which has traditionally been ensured through security auditing techniques. Security auditing in the cloud poses many unique challenges in data collection and processing (e.g., data format inconsistency and lack of correlation due to the heterogeneity of cloud infrastructures), and in verification (e.g., prohibitive performance overhead due to the sheer scale of cloud infrastructures and need of runtime verification for the dynamic nature of cloud). To this extent, existing security auditing solutions can mainly be categorized into three types: retroactive, intercept-and-check and proactive. The retroactive auditing approach is the traditional auditing technique, which audits after the fact and cannot prevent irreversible damages (e.g., leakage of sensitive information and denial of service attacks). The intercept-and-check approach offers runtime auditing and performs all the auditing steps after the occurrence of a critical event (i.e., which may potentially violate a security property). However, this approach results significant delay in responding each critical event. On the other hand, the existing proactive approach requires the changes (in the cloud configurations) planned for the future in advance to verify its compliance; however, this approach is not practical, because the future change plan is not always available due to cloud's dynamic and ad-hoc nature. In this thesis, we address all the above-mentioned limitations of the existing works by proposing a proactive security auditing system, which potentially can prevent irreversible damages, respond in significantly less time and offer a practical approach without requiring any future change plan. To this purpose, we conduct our work into three main phases. During the first phase, we propose a runtime security auditing system for the user-level of the cloud; where our proposed system audits wide range of security properties relevant to different authentication and authorization mechanisms, such as role-based access control (RBAC), attribute-based access control (ABAC) and single sign-on (SSO), and enhances the existing intercept-and-check solutions by adopting an incremental approach to improve the efficiency. In the second phase of our work, we propose a novel approach of proactive security auditing; which leverages the dependency relationship among cloud events and pre-computes the most expensive parts of the auditing process to keep the response time of the solution to a practical level. In our final phase, we utilize learning techniques to automatically capture these probabilistic dependency relationships, and propose an automated log processing approach to prepare the raw logs collected from cloud deployments for these learning methods to significantly enhance the practicality of our proactive security auditing system. Also, to demonstrate the applicability, scalability and efficiency of our proposed system, we integrate it to OpenStack, a major cloud platform, and evaluate it using both synthetic and real data. In summary, this thesis contributes towards enhancing security, efficiency and practicality of security auditing in the cloud environment.

Acknowledgments

This thesis work is the outcome of collaboration and support of many people, to whom I am sincerely grateful and would like to appreciate their help.

At the very beginning, I would like to thank my PhD supervisor Dr. Lingyu Wang. His continuous availability and guidance helped me the most to complete this thesis work. I am grateful to him for enlightening me with his profound knowledge and precise insights. His constructive criticism greatly helped me to improve my research skills throughout my PhD study.

I would also like to thank Dr. Mourad Debbabi for his mentorship during my PhD. In spite of his busy schedule, he always manages his time to listen to my various issues and to provide very practical solutions to them. I am grateful to the members of my PhD examination committee, Dr. Xiaodong Lin, Dr. Otmane Ait Mohamed, Dr. Chadi Assi and Dr. Jeremy Clark, for their insightful advice during different phases of this work. I also thank all other faculty members of the CIISE department, specially Dr. Mohammad Mannan, my master's supervisor, who continued his support to me during my PhD.

My heartfelt gratitude extends to all members of the Audit Ready Cloud group, specially Dr. Makan Pourzandi, Dr. Yosr Jarraya, Taous Madi, Yushun Wang, Amir Alimohammadifar, Momen Oqaily, Azadeh Tabiban and Gagandeep Singh Chawla, with whom I collaborated throughout my PhD study. I also acknowledge the financial support of NSERC, Ericsson Canada, Prompt Quebec and Concordia University.

At the end, I would like to acknowledge the unconditional affection and continuous support of my parents, sister and nephew in my life. They always fulfill all my silly demands and keep faith on my ability. They have always been the best inspiration in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Cloud computing has been gaining momentum as a promising IT solution specially for enabling ubiquitous, convenient, and on-demand accesses to a shared pool of configurable computing resources. From small to large sized companies nowadays leverage the cloud service for conducting their major operations (e.g., web service, inventory management, customer service, etc.). Based on the provided services, cloud computing has been divided into different categories such as infrastructure as a service (IaaS), platform as a service (PaaS) and hardware as a service (HaaS). In most of the categories, there exist at least three main stakeholders: cloud service providers, tenants and their users.

A cloud service provider owns a significant amounts of computational resources, e.g., servers, and offers different paid services (e.g., IaaS, PaaS, etc.) to its customers by utilizing this pool of resources. A cloud tenant, the direct customer of cloud providers, enjoys the ad-hoc and elastic (i.e., allocating/deprovisioning based on demands) nature of cloud to use the shared pool of resources for conducting its necessary operations. Usually, tenants are different companies or departments within a company. A user being a customer of a cloud tenant mainly avails different services offered by a tenant. Thus, by providing a dynamic (i.e., ever changing) and measured services (i.e., "pay as you go") to its users and tenants, cloud computing has become a popular choice for diverse business models in recent years.

While cloud computing has seen such increasing interests and adoption, the fear of losing control and governance still persists due to the lack of transparency and trust [98]. Security auditing and compliance validation may increase cloud tenants' trust in the service providers by providing assurance on the compliance with the applicable laws, regulations, policies, and standards. However, there are currently many challenges in the area of cloud auditing and compliance validation. For instance, there exists a significant gap between the high-level recommendations provided in most cloud-specific standards (e.g., Cloud Control Matrix (CCM) [18] and ISO 27017 [53]) and the low-level logging information currently available in existing cloud infrastructures (e.g., OpenStack [89]). In practice, limited forms of auditing may be performed by cloud subscriber administrators [84], and there exist a few automated compliance tools (e.g., [25, 108]) with several major limitations, which are discussed later in this section.

Furthermore, the unique characteristics of cloud computing may introduce additional complexity to the task, e.g., the use of heterogeneous solutions for deploying cloud systems may complicate data collection and processing and the sheer scale of a cloud, together with its self-provisioning, elastic, and dynamic nature, may render the overhead of many verification techniques prohibitive. In particular, the multi-tenant and self-service nature of clouds usually implies significant operational complexity, which may prepare the floor for misconfigurations and vulnerabilities leading to violations of security compliance. Therefore, the security compliance verification with respect to security standards, policies, and properties, is desirable to both cloud providers, and its tenants and users. Evidently, the Cloud Security Alliance (CSA) has recently introduced the Security, Trust & Assurance Registry (STAR) for security assurance in clouds, which defines three levels of certifications (self-auditing, third-party auditing, and continuous, near real-time verification of security compliance) [19]. However, above-mentioned complexities coupled with the sheer size of clouds (e.g., a decent-size cloud is said to have around 1,000 tenants and 100,000 users [2]) implies one of the main challenges in cloud security auditing. In summary, the major challenges are to handle the unique nature of cloud and to deal with the sheer size of cloud in providing a scalable and efficient security auditing solution for clouds.

To this end, existing approaches can be roughly divided into three categories (a more detailed review of related work will be given in Section 2.2). First, the retroactive approaches (e.g., [25, 108]) catch compliance violations after the fact by verifying different configurations and logs of the cloud. However, 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., [16, 88]) verify the compliance of each user request before either granting or denying it, which may lead to a substantial delay to user requests. Third, the proactive approach in [16, 88] verifies future change plan to identify any potential breach from the proposed plan. However, due to the dynamic and ad-hoc nature of clouds, providing future change plan in advance is not always feasible, and hence, this approach is not practical for clouds. In conclusion, existing works suffer from at least one of the following limitations: i) supporting a very limited set of security properties, ii) responding with a significant delay, and iii) involving unrealistic amounts of manual efforts.

1.2 Problem Statement

In this thesis work, we mainly address the aforementioned limitations of the existing cloud auditing solutions. To this end, we focus on providing a practical security auditing system for clouds, which significantly improves the existing state-of-the-art over at least three dimensions, i.e., security, efficiency and practicality. In particular, this thesis work mainly answers the following research questions:

1. How can we audit different security properties that are important to cloud tenants to ensure the accountability and transparency of cloud providers?

2. How can we provide an efficient runtime auditing solution with practical response time?

3. How can we automate different auditing steps to ensure better scalability, correctness and usefulness of the auditing tools?

We elaborate the aforementioned problems in the following.

1.2.1 Runtime Security Auditing for Clouds

During the first phase of our work, we target providing a continuous security auditing solution for the user-level (e.g., authentication and authorization) (which can potentially be adapted to other layers) of the cloud. To this purpose, our main focuses are to address the limitations of the existing solutions as well as to overcome the aforementioned challenges in the area of cloud security auditing. More specifically, we explore the off-the-shelf verification methods and alternatively consider customized verification algorithms to tackle the sheer scale of cloud (e.g., a large-size cloud is said to have around 10,000 tenants and 100,000 users [92]), together with its self-provisioning, elastic, and dynamic nature, which potentially render the overhead of runtime verification process prohibitive. Additionally, our work intends to bridge the gap between the high-level recommendations provided in most cloud-specific standards (e.g., Cloud Control Matrix (CCM) [18] and ISO 27017 [53]) and the low-level logging information currently available in existing cloud infrastructures (e.g., OpenStack [89]), and identify a wide range of security properties to mainly cover the user-level of the cloud. Furthermore, designing the data collection and pre-processing steps specifically to handle the use of heterogeneous solutions in a cloud system. Chapter 3 details how our runtime security auditing approach works.

1.2.2 Proactive Security Auditing for Clouds

In the second phase of this work, our target is to overcome the significant delay in all existing works and respond in a very practical time to audit a cloud at runtime. More specifically, after a careful observation, we identify one of the main reasons behind the inefficiency of existing intercept-and-check approaches (e.g., [16]) is that all the auditing steps are performed at a single point (a.k.a. critical event [1]) and not taking any advantage of the dependencies. Alternatively, leveraging dependencies (if any) among cloud events potentially may allow starting the auditing process in advance and conduct auditing incrementally. Therefore, in this work we explore the possibility of deriving the relationship (e.g., dependency) among cloud events and devise a proactive auditing technique leveraging those relationships. Chapter 4 elaborates our idea of

[1]The event type which potentially can violate a security property.

proactive security auditing approach.

1.2.3 Learning-Based Proactive Security Auditing

During the final phase, our main target is to strengthen the security guarantee and improve the practicality of our proactive auditing approach to offer a practical proactive security auditing system which potentially can be integrated to a popular cloud platform (e.g., OpenStack). To this end, our first objective is to capture various relationships (e.g., dependency) among cloud events to support wider range of security properties. Our second objective is to avoid error-prone and tedious manual efforts involved with the auditing process, and to contribute towards the automation of the auditing process. Chapter 5 further describes our idea on achieving both aforementioned goals, and provides the details of our learning-based proactive security auditing approach.

In summary, the three phases of this PhD research address the security, efficiency, and practicality aspects of security auditing in clouds, respectively. Those topics are complementary to each other, and the following details the link between them, and how they were identified. We start by addressing the security concerns of cloud tenants by enabling runtime auditing for them. A key challenge emerged during this first research is that the sheer size of cloud renders the overhead of runtime auditing prohibitive, and hence, the response time may not be practical. Therefore, in the second research, we address this limitation through a proactive auditing approach, which leverages the dependency relationship among cloud events. One key observation during this second research is that the dependency model must be manually created; which may turn to be an error-prone and tedious task especially considering the sheer size of cloud. Therefore, in the third research, we address this limitation and fully automate the dependency capturing step along with its pre-requisite steps (e.g., log processing) to offer a practical proactive auditing system.

1.3 Contributions

The main contributions of this thesis work are towards security, efficiency and practicality improvements in cloud security auditing. To this end, we propose a proactive security auditing system, which bridges the gap between high-level standards and low-level cloud logs and configurations, supports a wide range of security properties to audit both user and virtual infrastructure levels in the cloud, significantly reduces the response time of runtime auditing, and offers a fully automated auditing solution for clouds. We elaborate each contribution as follows.

First, this thesis concentrates on filling in the gap between standards and cloud implementations. To this end, we first study the major cloud specific standards (e.g., CCM [18], ISO27017 [53] and NIST 800-53 [81]). Then, we prepare a list of security properties, which mainly covers the security of both user and virtual infrastructure levels in the cloud. Next, we investigate different configurations and log files in the cloud to propose a uniform solution which can support various formats of those files. Finally, we transform the description of these security properties based on the real cloud implementations and their supporting configurations and logs.

Second, we propose a runtime security auditing system specially designed for the user-level of the cloud. More specifically, the user-level auditing is supported by covering a wide range of security properties from most popular access control mechanisms, such as role-based access control (RBAC) and attribute-based access control (ABAC), and major authentication plugins, such as single sign-on (SSO). The runtime auditing is offered by performing the expensive auditing operations during the one-time initialization phase and by keeping the runtime operations incremental and light-weight so that our proposed system can achieve a realistic practical time (e.g., the response time is less than 500 milliseconds for a large cloud with 100,000 users).

Third, we propose a proactive security auditing approach for both user and virtual infrastructure levels of the cloud. Note that the proactive security auditing approach is a complementary to the runtime security auditing as will be discussed in Section 4.3. In this work, we first capture the structural dependencies (which are mainly imposed by the cloud implementations) by studying the cloud platform specifications. Then, we utilize this dependency model to pre-compute expensive parts of the auditing incrementally. Finally, at a critical event (i.e., events that potentially can violate the security properties), our proposed solution simply checks the pre-computed results to verify any security property and responds to the request very quickly (e.g., 8.5 milliseconds to verify 100,000 virtual ports).

Finally, we automate major proactive auditing steps and integrate them into a popular cloud management platform (e.g., OpenStack) to offer a proactive security auditing system for clouds. For instance, we automate the dependency capturing step by leveraging different learning mechanisms (e.g., Bayesian network and structural pattern mining). We also propose a log processing approach, which automatically prepares the raw logs of clouds for the learning tools. Furthermore, we integrate our proposed system into OpenStack [89], one of the major cloud management platforms, and conduct experiments to measure the efficiency, scalability and applicability of this system.

In summary, main contributions of this thesis work are as follows.

- As per our knowledge, we are the first to propose a runtime security auditing for user-level of the cloud which verifies security properties covering important authentication and authorization mechanisms, such as RBAC, ABAC and SSO.

- We are the first to propose the concept of dependency-based proactive security auditing approach for clouds which potentially can reduce the response time of runtime auditing to a practical level.

- We are also the first to capture both structural and behavioral dependencies among cloud events which potentially can be leveraged in different security solutions to enhance the efficiency of those systems.

- We integrate our proposed auditing solution into OpenStack [89], a major cloud platform, and evaluate it with both synthetic and real data; the results of which show scalability, efficiency and applicability of our approach.

1.4 Thesis Structure

This thesis is organized into six chapters as follows. Chapter 2 provides background on the traditional security auditing steps and discusses the major auditing approaches through related works. Chapter 3 presents our runtime security auditing system for the user-level of the cloud, where we further elaborate our motivation and problem, then describe the methodology of our approach, and finally present implementation details and experimental results. Chapter 4 details the design and implementation of our proactive security auditing approach with its experimental evaluation with both synthetic and real data. In Chapter 5, we elaborate the methodology and implementation of the steps involved with learning the probabilistic dependencies and processing raw logs from real cloud environments along with the evaluation of how above-mentioned steps can enhance the auditing process. Chapter 6 concludes this thesis with the summary of this work and discussion on potential future works.

Chapter 2

Background

2.1 Security Auditing for Clouds

Security compliance auditing is in practice for years. However, the growing popularity and underlying design features in the cloud, revive the need of security compliance of auditing with newer challenges e.g., scalability and practical response time. In this section, we first describe different categories of security auditing in cloud and then, explain different steps of a traditional security auditing process.

2.1.1 Categorization of Cloud Security Auditing

Generally, there are two categories of cloud security auditing: 1) offline auditing and 2) online auditing. Offline auditing is performed on the snapshots (e.g., configurations and logs) collected from a real cloud deployment or the simulated model of a real cloud. Online auditing is performed on the cloud at runtime or near runtime. The basic goal of the offline auditing is to detect security violations either after the fact or to verify a future change plan. Whereas, the online auditing either prevents a violation, or detects them as soon as they occur (i.e., with a very little delay). In the following, we provide a brief description along with their limitations. A detailed literature review is presented in Section 2.2.

Furthermore, the offline auditing approaches can be divided into two types: retroactive auditing and proactive auditing. The retroactive auditing is performed periodically (e.g., daily, weekly and monthly) irrespective to the occurrence of any event or security violation. This form of auditing is more traditional, however, the fundamental problem with this approach is that it cannot prevent any irreversible damages, e.g., DoS attack and leaking sensitive information. The existing proactive approaches verify future change plans and simulate them on the devised model of the real cloud. The implicit assumption is that the future change plan is known in advance so that at first the plan can be verified against security properties, and then be applied.

On the other hand, the online auditing approaches interact with the cloud system and identify security violations at real-time or near real-time. Near real-time (e.g., [14]) approaches detect security violations with a little delay. On the other hand, real-time auditing solutions (e.g., [61]) are inline with the continuous monitoring to detect any violation right away. The response time plays a critical role in adopting this approach.

2.1.2 Structure of the Automated Security Auditing Process

Though security auditing is not a new process, automation of this process and complexity of targeted infrastructures introduce non-trivial challenges. Manual auditing is still in practice, where internal or third party auditors conduct the auditing process based on the collected data/evidence. Initial approaches of automating the auditing process are mostly to detect network intrusions. Later it has been adapted in other domains, such as data systems, access control and distributed systems. One of the most recent additions in the list is the cloud infrastructure. Based on the proposed solutions and best practices, we identify different phases of an automated security auditing process.

Defining the Scope and Threat Model. As a very first step, an organization should define the scope of its auditing. Part of it is to identify the critical and sensitive assets, operations and the modules in the system that deal with those assets and operations. The following step is to identify threats or nature of threats to be considered for the auditing process. Most of the time, threat model depends on the nature of the business and demand of customers. Part of this step is to describe security assumptions while considering each threat. To this end, last few years different studies have been conducted to identify risks and threats in the cloud computing ecosystem. Based on those threats, several security properties are proposed by CSA [17], ENISA [28], ISO [53], NIST [81], CUMULUS [20], etc.

Data/evidence Collection. The next phase is to gather evidences/data to conduct the audit process. Based on the target system and threat model, audit data is enlisted. In some cases (e.g., cloud and distributed systems), locating those audit data is non-trivial.

The data collection phase has become more dynamic with the virtualization and multi-tenancy; which results an increase in the amount of data to be collected. We also consider security aspects of data collection in addition to the different runtime and continuous data collection techniques of different data types. The trust model ensures that the audit data provided by a tenant is real and fresh. At the same time, there might exists the privacy concerns in a central auditing system, such as any tenant must not leak any sensitive information to the auditor, which can benefit any other tenants in case of colluding with the auditor.

In the cloud, most of the audit data is any of events, logs and system configurations. Different collection techniques vary each other in terms of targeted environments and data, e.g., what data to collect based on the scope, threat model and objectives, and how to collect data (more challenging in a cloud-based system).

Data/evidence Processing. The previous step collects raw data from the system. It requires further processing to be able to conduct auditing. In case of verifying compliance with a policy language, it depends on the language. Collected data needs to be sanitized, as data is collected from different sources. For better understanding and interpretation, different correlation methods are applied on sanitized data to categorize them. There are different techniques (e.g., call graph, information flow graph, reachability graph) to represent the audit data. Heterogeneous data is normalized by different methods, e.g., [23]. Storing this processed audit data is also an important phase specially when dynamic cloud auditing generates enormous amount of data over time.

Auditing. In the auditing phase, processed data is verified against the policies for any violation. The process either validates the system or detects if any anomaly exists. There are different auditing techniques proposed over time, though comparatively less automated techniques exist for the cloud. To understand better and to adapt other approaches, automated auditing methods in other analogous environments, such as intrusion detection systems and event correlation in multi-domain network/infrastructure, might be interesting. We consider different techniques of verifying policy compliance or detection of any policy violation including formal verification and validation (V&V) methods.

Audit Output. The proper representation of auditing output is the last and one of the important phases of security auditing. The audit report varies depending on the different demands and requirements of the customers (e.g., tenants). Hierarchy-based reporting helps to fulfill different levels of expectation. Major concern in outputting the result is not to leak any sensitive and unnecessary information to any tenant. Proper information isolation must be ensured.

2.2 Literature Review

This section discusses existing cloud security auditing approaches.

2.2.1 Retroactive Auditing

In the context of cloud auditing, there are several works that target auditing data location and storage in the cloud (e.g., [58]) and others target infrastructure change auditing (e.g., [25]). Particularly, Ullah et al. [108] propose an architecture to build automated security compliance tool for cloud computing platforms focusing on auditing clock synchronization and remote administrative & diagnostic port protection. Doelitzscher [24] proposes on-demand audit architecture for IaaS clouds and an implementation based on software agents to enable anomaly detection system to identify anomalies in IaaS clouds for the purpose of auditing. The works in [108, 24] have the same general objective, which is cloud auditing, as ours, but they use empirical techniques to perform auditing whereas we use formal techniques to model and solve the auditing problem. Tang et al. [107] formalize the core OpenStack access control (OSAC) and propose a domain trust extension for OSAC to facilitate secure cross-domain authorization. We adapt this model in our work. To the best of our knowledge, none of the aforementioned works support auditing a wide variety of security properties in the cloud.

Several industrial efforts include solutions to support cloud auditing in specific cloud environments. For instance, Microsoft proposes SecGuru [12] to audit Azure datacenter network policy using the SMT solver Z3. IBM also provides a set of monitoring tool integrated with QRadar [49], which is their security information and event management system, to collect and analyze events in the cloud. Amazon is offering web API logs and metric data to their AWS clients by AWS CloudWatch & CloudTrail [6] that could be used for the auditing purpose. Although those efforts may potentially assist auditing tasks, none of them directly supports auditing a wide range of security properties covering authentication, authorization and virtual infrastructure on cloud standards.

Several existing efforts consider the verification of access control policies at the design time expressed in the standard eXtensible Access Control Markup Language (XACML) using formal reasoning. Among them, Fisler et al. [30] propose Binary Decision Diagrams (BDD) and custom algorithms to verify access-control policies. Ahn et al. [2] use answer set programming (ASP) and leverage existing ASP reasoning models to conduct policy verification. Arkoudas et al. [7] propose a Satisfiability Modulo Theory (SMT) policy analysis framework. In most of those works, multi-domain access control models are not considered.

To accommodate the need of secure collaborative environments such as cloud computing, there have been some efforts towards proposing multi-domain/multi-tenant access control models (e.g., [36, 107, 40]). Gouglidis and Mavridis [40] leverage graph theory algorithms to verify a subset of the access control security properties. Gouglidis et al. [41] utilize model-checking to verify custom extensions of RBAC with multi-domains [40] against security properties. Lu et al. [68] use set theory to formalize policy conflicts in the context of inter-operation in the multi-domain environment. However, auditing encompasses more than a verification approach. In contrast to these works, we are dealing with the verification of not only the policies but also their implementations, which involve efficient techniques to collect, process, and verify large amount of data.

2.2.2 Intercept-and-Check Auditing

Existing intercept-and-check approaches (e.g., [16, 88]) 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., [61, 59]) monitoring network events and checking network policies at runtime. Weatherman [16] and OpenStack Congress [88] offer security verification of virtual infrastructure using the intercept-and-check approach. These works focus on operational network properties (e.g., black holes and forwarding loops) in traditional networks, whereas our effort is oriented toward preserving compliance with structural security properties that impact isolation in cloud virtualized infrastructures. Designing cloud monitoring services based on security service-level agreements have been discussed in [96].

2.2.3 Proactive Auditing

Proactive security analysis has been explored for software security enforcement through monitoring programs' behaviors and taking specific actions (e.g., warning) in case security policies are violated. Many state-based formal models are proposed for those program monitors over the last two decades. First, Schneider [103] modelled program monitors using an infinite-state-automata model to enforce safety properties. Those automata recognize invalid behaviors and halt the target application before the violation occurs. Ligatti [65] builds on Schneider's model and defines a more general program monitors model based on the so called edit/security automata. Rather than just recognizing executions, edit automata-based monitors are able to suppress bad and/or insert new actions, transforming hence invalid executions into valid ones. Mandatory Result Automata (MRA) is another model proposed by Ligatti et al. [66, 26] that can transform both actions and results to valid ones. Narain [80] proactively generates correct network configurations using the model finder Alloy, which leverages a state of the art SAT solver. To this end, they specify a set of end-to-end requirements in First Order Logic and determine the set of existing network components. Alloy uses a state of the art SAT solver to provide the configurations that satisfy the input requirements for each network component. Considering the huge size of cloud environments and the tremendous space of possible events, adapting those solutions in the cloud is possibly very challenging.

Weatherman [16] is the most closely related work to ours. Aiming at mitigating misconfigurations and enforcing security policies in a virtualized infrastructure, Weatherman has both online and offline approaches. Their online approach intercepts management operations for analysis, and relays them to the management hosts only if Weatherman confirms no security violation caused by those operations. Otherwise, they are rejected with an error signal to the requester. The work defines a realization model, that captures the virtualized infrastructure configuration and topology in a graph-based model. The latter is synchronized with the actual infrastructure using the approach in [14]. Two major limitations of this proposition are: i) the model capturing the whole infrastructure causes a scalability issue for the solution, and ii) the time consuming operation-checking that should be performed on the emergence of each event, makes security enforcement not feasible for large size data centers. Our work overcomes these limitations using dependency models, which are not context-dependent, and the pre-computation steps, which considerably reduce the response-time.

Congress [88] is an OpenStack project offering both online and offline policy enforcement approaches. The offline approach requires submitting a future change plan to Congress, so that the changes can be simulated and the impacts of those changes can be verified against specific properties. In the online approach, Congress first applies the operation to the cloud, then checks its impacts. In case of a violation, the operation is reverted. However, the time elapsed before reverting the operation can be critical to perform some illicit actions, for instance, transferring sensitive files before loosing the assigned role. Foley et al. [31] provide an algebra to assess the effect of security policies replacement and composition in OpenStack. Their solution can be considered as a proactive approach for checking operational property violations.

2.3 Notations

In this thesis, we use several terminologies and notations. Table 2.1 describes the terminologies and notations that we frequently use in this work.

Terminology/Notation	**Description**
User-Level	A cloud level includes the authentication and access control mechanisms
Multi-Domain Cloud	A cloud that consists of multiple domains (i.e., a collection of tenants)
RBAC	Role-based access control
ABAC	Attribute-based access control
SSO	Single sign-on
V&V	Verification and validation
Event Type	The generic name of each cloud event independent of any cloud platform (e.g., create VM and delete port)
Event Instance	An instance of an event type that is observed in logs
Runtime Event	An event instance that is intercepted at runtime
Session	The period within which a user remains active
Watchlist	A list of resources that are allowed as parameters
Critical Event	The events that may violate a security property

Table 2.1: Description of frequently used terminologies in this work.

Chapter 3

Runtime User-Level Auditing for Clouds

3.1 Introduction

The widespread adoption of cloud is still being hindered with the fear of losing control and governance due to the lack of transparency and trust [98, 1]. Particularly, the multi-tenancy and ever-changing nature of clouds usually implies significant design and operational complexity, which may prepare the floor for misconfigurations and vulnerabilities leading to violations of security properties. Runtime security auditing may increase cloud tenants' trust in the service providers by providing assurance on the compliance with security properties mainly derived from the applicable laws, regulations, policies, and standards. Evidently, the Cloud Security Alliance has recently introduced the Security, Trust & Assurance Registry (STAR) for security assurance in clouds, which defines three levels of certifications (self-auditing, third-party auditing, and continuous, near real-time verification of security compliance) [19].

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

- Section 13.2.1 of ISO 27017 [53], which provides security guidelines for the use of cloud computing, recommends *"checking that the user has authorization from the owner of the information system or service for the use of the information system or service..."*.
- The corresponding logging information is available in OpenStack [89] from at least three different sources:

  - Logs of user events (e.g., `router.create.end 1c73637 94305b c7e62 2899` meaning user 1c73637 from domain 94305b is creating a router).
  - Authorization policy files (e.g., `"create_router": "rule:regular_user"` meaning a user needs to be a regular user to create a router).
  - Database record (e.g., `1c73637 Member` meaning user 1c73637 holds the *Member* role).

- Continuously allocating and deprovisioning of resources and user roles for up to 100,000 users mean any verification results may only be valid for a short time. For instance, a re-verification might be necessary after certain frequently-occurred operations such as: `user create 1c73637` (meaning the 1c73637 user is created), and `role grant member 1c73637` (meaning the member role is granted to the 1c73637 user).

Existing approaches can be roughly divided into three categories. First, the *retroactive* approaches (e.g., [70, 73]) catch security violations after the fact. Second, the *intercept-and-check* approaches (e.g., [88, 16]) verify security invariants for each user request before granting/denying it. Third, the *proactive* approaches (e.g., [88, 16, 71]) verify user requests in advance. Our work falls into the second category. Therefore, this work potentially prevents the limitation of the retroactive approaches, and also requires no future change plan unlike proactive approaches (e.g., [88, 16]). In comparison with existing intercept-and-check solutions, our approach reduces the response time significantly and supports a wide range of user-level security properties.

Clearly, during the runtime security auditing, collecting and processing all the data again after each operation can be very costly and may represent a bottleneck for achieving the desired response time due to the performance overhead involved with data collection and processing operations (as reported in Section 3.5). In addition to data collection and processing, runtime verification of ever-changing clouds within a practical response time is essential and non-trivial. In this specific case, no automated tool exists yet in OpenStack for these purposes.

Objectives and Contributions. In this work, we propose a user-level runtime security auditing framework in a multi-domain cloud environment. We compile a set of security properties from both the existing literature on authorization and authentication and common cloud security standards. We perform costly auditing operations (e.g., data collection and processing, and initial verification on whole cloud) only once during the initialization phase so that later runtime operations can be performed in an incremental manner to reduce the cost of runtime verification significantly with a negligible delay. We rely on formal verification methods to enable automated reasoning and provide formal proofs or counter examples of compliance. We implement and integrate the proposed runtime auditing framework into OpenStack, and report real-life experiences and challenges. Our framework supports several popular cloud access control and authentication mechanisms (e.g., role-based access control (RBAC) [29], attribute-based access control (ABAC) [48] and single sign-on (SSO)) with the provision of adding such more extensions. Our experimental results confirm the scalability and efficiency of our approach.

The main contributions of this work are as follows.

- We propose an efficient user-level runtime security auditing framework in a multi-domain cloud environment.
- The study on security properties provides a bridge between the cloud security standards and the literature on multi-domain access control and authentication.
- Our prototype system can be a part of the auditing system for OpenStack-based cloud infrastructure management systems providing a practical auditing solution with the support of common access control and authentication mechanisms (e.g., RBAC, ABAC and SSO).
- The experimental results show that our proposed system is realistic for large-scale cloud environments (e.g., the response time is less than 500 ms for a large cloud with 100,000 users).

3.2 User-Level Security Properties

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

3.2.1 Models

We now describe RBAC, ABAC and SSO models.

RBAC Model. We focus on verifying multi-domain role-based access control (RBAC), which is adopted in real world cloud platforms (as shown in Table 3.1). In particular, we assume the extended RBAC model as in [107], which adds multi-tenancy support in the cloud. The brief definitions of different components of this model are as follows. The details can be found in [107].

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

Plugins	**Cloud Platforms**				
	OpenStack [89]	Amazon EC2 [5]	Microsoft Azure [77]	Google GCP [38]	VMware [109]
RBAC	•	•	•	•	•
ABAC	Blueprint [56]	•	Azure AD	Firebase	•
SSO	Federation	AWS Directory	Microsoft account	G Suite	myOneLogin

Table 3.1: Usage of RBAC, ABAC and SSO in major cloud platforms

Example 1 Figure 3.1 depicts our running example, which is an instance of the access control model presented in [107]. In this scenario, Alice and Bob are the admins of domains, *Da* and *Db*, respectively, with no collaboration (trust) between the two domains; *Pa* and *Pb* are two tenants[2], respectively, owned by the two domains. In such a scenario, we consider a real

[1]We interchangeably use the terms, tenant and project, in Figure 3.1.
[2]We interchangeably use the terms, tenant and project in Figures 3.1 and 3.2

world vulnerability, OSSN-0010[1], found in OpenStack, which allows a tenant admin to become a cloud admin and acquire privileges to bypass the boundary protection between tenants, and illicitly utilize resources from other tenants while evading the billing. Suppose Bob has exploited this vulnerability to become a cloud admin. Figure 3.1 depicts the resultant state of the access control system after this attack. Therein, Mallory belonging to domain, *Da*, is assigned a tenant-role pair (Pb, Member), which is from domain, *Db*. This violates the security requirement of these domains as they do not trust each other.

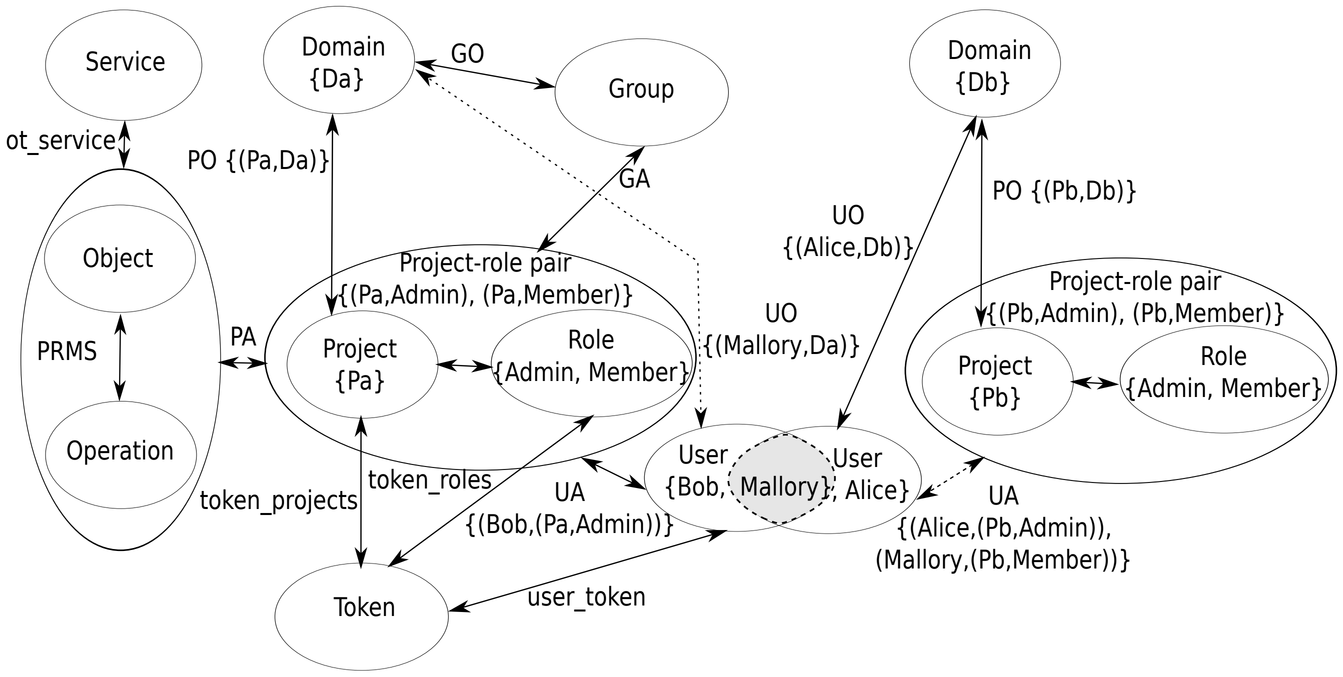


Figure 3.1: Two domain instances of the access control model of [107] depicting the resultant state of the access control system after the exploit of the vulnerability, OSSN-0010. The shaded region and dotted arrows show an instance of the exploit described in Example 1.

ABAC Model. ABAC [48], is considered as a strong candidate to replace RBAC in Sandhu [99], which identifies several limitations of RBAC and thus emphasizes the importance of ABAC specially for large infrastructures (e.g., cloud). In fact, major cloud platforms have started supporting ABAC (as shown in Table 3.1). We briefly review the central concepts of ABAC model here, while leaving more details to [48]. Attributes are name and value pairs, and associated with different entities. The attribute is considered as a function, which takes users or objects

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

as inputs and returns a value from the attribute's scope. A user is a person or a non-person entity, e.g., an application, which requests for actions on objects. A user is described with a set of attributes (*UATT*), e.g., name, salary, role, etc. Objects are system resources (e.g., VMs, files, etc.) which can be accessed only by authorized users. Object attributes (*OATT*) represent resource properties such as risk level, classification and location. Actions are the list of allowed operations on an object by a user. In this work, we mainly use two ABAC functions, i.e., user attribute (*UATT*) and object attribute (*OATT*).

Example 2 Figure 3.2 depicts our running example for ABAC, and shows a similar attack scenario as Example 1. The model in the figure is an instance of the access control model presented in [97], and shows the resultant state of the access control system after this attack.

SSO Mechanism. SSO, which is a popular cloud authentication extension and supported by major cloud platforms (shown in Table 3.1), only requires a single user action to permit a user to access all authorized computers and systems. In this work, we detail two SSO protocols: OpenID [85] and SAML [82] supported by OpenStack and many other cloud platforms.

However, there are several attacks (e.g., [111, 42, 83, 8]) on two above-mentioned SSO protocols. The following describes several security concerns specific to these protocols.

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

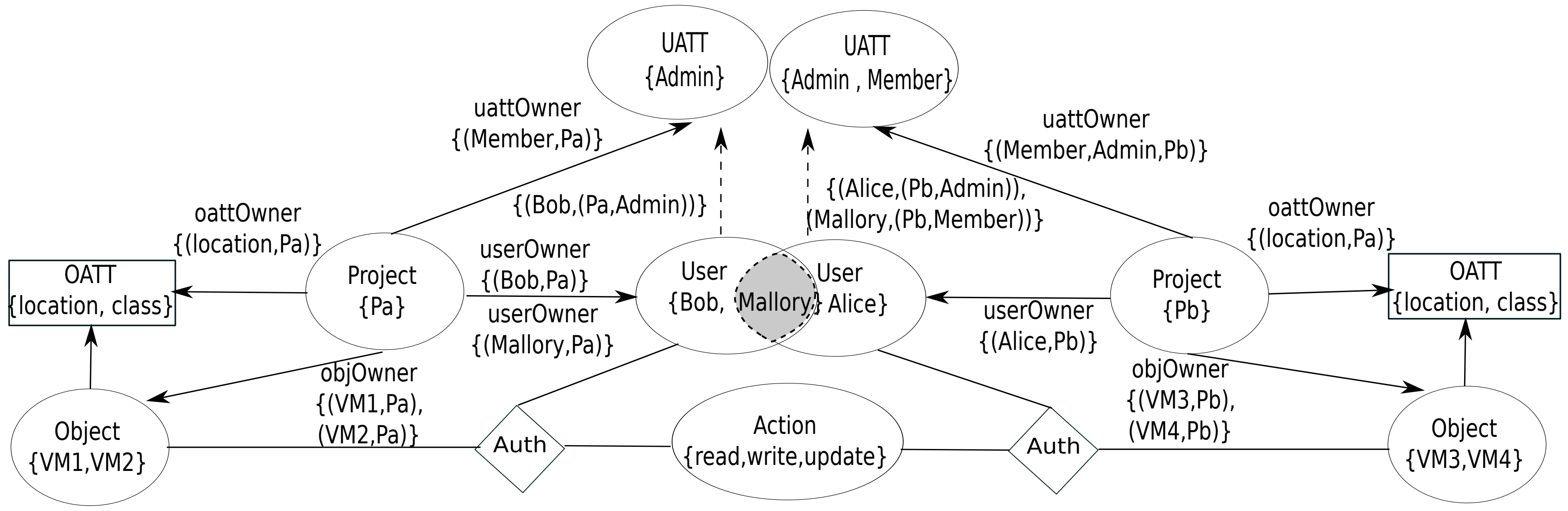


Figure 3.2: Two tenant instances of the access control model of [97] depicting the resultant state of the access control system after the exploit of the vulnerability, OSSN-0010. The shaded region and dotted arrows show an instance of the exploit described in Example 2.

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

3.2.2 Security Properties

Table 3.2 presents an excerpt of the list of user-level security properties that we identify from the access control and authentication literature, relevant standards (e.g., ISO 27002 [52], NIST SP 800-53 [81], CCM [18] and ISO 27017 [53]), and the real-world cloud implementation (e.g., OpenStack). Even though some properties (e.g., cyclic inheritance) are not directly found in any standard, they are included based on their importance and impact described in the literature (e.g., [41]).

RBAC Security Properties. RBAC-specific security properties are shown in Table 3.2. For our running example, we will focus on following two properties. Common ownership: based on the challenges of multi-domain cloud discussed in [41, 107], users must not hold any role from another domain. Minimum exposure: each domain in a cloud must limit the exposure of its information to other domains [107].

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

ABAC Security Properties. Table 3.2 provides an excerpt of ABAC-related security properties supported by our auditing system. Some properties are specific to ABAC, and rests are adopted from RBAC. We only discuss the following properties, which are extended or added for ABAC.

- Consistent constraints: Jin et al. [57] define constraints for different basic changes in ABAC elements e.g., adding/deleting users/objects. After each operation, certain changes are necessary to be properly applied. This property verifies whether all constraints have been performed.
- Common ownership: For ABAC, the common ownership property also includes objects and their attributes so that an object owner and the owner of the allowed user performing certain

Properties	**Standards**				RBAC	ABAC	SSO
	ISO27002 [52]	ISO27017 [53]	NIST800 [81]	CCM [18]			
Role activation [54]	*13.2.2b*	*15.2.2b*	*AC-1*	*IAM-09*	●	●	
Permitted action [54]	*11.2.1.b, 1.2.2c*	*13.2.1b, 13.2.2c*	*AC-14*	*IAM-10*	●	●	
Common ownership [41]	*11*	*13*	*AC*	*IAM*	●	●	
Minimum exposure [107]	*11.6.1*	*13.4.1*	*AC-4*	*IAM-04,06*	●	●	
Separation of duties [107]	*11.6.2*	*13.6.2*	*AC-5*	*IAM-02,05*	●		
Cyclic inheritance [41]					●		
Privilege escalation [41]	*11.2.2.b*	*13.2.2b*	*AC-6*	*IAM-08*	●	●	
Cardinality [54]	*11.2.4*	*13.2.4*	*AC-1*		●	●	
Consistent constraints [57]							
add/delete user	*11.5.1*	*13.4.2*	*AC-7,9*	*IAM-02*		●	
modify user attributes	*13.2.2b*	*15.2.2b*	*AC-1*	*IAM-09*		●	
add/delete object	*13.2.2b*	*15.2.2b*	*AC-1*	*IAM-09*		●	
modify object attributes	*13.2.2b*	*15.2.2b*	*AC-1*	*IAM-09*		●	
Session de-activation [81]	*11.5.5*	*13.2.8*	*AC-12*		●	●	
User-access validation [53]	*11.5.2*	*13.4*	*AC-3*	*IAM-10*			●
User-access revocation [52]	*11.2.1h*	*13.2.1h*	*AC-2*	*IAM-11*			●
No duplicate ID [18]	*11.5.2*	*13.5.2*	*AC-2*	*IAM-12*			
Secure remote access [53]	*11.4.2*	*13.4.2*	*AC-17*	*IAM-02,07*			
Secure log-on [53]	*11.5.1*	*13.4.2*	*AC-7,9*	*IAM-02*			
Session time-out [81]	*11.5.5*	*13.2.8*	*AC-12*				●
Concurrent session [81]		*13.5.4*	*AC-10*				
Brute force protection	*11.2.2.b*	*13.2.2b*	*AC-1*	*IAM-09*			●
No caching	*11.2.1.b, 1.2.2c*	*13.2.1b, 13.2.2c*	*AC-14*	*IAM-10*			●

Table 3.2: An excerpt of user-level security properties

actions on that object must be the same.

Authentication-Related Security Properties. Table 3.2 shows an excerpt of security properties related to generic authentication mechanisms and extensions (e.g., SSO). We discuss the SSO-related properties as follows. Brute force protection: account lockout, CAPTCHA or any such brute force protection must be applied in SSO. No caching: SSO and associated applications should set no-cache and no-stored-cache directives. User access revocation: logout from one application must end sessions of other applications. User access validation: only a valid authentication token must pass the authentication step.

3.2.3 Threat Model

Our threat model is based on two facts. First, our solution focuses on verifying the security properties specified by cloud tenants, instead of detecting specific attacks or vulnerabilities (which is the responsibility of IDSes or vulnerability scanners). Second, the correctness of our verification results depends on the correct input data extracted from logs and databases. Since an attack may or may not violate the security properties specified by the tenant, and logs or databases may potentially be tampered with by attackers, our results can only signal an attack in some cases. Specifically, the in-scope threats of our solution are attacks that violate the specified security properties and at the same time lead to logged events. The out-of-scope threats include attacks that do not violate the specified security properties, attacks not captured in the logs or databases, and attacks through which the attackers may remove or tamper with their own logged events. More specifically, in this work we focus on user-level security threats and rely on existing solutions (e.g., [16, 71]) to identify virtual infrastructure level threats. We assume that, before our runtime approach is launched, an initial verification is performed and potential violations are resolved. However, if our solution is added from the commencement of a cloud, obviously no prior security verification (including the initial phase) is required. We also assume that tenant-defined policies are properly reflected in the policy files of the cloud platforms.

3.3 Runtime Security Auditing

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

3.3.1 Overview

Figure 5.20 shows an overview of our runtime auditing approach. This approach contains two major phases: i) initialization, where we conduct a full verification on the collected and processed cloud data, and ii) runtime, where we incrementally verify the compliance upon dynamic changes in the cloud. The initialization phase is performed only once through an offline verification. This phase performs all costly operations such as data collection and processing, and an initial full compliance verification for a list of security properties. The initial verification result is stored in the result repository. For the latter, we devise an incremental verification approach to minimize the workload at the runtime. During the runtime phase, each management operation (e.g., create/delete a user/tenant) is intercepted, its parameters are processed with the previous result, and finally the verification engine evaluates the compliance and provides the latest verification result. We elaborate major phases of our system as follows.

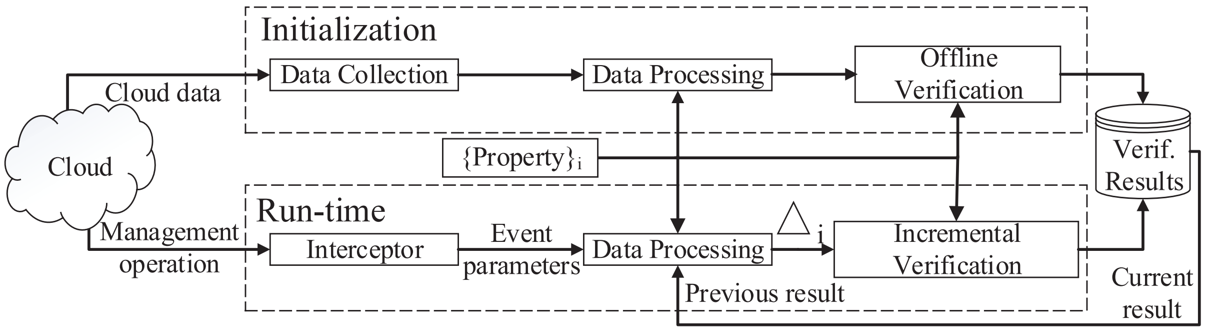


Figure 3.3: An overview of our runtime security auditing approach

3.3.2 Initialization Phase

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

The collection engine is responsible for collecting the required data in a batch mode from the cloud management system. The role of the processing engine is to filter, format, aggregate, and correlate this data. The required data is distributed throughout the cloud and in different formats (e.g., files and databases). The processing engine must pre-process the data in order to provide specific information needed to verify given properties. A final processing step is to generate and store the inputs to be used by the compliance verification engine. Note that the format of the inputs depends on the selected back-end verification engine.

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

3.3.3 Runtime Phase

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

We update the verification results continuously by verifying the cloud at runtime. The runtime verification is event-driven and property-specific. Table 3.3 shows the events that may affect the result of verification for certain properties. The bottom part of Figure 5.20 depicts the steps of this phase. We intercept each operation request generated from the cloud management interface. We further retrieve the parameters of the request and pass them to the data processing module. The data processing module performs similarly as described in Section 3.3.2. However, during the runtime phase, mostly partial data are sent for the incremental verification for each security property. Thus, the incremental verification is only conducted on the impact of the current change. Then, the final verification result is inferred from the result of current incremental verification and the previous result. Incremental verification is performed using any of the two methods: i) *deltaVerify*, where compliance verification mechanism discussed in the initialization phase is applied on the delta data, and ii) *customAlgo*, where security property specific customized algorithms are performed. We discuss our runtime verification algorithm in more details in Section 3.4.2. In the following examples, we assume that the previous verification result for a specific property is stored in $Result_{t0}$, the parameters of the intercepted event is in Δ_i and the updated result will be stored in $Result_t$. For example, all user-role pairs violating the common ownership property at time $t0$ are stored in $Result_{t0}$ as $\{Mallory, Da, (Pb, Member), Db\}$.

Example 4 Table 3.3 shows that the verification result for the common ownership property may change for following events: grant role, delete role, delete user, delete tenant and delete domain. Upon intercepting any of these events, we conduct incremental verification as follows:

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

similarly handled.

Example 5 Similarly, upon intercepting any of the events marked for the permitted action property in Table 3.3, we conduct incremental verification as follows:

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

Identifying the Impacts of Events. By observing the impacts of cloud events, Table 3.3 lists all events that may change the verification result of certain security properties. However, identifying impacts of events in cloud can be challenging. Also, the completeness of the method of identifying the impacts, relies on the specifications of APIs by the cloud platforms. In this work, we mainly follow two methods (i.e., API documentation and infrastructure change inspection) proposed by Bleikertz et al. [16]. Firstly, we go through the API documentation provided by cloud platforms to obtain API specifications including their functionality, parameters and impacts on the infrastructure. Secondly, we perform different events and observe the infrastructure configuration change to capture the impact of those events. Finally, we combine this knowledge with the definition of security properties to populate Table 3.3.

Provision of Enriching the Security Property List. Beside the security properties in Section 3.2.2, tenants might intend to add new security properties over time. Our framework provides the provision of adding new security properties through following simple steps. First, the new security property is defined in the cloud system context, which can be simply performed by following our existing techniques discussed in Section 3.2.2 to apply high-level standard terminologies to cloud-specific resources. Next, the property is translated to the first order logic and then to Constraint Satisfaction Problem (CSP) constraints, and in many cases the existing relations discussed in Section 3.3.4 can be re-used as they include basic relations such as *belongs to,*

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

Table 3.3: Events that influence verification results for certain properties

owner of, authorized for, etc. Our data collection engine already collects data from all relevant sources of data of a cloud platform regardless of security properties. Therefore, no extra effort is needed in the data collection phase, unless the new property requires data from a different layer in the cloud (e.g., SDN). Then, the data processing effort for a new property mainly involves building correlation between data from different sources, because other processing steps are mostly property-independent. The remaining initial verification step is only to add constraints of the new property to the verification list. Finally, we identify the events that may alter the verification result of the new property by re-utilizing the knowledge of impacts of events, and perform the runtime verification through incremental steps either using the deltaVerify method or by a customized algorithm (as in Section 3.4.2). Additionally, whenever there is any change in the event specification for a cloud system, we capture the update on impacts (if any) of events on the security properties.

3.3.4 Formalization of Security Properties

As a back-end verification mechanism, we formalize verification data and properties as Constraint Satisfaction Problem (CSP) and use a constraint solver, namely, Sugar [105], to validate the compliance. CSP allows formulation of many complex problems in terms of variables defined over finite domains and constraints. Its generic goal is to find a vector of values (a.k.a. assignment) that satisfies all constraints expressed over the variables. If all constraints are satisfied, the solver returns SAT, otherwise, it returns UNSAT. In the case of a SAT result, a solution to the problem is provided. In our case, we formalize each security property in CSP to verify in Sugar. After verification, Sugar provides proper evidence (a.k.a counter examples) of a violation (if any) of a security property. In the following, we first provide a generic description of model formalization, then illustrate examples of property formalization, and finally show some counter examples for those security properties.

Model Formalization. Referring to Figures 3.1 and 3.2, entities are encoded as CSP variables with their domains definitions (over integer), where instances are values within the corresponding domain. For example, *User* is defined as a finite domain ranging over integer such that (domain *User* 0 *max_user*) is a declaration of a domain of users, where the values are between 0 and *max_user*. Relationships and their instances are encoded as relation constraints and their supports, respectively. For example, *AuthorizedR* is encoded as a relation, with a support as follows: (relation *AuthorizedR* 3 (supports (r1,u1,t1) (r2,u2,t2))). The support of this relation will be fetched and pre-processed in the data processing step. The CSP code mainly consists of four parts:

- *Variable and domain declaration.* We define different entities and their respective domains. For example, *u* and *op* are entities (or variables) defined respectively over the domains *User* and *Operation*, which range over integers.
- *Relation declaration.* We define relations over variables and provide their support from the verification data.
- *Constraint declaration.* We define the negation of each property in terms of predicates over the involved relations to obtain a counter example in case of a violation.
- *Body.* We combine different predicates based on the properties to verify using Boolean operators.

Properties Formalization for RBAC. Security properties are presented as predicates over relation constraints and predicates. We detail two representative properties in this work. We first express these properties in first order logic [11] and then in their CSP formalization (using Sugar syntax). Table 3.4 summarizes the relations that we use in these properties.

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

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

The corresponding CSP constraint is

Relations in Properties	**Evaluate to $True$ if**	**Corresponding Relations in Fig. 3.1**
$AuthorizedOp(d,t,u,r,o,op)$	In domain d, and tenant t, the user u, with the role r is authorized to perform operation op on object o	UA, PO, tenant-role pair, PA, PRMS
$OwnerD(od,t,o)$	Domain od is the owner of the object o in tenant t	PO, tenant-role pair, PA
$AuthorizedR(u,t,r)$	User u belonging to tenant t is authorized for the role r	UA, tenant-role pair
$BelongsToD(u,d)$	User u belongs to the domain d	UO
$TenantRoleDom(t,r,d)$	Role r is defined within the domain d in tenant t	PO, tenant-role pair
$LogEntry(d,t,u,r,o,op)$	Operation op on object o is actually performed by user u having role r in tenant t and domain d	ND
$ActiveToken(tok,d,t,u,r,time)$	Token tok is active at time $time$ and in use by user u having role r in tenant t and domain d	UA, token_tenants, token_roles, PO, tenant-role pair

Table 3.4: Correspondence between relations in our formalism and relationships/entities in Figure 3.1. Note that one of the relations (in third column) is denoted by ND as it is inferred from dynamic data (e.g., logs).

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

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

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

The CSP constraint for this property is:

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

Properties Formalization for ABAC. In the following, we show formalization of one security property for ABAC.

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

Since ABAC is more expressive, there might be a larger set of properties for ABAC (as shown in Table 3.2). However, the verification complexity depends more on the security properties, and less on the model. For example, the common ownership, permitted action and minimum exposure properties show different level of complexities, as shown through their formal representation and as supported by the experiment results in Section 3.5.

Properties Formalization for SSO. In the following, we present formalization steps of one SSO related security property (i.e., user access revocation). The user access revocation property is for the token-based user access. At a given time, for active tokens, we check that none of the situations leading to their revocation has been occurred. Function *TimeStamp(tok)* returns the token expiration time.

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

Thus, the corresponding CSP constraint is:

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

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

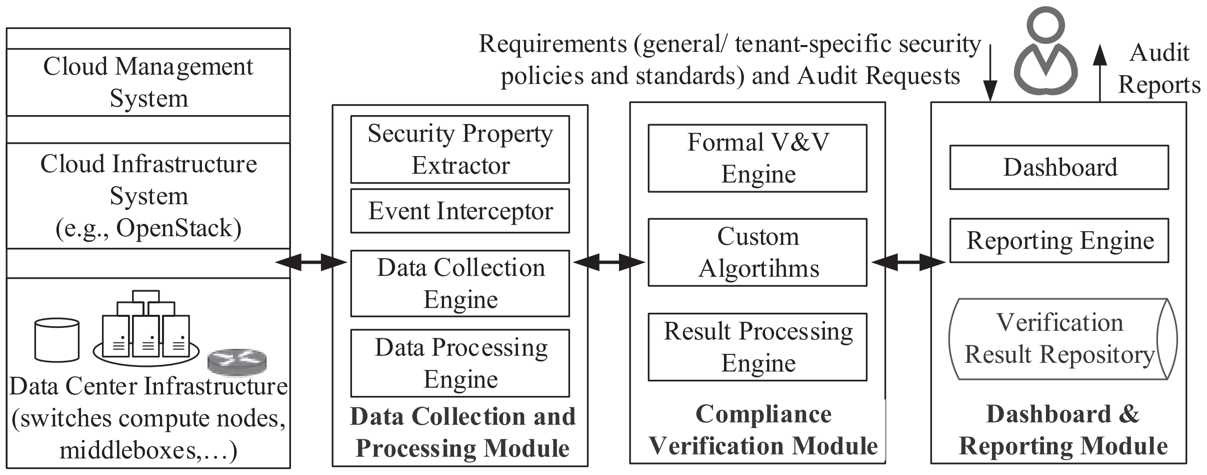
Example 6 The CSP predicate for the common ownership property is as follows: $(and\ BelongsToD(u,d)\ AuthorizedR(u,t,r)\ (not\ Tenant\text{-}\ RoleDom(t,r,d)))$. The property is violated when a user from one domain holds a tenant-role pair from another domain. In other words, in case of a violation there exists at least a set of predicates as follows: $(and\ BelongsToD(u1,d1)\ AuthorizedR(u1,t2,r2)\ (notTenant\text{-}RoleDom(t2,r2,d1)))$; meaning that the user, $u1$, from domain, $d1$, holds a role pair, $t2$-$r2$, which is not from domain, $d1$. In such cases, our auditing system using Sugar identifies that the (u1, d1, t2, r2) tuple is the cause for a violation of the common ownership property. In Section 3.4, Example 7 further extends this example to show concrete examples of evidences provided by our auditing system.

3.4 Implementation

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

3.4.1 Architecture

Figure 3.4 shows a high-level architecture of our runtime verification framework. It has three main components: data collection and processing module, compliance verification module, and dashboard & reporting module. In the following, we describe different engines inside the data collection and processing module. The security property extractor identifies the sources of required data for a list of security properties. The event intercepter intercepts each management operation requested by the user in the cloud infrastructure system. The data collection engine interacts mainly with the cloud management system, the cloud infrastructure system (e.g., OpenStack), and elements in the data center infrastructure to collect various types of audit data. Then the data processing engine aids to build the correlation and to uniform the collected data. Our compliance verification module is responsible for the offline and runtime verification using the formal verification and validation (V&V) tools and our custom algorithms. Finally, the dashboard & reporting module interacts with the cloud tenant through the dashboard to obtain the tenant requirements and to provide the tenant with the verification results in a report. Tenant requirements encompass both general and tenant-specific security policies, applicable standards, as well as verification queries.



Cloud Management System
Cloud Infrastructure System (e.g., OpenStack)
Data Center Infrastructure (switches compute nodes, middleboxes,...)
Security Property Extractor
Event Interceptor
Data Collection Engine
Data Processing Engine
Data Collection and Processing Module
Requirements (general/ tenant-specific security policies and standards) and Audit Requests
Formal V&V Engine
Custom Algortihms
Result Processing Engine
Compliance Verification Module
Audit Reports
Dashboard
Reporting Engine
Verification Result Repository
Dashboard & Reporting Module

Figure 3.4: A high-level architecture of our runtime verification framework

3.4.2 Integration into OpenStack

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

Background. OpenStack [89] is an open-source cloud infrastructure management platform in which Keystone is its identity service, Neutron is its network component, Nova is its compute component, and Ceilometer is its telemetry.

Data Collection Engine. The collection engine involves several components of OpenStack e.g., Keystone and Neutron for collecting data from log files, policy files, different OpenStack databases and configuration files from the OpenStack ecosystem to fully capture the configuration. We present hereafter different sources of data in OpenStack along with the current support for auditing offered by OpenStack. The main sources of data in OpenStack are logs, configuration files, and databases. Table 3.5 shows some sample data sources. The OpenStack logs are maintained separately for each service, e.g., Neutron, Keystone, in a directory named *var/log/component_name*, e.g., *keystone.log* and *keystone_access.log* are stored in the *var/log/keystone* directory. Two major configuration files, namely, *policy.json* and *policy.v3cloudsample.json*, contain policy rules defined by both the cloud provider and tenant

Relations	Sources of Data
AuthorizedOp	user, assignment, role in Keystone database and *policy.json* and *policy.v3cloudsample.json*
OwnerD	user, assignment in Keystone database and *policy.json*
AuthorizedR	user, tenant, assignment in Keystone database
BelongsToD	user, domain tables in Keystone database
TenantRoleDom	tenant, assignment, domain tables in Keystone database
LoggedEntry	*keystone_access.log* and Ceilometer database
ActiveToken	Keystone database and *keystone_access.log*

Table 3.5: Sample data sources in OpenStack for relations in Table 3.4

admins, and are stored in the *keystone/etc/* directory. The third source of data is a collection of databases, hosted in a MySQL server, that can be read using component-specific APIs such as Keystone and Neutron APIs. With the proper configuration of the OpenStack middleware, notifications for specific events in Keystone, Neutron and Nova can be gathered from the Ceilometer database.

The effectiveness of a verification solution critically depends on properly collected evidences. Therefore, to be comprehensive in our data collection process, we firstly check fields of all varieties of log files available in Keystone and more generally in OpenStack, all configuration files and all Keystone database tables (18 tables). Through this process, we identify all possible types of data with their sources. Due to the diverse sources of data, there exist inconsistencies in formats of data. On the other hand, to facilitate verification, presenting data in a uniform manner is very important. Therefore, we facilitate proper formatting within our data processing engine.

Data Processing Engine. Our data processing engine, which is implemented in Python, mainly retrieves necessary information from the collected data, converts it into appropriate formats, recovers correlation, and finally generates the source code for Sugar. First, our tool fetches the necessary data fields from the collected data, e.g., identifiers, API calls, timestamps. Similarly, it fetches access control rules, which contain API and role names, from *policy.json* and *policy.v3cloudsample.json* files. In the next step, our processing engine formats each group of data as an n-tuple, i.e., (user, tenant, role, etc.). To facilitate verification, we additionally correlate different data fields. In the final step, the n-tuples are used to generate the portion of the Sugar's source code, and the relationships for security properties (discussed in Section 3.3.4) are also appended with the code. Different scripts are needed to generate the Sugar source code for the verification of different properties, since relationships are usually property-specific.

The logs generated by each component of OpenStack usually lack correlation. Even though Keystone processes authentication and authorization steps prior to a service access, Keystone does not reveal any correlated data. Therefore, we build the data correlation support within the processing engine. For an example, we infer the relation $(user\ operation)$ from the available relations $(user\ role)$ and $(role\ operation)$. In our settings, we have $61,031$ entries in the $(user\ role)$ relations for $60,000$ users. The number of entries is larger than the number of users, because there are some users with multiple roles. With the increasing number of users having multiple roles, the size of this relation grows, and as a result, it increases the complexity of the correlation step.

Initial Compliance Verification. The compliance verification module contains two major modules responsible for the initial verification and runtime verification, respectively. The prerequisite formalization steps of the initial verification are already discussed in Section 3.3.4. Here, we explain different parts of a Sugar source code through a simple example and verification algorithm (as in Algorithm 1) in the following.

Listing 3.1: Sugar source code for the common ownership property

```
//Declaration
(domain Domain 0 500) (domain Tenant  0 1000)
(domain Role 0 1000) (domain User 0 60000)
(int D Domain) (int R Role)
(int P Tenant) (int U User)
//Relations Declarations and Audit Data as their Support
(relation BelongsToD 2 (supports (100 401) (40569 123)
(102 452) (145 404) (156 487) (128 463)))
(relation AuthorizedR 3 (supports (100 301 225)
(40569 1233 9) (102 399 230) (101 399 231)))
```

```
11 (relation TenantRoleDom 3 (supports (301 225 401)
12 (1233 9 335) (399 230 452) (399 231 452)))
13 //Security Property: Common Ownership
14 (predicate(ownership D R U P)
15 (and (AuthorizedR  U P R ) (BelongsToD U D)
16 (not(TenantRoleDom P R D)) ))
17 (ownership D R U P)
```

Example 7 Listing 3.1 is the CSP code to verify the common ownership property. Each domain and variable are first declared (lines 2-5). Then, the set of involved relations, namely, *BelongsToD*, *AuthorizedR*, and *TenantRoleDom*, are defined and populated with their supporting tuples (lines 7-12), where the support is generated from actual data in the cloud. Then, the common ownership property is declared as a predicate, denoted by *ownership*, over these relations (lines 14-16). Finally, the predicate is instantiated (line 17) to be verified. As we are formalizing the negation of the properties, we are expecting the UNSAT result, which means that all constraints are not satisfied (i.e., no violation of the property). Note that the predicate is unfolded internally by the Sugar engine for all possible values of the variables, which allows to verify each instance of the problem among possible values of domains, users and roles.

In this example, we also describe how a violation of the common ownership property may be caught by our verification process. Firstly, our program collects data from different tables in the Keystone database including *users*, *assignments*, and *roles*. Then, the processing engine converts the collected data and represents as tuples; for our example: (40569 123) (40569 1233 9) (1233 9 335), where Mallory: 40569, Da: 123, Pb: 1233, member: 9 and Db: 335. Additionally, the processing engine interprets the property and generates the Sugar source code (as Listing 3.1) using processed data and translated property. Finally, the Sugar engine is used to verify the security properties. The CSP predicate for the common ownership property is as follows: $(and\ BelongsToD(u,d)\ AuthorizedR(u,t,r)\ (not\ Tenant-RoleDom(t,r,d)))$. As Mallory belongs to domain, Da, $BelongsToD(Mallory,Da)$ evaluates to true. Mallory has been authorized a tenant-role pair, $(Pb,member)$, thus $AuthorizedR(Mallory,Pb,member)$ evaluates to true. However, $TenantRoleDom(Pb,member,Da)$ evaluates to false, as the pair $(Pb,member)$

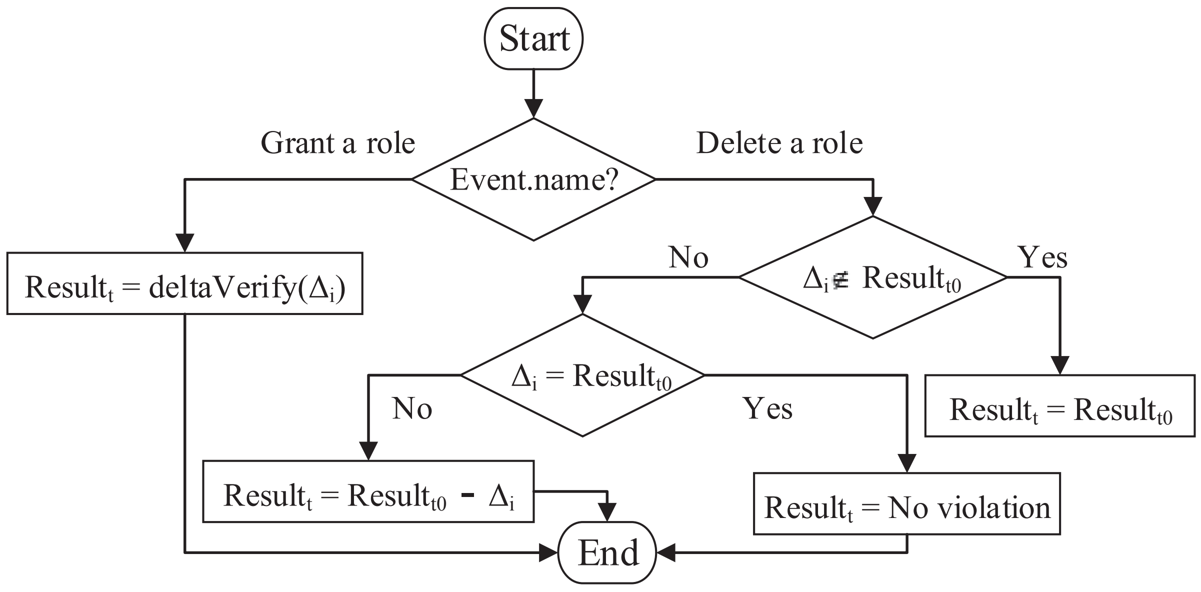
does not belong to domain Da. Then, the whole *ownership* predicate unfolded for this case is evaluated to true. In this case, the output of sugar is SAT, which confirms that Mallory violates the common ownership property and further presents the cause of the violation, i.e., $(d = 123, r = 9, t = 1233, u = 40569)$.

Algorithm 1 Runtime Compliance Verification

```
1:  procedure INITIALIZE(Properties,CloudOS)
2:      rawData = collectData(CloudOS)
3:      verData = processData(rawData)
4:      for each property $p_i \in Properties$ do
5:          $Result_{t0,pi}$= Verify ($p_i$,verData)
6:  procedure RUNTIME(Event,$Result_{t0}$,Properties)
7:      for each property $p_i \in Properties$ do
8:          $\Delta_i$ = processData(event.parameters)
9:          if incremental-method($p_i$) = $custom$ then
10:             custom-algo($event$,$p_i$,$Result_{t0,pi}$,$\Delta_i$)
11:         else
12:             deltaVerify($event$,$p_i$,$Result_{t0,pi}$,$\Delta_i$)
13:     return $Result_t$
14: procedure DELTAVERIFY($event$,$p_i$,$Result_{t0,pi}$,$\Delta_i$)
15:     $Result_{t,pi}$ = verify($p_i$,$\Delta_i$)
```

Runtime Verification. Our runtime verification engine implements Algorithm 1. Firstly, the interceptor module intercepts each management operation based on the existing intercepting methods (e.g., audit middleware [90]) supported in OpenStack. Events are primarily created via the notification system in OpenStack; Nova, Neutron, etc. emit notifications in a JSON format. Here, we leverage the audit middleware in Keystone to intercept Keystone, Neutron and Nova events by enabling the audit middleware and configuring filters. Secondly, the data processing engine handles the intercepted parameters to perform similar data processing operations as discussed previously. The processed data is denoted as Δ_i. Finally, the runtime verification engine performs incremental steps either using the deltaVerify method, which involves Sugar, or custom algorithms. Figures 3.5 and 3.6 show the incremental steps for the common ownership and permitted action properties, respectively.

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



Start
Event.name?
Grant a role
Delete a role
Result$_t$ = deltaVerify(Δ_i)
$\Delta_i \not\equiv$ Result$_{t0}$
No
Yes
Δ_i = Result$_{t0}$
No
Yes
Result$_t$ = Result$_{t0}$
Result$_t$ = Result$_{t0}$ - Δ_i
Result$_t$ = No violation
End

Figure 3.5: Showing the runtime steps for the common ownership property

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

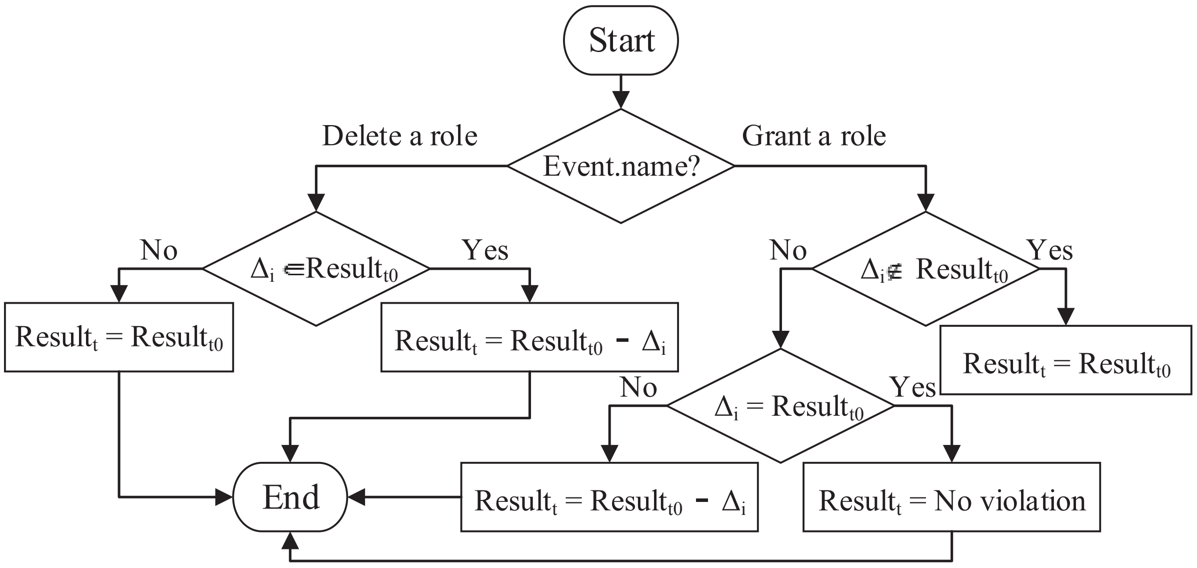

Start
Event.name?
Delete a role
Grant a role
No
$\Delta_i \in Result_{t0}$
Yes
$Result_t = Result_{t0}$
$Result_t = Result_{t0} - \Delta_i$
No
$\Delta_i \notin Result_{t0}$
Yes
$Result_t = Result_{t0}$
No
$\Delta_i = Result_{t0}$
Yes
End
$Result_t = Result_{t0} - \Delta_i$
$Result_t$ = No violation

Figure 3.6: Showing the runtime steps for the permitted action property

3.4.3 Integration to OpenStack Congress

To demonstrate the service agnostic nature of our framework, we further integrate our auditing method with OpenStack Congress [88]. Congress implements policy as a service in OpenStack in order to provide governance and compliance for dynamic infrastructure. Congress can integrate third party verification tools using a data source driver mechanism. Using Congress policy language that is based on Datalog, we define several tenant specific security policies as same as security properties described in Section 3.2.2. We then use our processed data to detect those security properties for multiple tenants. The outputs of the data processing engine in both cases of initialization and runtime are in turn provided as inputs for Congress to be asserted by the policy engine. This integrates compliance status for some policies whose verification is not yet supported by Congress (e.g., permitted action and minimum exposure).

3.5 Experiments

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

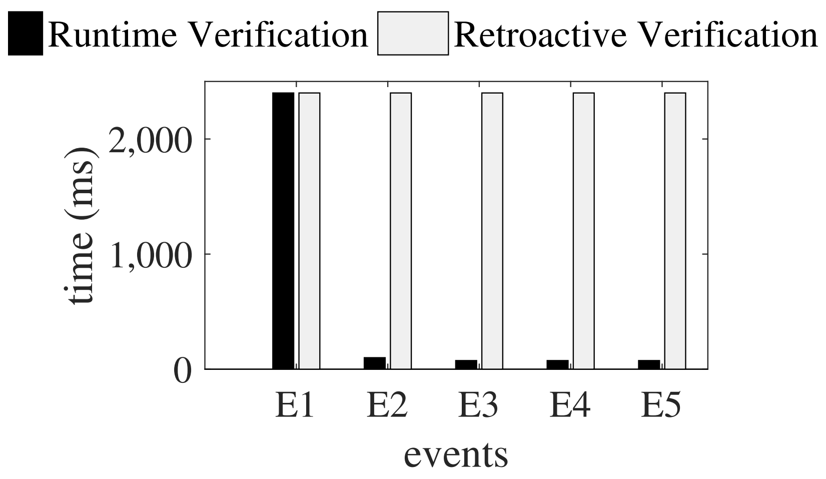


Figure 3.7: Comparing the verification time required after each event for our system and the retroactive approach (e.g., [73]) for the common ownership property. Here, E1=initialization, E2=grant a role, E3=delete a role, E4=delete a user and E5=delete a tenant. The results are for our largest dataset.

3.5.1 Experimental Settings

We collect data from the OpenStack setup inside a lab environment. Our OpenStack version is Mitaka (2016.10.15) with Keystone API version v3. There are one controller node and three compute nodes, each having Intel i7 dual core CPU and 2 GB memory with the Ubuntu 16.04 server. To make our experiments more realistic, we follow recently reported statistics (e.g., [92] and [35]) to prepare our largest dataset consisting 100,000 users, 10,000 tenants, and 500 domains. For verification, we use the V&V tool, Sugar V2.2.1 [105]. We conduct the experiment for 12 different datasets in total. All data processing and V&V experiments are conducted on a PC with 3.40 GHz Intel Core i7 Quad core CPU and 16 GB memory, and we repeat each experiment 1,000 times.

3.5.2 Results

The objective of the first set of our experiments (see Figures 3.7, 3.8 and 3.9) is to demonstrate the time and memory efficiency of our solution, and to compare the performance with

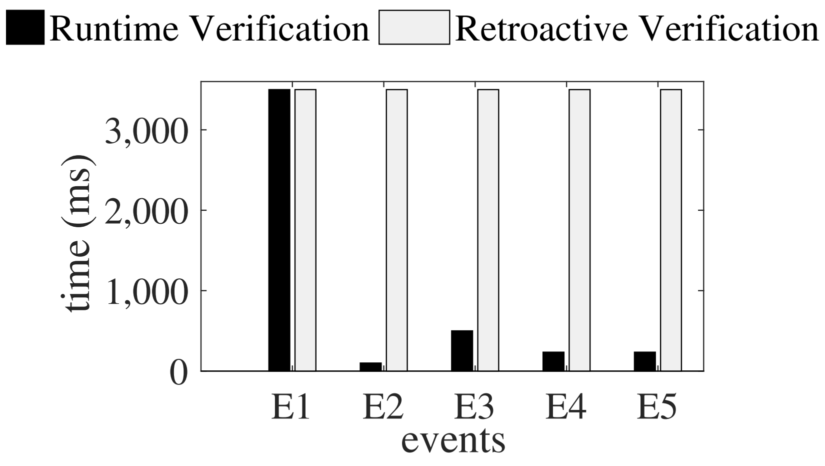


Figure 3.8: Comparing the verification time required after each event for our system and the retroactive approach (e.g., [73]) for the permitted action property. Here, E1=initialization, E2=grant a role, E3=delete a role, E4=delete a user and E5=delete a tenant. The results are for our largest dataset.

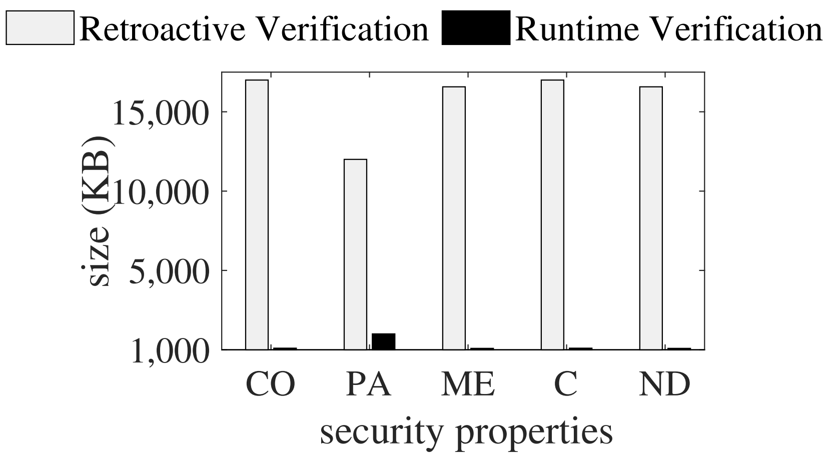


Figure 3.9: Total size (in Kilo Bytes) of the data to be verified both for our approach and a naive approach for different properties (where CO: common ownership, PA: permitted action, ME: minimum exposure, C: cardinality, ND: no duplicate ID). The results are for our largest dataset.

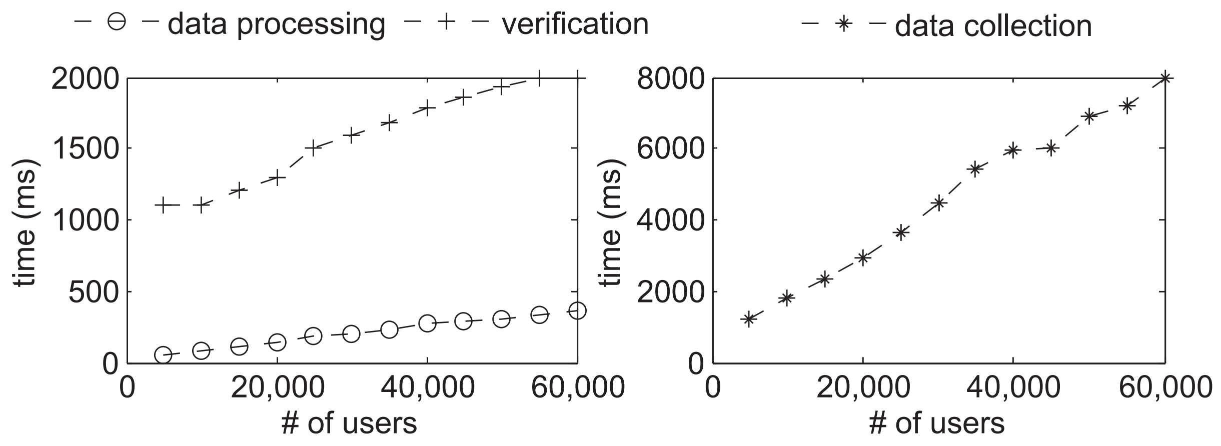


Figure 3.10: Time required for each step during the initialization phase for the common ownership property while varying the number of users. Time for the data collection (right) is shown separately, as it is a one-time effort. In all cases, the number of domains is 500 and number of tenants is 10,000.

a retroactive auditing approach similar as in [73]. Firstly, Figure 3.7 shows time in milliseconds required for our runtime verification framework for the common ownership property. Our runtime verification requires a relatively expensive (i.e., about 2.5 seconds) initialization phase, similar to that of the retroactive approach. Afterwards, our runtime approach takes less than 100 ms; whereas, the retroactive approach always takes 2.5 seconds. Secondly, Figure 3.8 compares time in milliseconds required for verifying the permitted action property by our framework and a retroactive verification method. For this property, we obtain results of the same nature as the previous one i.e., requiring only a relatively expensive (i.e., about 3.5 seconds) initialization phase followed by runtime verification costing maximum 500 ms. For the permitted action property, after the *delete a role* event, a search for a certain role is performed; hence the verification time reaches the maximum value. Otherwise, verification time is within 100 ms for both properties. Finally, Figure 3.9 depicts the comparison between memory requirement for both approaches while verifying different properties. The retroactive approach requires 12 MB to 17 MB space, as each time we have to load the whole verification data. Whereas, in the runtime approach, mostly we perform verification only on the changed data, therefore it takes maximum 1 MB memory.

Our second set of experiments (see Figures 3.10, 3.11, 3.12, 3.13, 3.14 and 3.15) is to demonstrate the time efficiency of individual phases of our solution. Firstly, Figure 3.10 shows

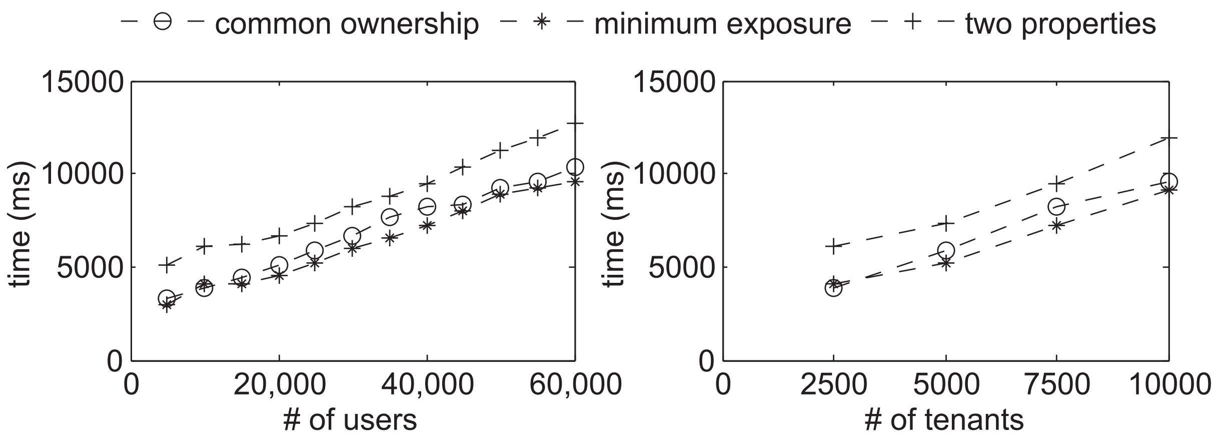


Figure 3.11: Total time required to perform the initialization phase for common ownership, minimum exposure and both properties together, by varying the number of users with fixed 5,000 tenants (left) and the number of tenants with fixed 30,000 users (right). In all cases, the number of domains is 500. Note that time in curves encompasses all three steps (collection, processing and verification). For the curve of two properties, data collection is performed one time.

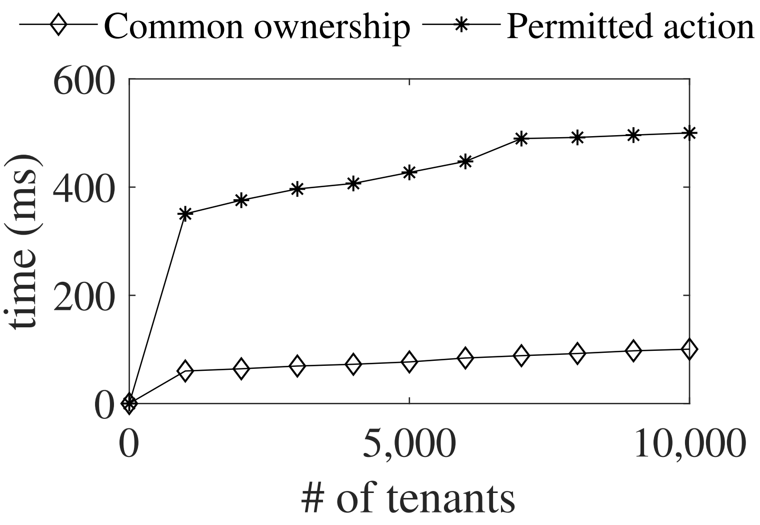


Figure 3.12: Total time required to perform the runtime phase of the common ownership and permitted action properties, by varying the number of tenants with fixed 30,000 users. In all cases, number of domains is 500.

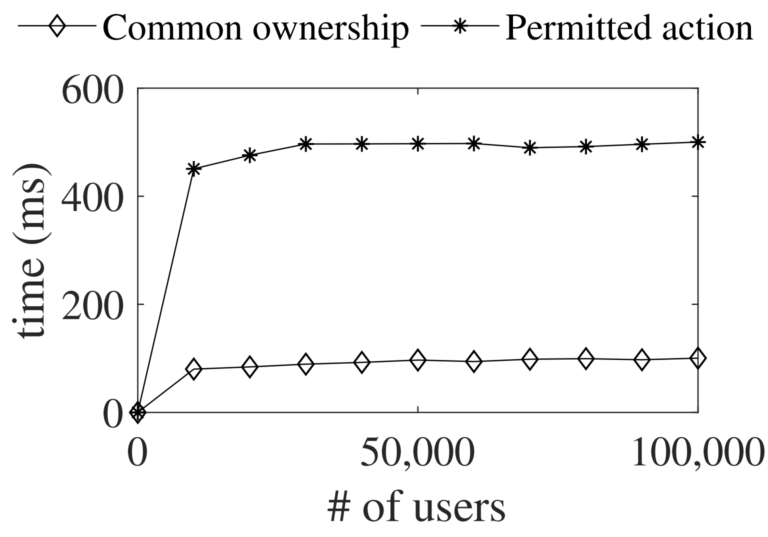


Figure 3.13: Total time required to perform the runtime phase of the common ownership and permitted action properties, by varying the number of users with fixed 5,000 tenants. In all cases, number of domains is 500.

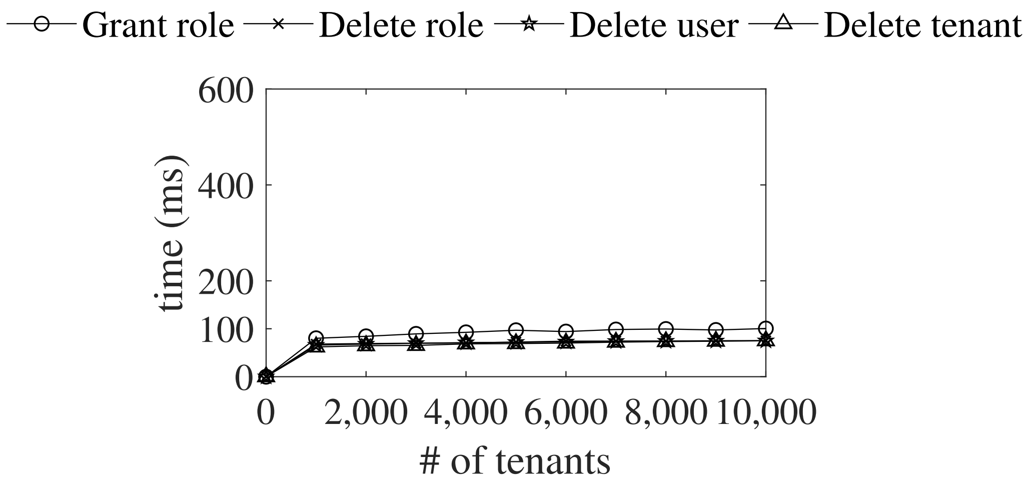


Figure 3.14: Time required to perform the runtime phase of the common ownership property for different events, by varying the number of tenants with 10 users per tenant. In all cases, number of domains is 500.

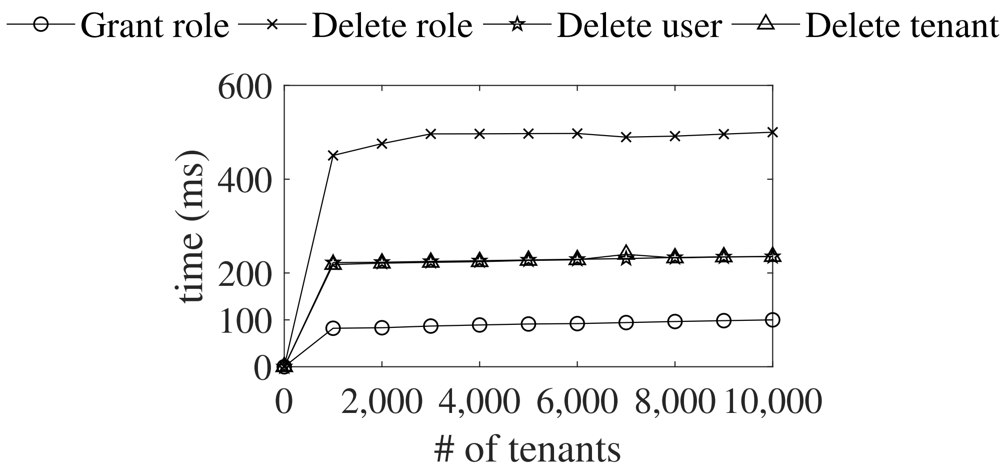


Figure 3.15: Time required to perform the runtime phase of the permitted action property for different events, by varying the number of tenants with 10 users per tenant. In all cases, number of domains is 500.

time in milliseconds required for data collection, data processing and compliance verification during the initialization phase to verify the common ownership property for different cloud sizes (e.g., the number of users). The obtained results show that the verification execution time is less than two seconds for fairly large numbers of users. Knowing that this task is performed only once upon each request, we believe that this is an acceptable overhead for verifying a large setup. Figure 3.11 shows the total time required for separately performing the initialization phase for common ownership and minimum exposure properties, and also for both of the properties together. We can easily observe that the execution time is not a linear function of the number of security properties to be verified. In fact, we can see that verifying more security properties would not lead to a significant increase in the execution time. Figures 3.12 and 3.13 show the total time required for separately performing the runtime phase for common ownership and permitted action properties for different cloud sizes. The obtained results support that the verification time for the permitted action (i.e., up to 500 ms) is more than that of the common ownership (i.e., up to 100 ms). Figures 3.14 and 3.15 further depict the effects of different events on the runtime phase for different security properties, while varying the number of tenants up to 10,000. As our runtime phase is an incremental approach and verifies mainly parameters of the events (as shown in Figure 3.9), the size of the cloud affects the verification time very less.

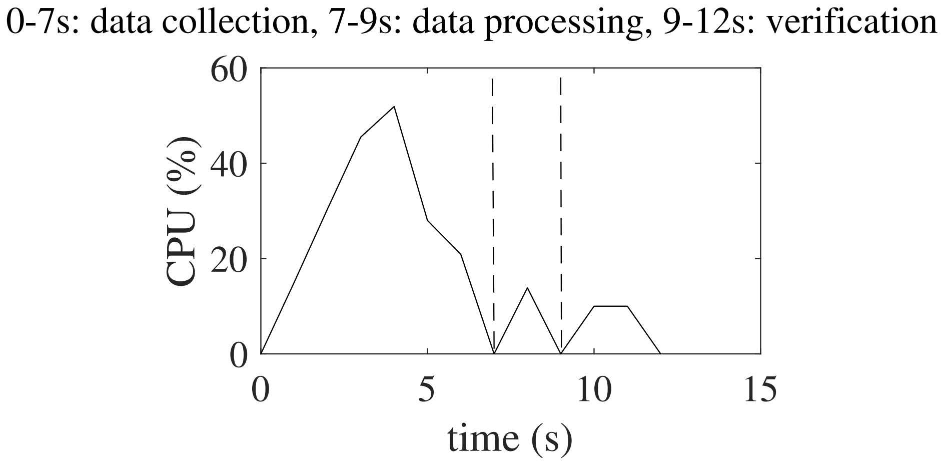


Figure 3.16: CPU usage for each step during the initialization phase over time with 60,000 users, 10,000 tenants and 500 domains for the common ownership property.

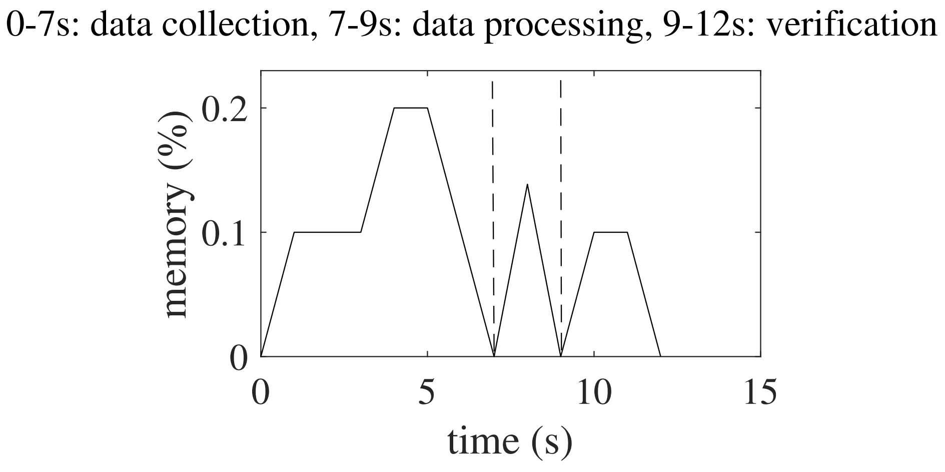


Figure 3.17: Memory usage for each step during the initialization phase over time with 60,000 users, 10,000 tenants and 500 domains for the common ownership property.

Our third experiment (see Figures 3.16, 3.18 and 3.20) measures the CPU usage (in %) during the initialization and runtime phases. Figure 3.16 depicts the fact that the data collection step requires significantly higher CPU usage than the other two steps. However, the average CPU usage for data collection is 30%, which is reasonable since the verification process lasts only a few seconds. Note that, we conduct our experiment in a single PC; if the security properties

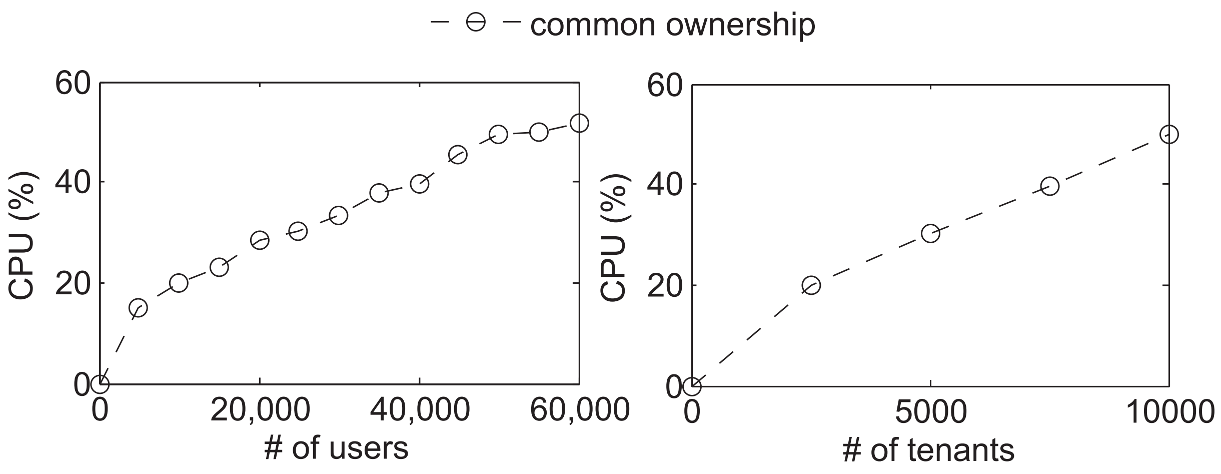


Figure 3.18: Peak CPU usage to perform the initialization phase for the common ownership property by varying the number of users with 10,000 tenants (left) and number of tenants with 60,000 users (right). In both cases, there are 500 domains.

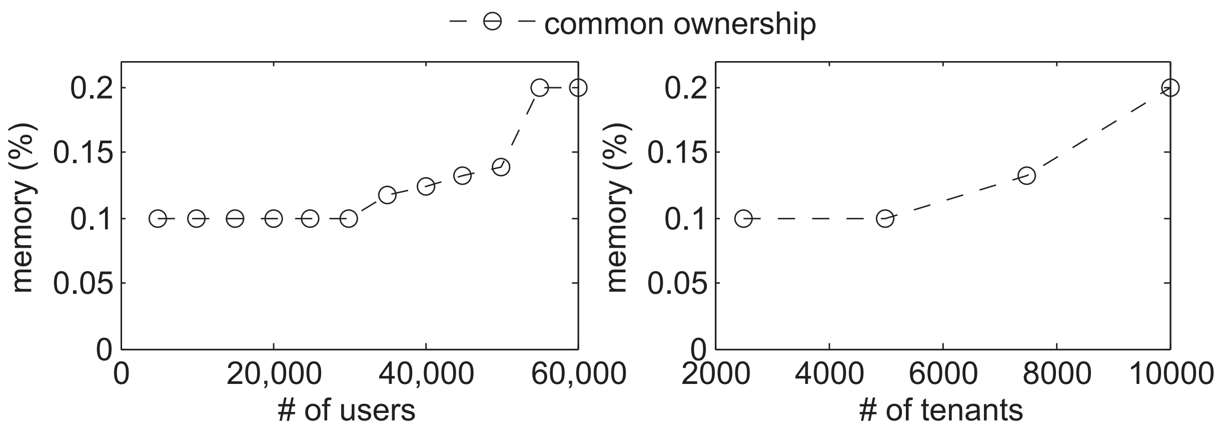


Figure 3.19: Peak memory usage to perform the initialization phase for the common ownership property by varying the number of users with 10,000 tenants (left) and number of tenants with 60,000 users (right). In both cases, there are 500 domains.

can be verified through concurrent independent Sugar executions, we can easily parallelize this task by running several instances of Sugar on different VMs in the cloud environment. Thus, performing verification using the cloud or even with multiple servers possibly reduces the cost significantly. For the other two steps, the CPU cost is around 15%. In Figure 3.18, we measure the peak CPU usage (in %) consumed by different steps while verifying the common ownership property. Accordingly, the CPU usage grows almost linearly with the number of users and tenants. We observe a significant reduction in the increase rate of CPU usage for datasets with 45,000 users or more. Note that, other properties show the same trend in CPU consumption, as

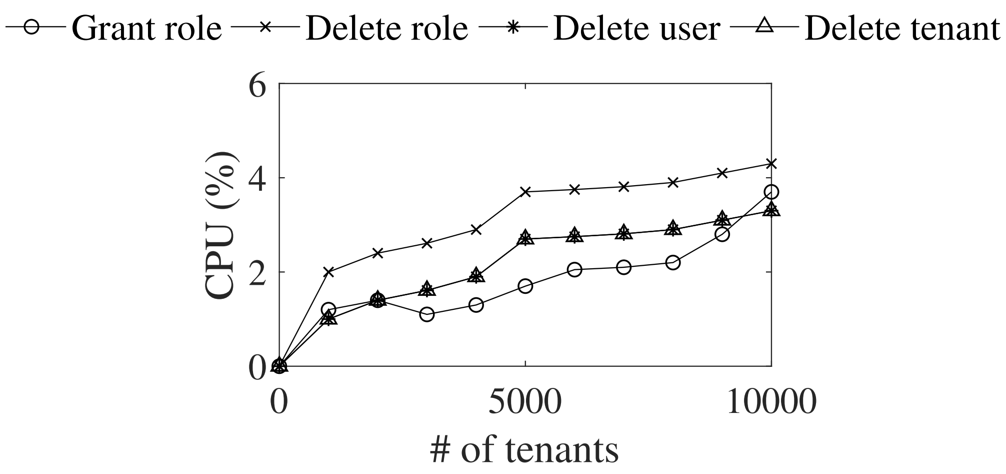


Figure 3.20: Peak CPU usage (left) and peak memory usage (right) to perform the runtime phase of the common ownership property for different events, by varying the number of tenants with 10 users per tenant. In all cases, the number of domains is 500.

the CPU cost is mainly influenced by the data collection step. Figure 3.20 shows that runtime phase expectedly requires negligible CPU (i.e., up to 4.7%) in comparison to the initialization phase.

Our final experiment (Figures 3.17, 3.19 and 3.21) measures the memory usage during the initialization and runtime phases. Figure 3.17 shows that the data collection step is the most costly in terms of memory usage. However, the highest memory usage observed during this experiment is only 0.2%. Figure 3.19 shows that the rise in memory consumption is only observed beyond 50,000 users (left) and 8,000 tenants (right). We investigated the peak in the memory usage for 50,000 users and it seems that this is due to the internal memory consumption by Sugar. Figure 3.21 depicts the memory usage by our runtime phase and further supports that the runtime phase deals with significantly smaller data set (as also shown in Figure 3.9).

Although we report results for a limited set of security properties, the use of formal methods for verifying these properties shows very promising results. Particularly, we show that the time required for our solution grows very slowly with the number of security properties. As seen in Figure 3.11, an additional security property adds only about three seconds to the initial effort. Therefore, we anticipate that verifying a large list of security properties would still be practical.

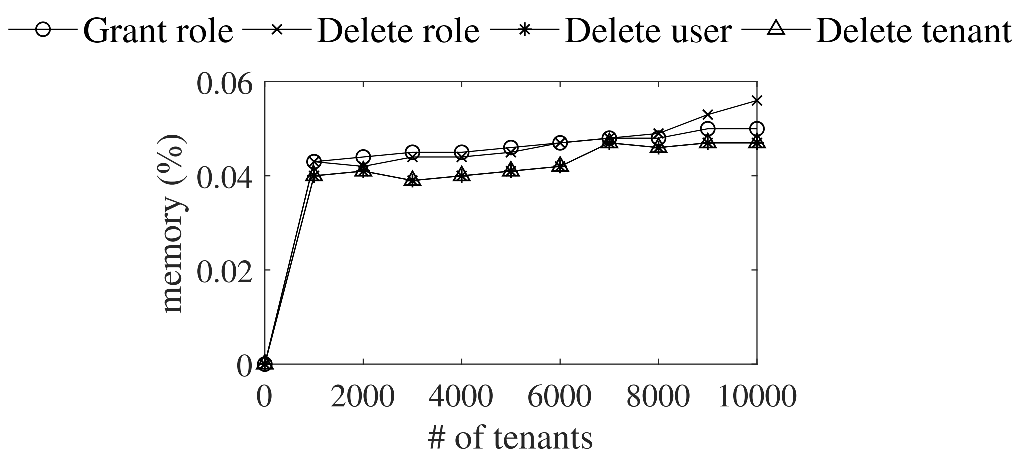


Figure 3.21: Peak CPU usage (left) and peak memory usage (right) to perform the runtime phase of the common ownership property for different events, by varying the number of tenants with 10 users per tenant. In all cases, the number of domains is 500.

3.6 Discussion

Adapting to Other Cloud Platforms. Our solution is designed to work with most popular cloud platforms (e.g., OpenStack [89], Amazon EC2 [5], Google GCP [38], Microsoft Azure [77]) with a minimal one-time effort. Once a mapping of the APIs from these platforms to the generic event types are provided, rest of the steps in our auditing system are platform-agnostic. Table 3.6 enlists some examples of such mappings.

Handling Extreme Situations. There might be some extreme cases where our solution may act differently. For instance, if the cloud logging system fails resulting from any disruption or failure in the cloud, then our auditing system will be affected. As in our threat model (in Section 3.2.3), we assume that our solution relies on the correctness of the input data (including the logs) from the cloud. Any other failure or disruption in the cloud must be detected by our system. Also, if our system including the formal verification tool (e.g., Sugar) fails, till now there is no self-healing or self-recovery feature. Therefore, in this extreme case, the efficiency of the system will be affected and a full (instead of incremental) verification will be required to recover from this failure.

The Rationale behind our Incremental Approach. The incremental verification of a given

Generic Event Type	OpenStack [89]	Amazon EC2-VPC [5]	Google GCP [38]	Microsoft Azure [77]
create user	POST /v3/users	aws iam create-user	gcloud beta compute users create	az ad user create
delete user	DELETE /v3/users/{user_id}	aws iam delete-user –user-name	gcloud beta compute users delete	az ad user delete
assign role	/v3/users/{user_id} /roles/{role_id}	aws iam attach-role-policy	gcloud projects add-iam-policy-binding	az role assignment create
create role	POST /v3/roles	aws iam create-role	gcloud beta iam roles create	az role definition create
delete role	DELETE /v3/roles/{role_id}	aws iam delete-role	gcloud beta iam roles delete	az role definition delete

Table 3.6: Mapping event APIs of different cloud platforms to generic event types.

security property involves instantiating and solving the security property predicates for the affected elements in the supports of the involved relations (as stated in Section 3.3.4). Therefore, any modification to the system data resulted from cloud events (e.g., grant role, delete role, etc.) would not directly change the security property expression itself although the corresponding support may need to be changed. For example, if a role is granted, the only change is that the relationships involving the entity role in the model would include a new element in their supports.

3.7 Related Work

Table 4.6 compares existing related works for the cloud. Firstly, the existing approaches are categorized into: retroactive, intercept-and-check and proactive. Secondly, these works mainly cover three major levels: user, network and virtual infrastructure. Thirdly, we identify several features to differentiate our work from others. The *no-future-plan* feature is checked when a proactive or intercept-and-check approach does not require any future change plan; for retroactive approaches this feature is not applicable (N/A). The *first-order-logic* feature is checked when a work can verify any security property that is expressed in first order logic. We also identify the works that support verification on RBAC, ABAC and SSO. Finally, the most works are specifically designed for a particular cloud platform. In summary, our work differs from the existing works as follows. First, this work offers an intercept-and-check approach to audit the user-level at runtime. Second, only our work supports security properties related to RBAC, ABAC and SSO. Thirdly, our approach requires no future change plan for the verification process. Finally, this work explains how it can be adapted to other cloud platforms.

Verifying security compliance in the cloud has recently been explored. For instance, in [70, 73], formal auditing approaches are proposed for retroactive security compliance checking in the cloud. The works in [108, 24] also support retroactive auditing. Unlike our proposal, those approaches can detect violations only after they occur, which may expose the system to high risks. There are several works (e.g., [62, 60, 96]) offering runtime security check in the cloud. VeriFlow [62] and NetPlumber [60] monitor network events and check network policies at runtime to capture bugs before or as soon as they occur. Designing cloud monitoring services based on security service-level agreements have been discussed in [96]. There are several other works that target auditing data location and storage in the cloud (e.g., [110, 58, 51, 112]) and others target infrastructure change auditing (e.g., [108, 25]).

Several existing efforts (e.g., [30, 2, 7, 47]) verify access control policies at the design time. In most of these works, cloud-related user-level security properties are not considered. There are some efforts (e.g., [36, 107, 3, 40]) towards proposing multi-domain/tenant access control models. Gouglidis et al. [41] utilize model-checking to verify custom extensions of RBAC with multi-domain against security properties. Lu et al. [68] use set theory to formalize policy conflicts in the context of inter-operation in the multi-domain environment. In contrast to those works, we are dealing with the verification of not only the policies but also their implementations, which involve efficient techniques to collect, process, and verify large amount of data at runtime.

There are few other works (e.g., [71, 16, 88]) offering runtime security policy checking in the cloud. Our previous work in [71] proactively verifies security compliance very efficiently through pre-computation by utilizing dependency models. However, there are several properties (e.g., minimum exposure, proper constraint checking, session time-out) which cannot be captured through the dependency models. On the other hand, this work is capable of verifying a wider range of properties. Weatherman [16] aims at mitigating misconfigurations and enforcing

Proposals	Approaches			Coverage			Features					Platforms			
	Retroactive	Intercept-and-check	Proactive	User-level	Network-level	Virtual Inf.	No-future-plan	First-order-logic	Verifying RBAC	Verifying ABAC	Verifying SSO	Supporting OpenStack	Supporting Azure	Supporting VMware	Adaptable to others
CloudRadar [15]	-	●	-	-	-	●	N/A	-	-	-	-	-	-	●	-
Weatherman [16]	-	●	●	-	-	●	-	-	-	-	-	-	-	●	-
Majumdar et al. [73]	●	-	-	●	-	-	N/A	●	●	-	-	●	-	-	-
Madi et al. [70]	●	-	-	-	●	●	N/A	●	-	-	-	●	-	-	-
Majumdar et al. [71]	-	●	●	●	●	●	●	-	●	-	-	●	-	-	-
Doelitzscher et al. [25]	●	-	-	-	-	●	N/A	-	-	-	-	●	-	-	-
Ullah et al. [108]	●	-	-	-	-	●	N/A	-	-	-	-	●	-	-	-
Congress [88]	●	●	●	●	●	●	-	-	-	-	-	●	-	-	-
SecGuru [12]	●	-	-	-	●	-	N/A	●	-	-	-	-	●	-	-
QRadar [49]	●	-	-	●	●	●	N/A	-	-	-	●	-	-	●	-
This work	-	●	-	●	-	-	●	●	●	●	●	●	-	-	●

Table 3.7: Comparing different existing solutions. The symbol (●) indicates that the proposal offers the corresponding feature.

security policies in a virtualized infrastructure. However, expensive computations after each critical event causes significant delay. Our work overcomes this limitation by using incremental verification. Congress [88] is an OpenStack project offering similar features as Weatherman. Several industrial efforts include solutions to support auditing in specific cloud environments. For instance, SecGuru [12] audits Microsoft Azure datacenter using the SMT solver Z3. IBM provides a monitoring tool integrated with QRadar [49], to collect and analyze events in the cloud. Amazon offers web API logs and metric data to their AWS clients by AWS CloudWatch & CloudTrail [6] to facilitate auditing. Although those efforts may assist auditing tasks, we support a wider set of user-level security properties.

3.8 Conclusion

Despite existing efforts, runtime security auditing in cloud still faces many challenges. In this work, we proposed a runtime security auditing framework for the cloud with special focus on the user-level including different access control and authentication mechanisms e.g., RBAC, ABAC and SSO, and we implemented and evaluated the framework based on OpenStack, a popular cloud management system. Our experimental results showed that our incremental approach in runtime verification reduces the response time to a practical level (e.g., less than 500 milliseconds to verify 100,000 users). This response time is satisfactory when the management operations are manually done by the administrators. The current approach would be insufficient to provide the same response time in the case of batch execution for management operations, when these operations are executed in short intervals and if the subsequent operations impact the same property. As future work, to address this use case, we consider maintaining a scheduler including an event queue with different threads for different tasks in order to verify properties concurrently and therefore reduce the response time in this case. Also, verifying sequence of events, in addition to our current method of verifying single event, may further reduce the impact of this concern. Furthermore, currently we do not consider any incorrectness in the formalization of the security properties. In out future work, we intend to address this concern by validating these formalizations through unit-test-like approaches. Additionally, in our current method, an update in the security property requires to re-launch the auditing process from the initialization phase. Our future work will aim to provide an incremental way of updating security properties.

Chapter 4

Proactive Security Auditing through Caching and Pre-Computation

4.1 Introduction

The multi-tenant and self-service nature of clouds usually implies significant operational complexity, which may prepare the floor for misconfigurations and vulnerabilities leading to violations of security compliance. Therefore, the security auditing w.r.t. security standards, policies, and properties, is desirable to both cloud providers and users. Evidently, the Cloud Security Alliance (CSA) has recently introduced the Security, Trust & Assurance Registry (STAR) for security assurance in clouds, which defines three levels of certifications (self-auditing, third-party auditing, and continuous, near real-time verification of security compliance) [19]. However, above-mentioned complexities coupled with the sheer size of clouds (e.g., a *decent-size* cloud is said to have around 1,000 tenants and 100,000 users [92]) implies one of the main challenges in cloud security auditing, specifically the scalability and response time, especially in near real-time verification of security compliance.

To this end, existing approaches can be roughly divided into three categories (a more detailed review of related work will be given in Section 4.7). First, the *retroactive* approaches (e.g., [70, 73]) catch compliance violations after the fact by verifying different configurations and logs of the cloud. 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., [16, 88]) verify the compliance of each user request before either granting or denying it, which may lead to a substantial delay to users' requests, as will be further illustrated later in this section. Third, the *proactive* approaches in [16, 88] verify user requests in advance, and somewhat overcome the limitations of first two approaches.

However, existing proactive solutions still pose following limitations. i) To avoid the significant delay at runtime, existing works (e.g., Weatherman [16]) alternatively require a future change plan (which means configuration changes scheduled in near future) in advance from the tenant admins; however, this requirement is not always practical for cloud due to its dynamic and ad-hoc nature. ii) Moreover, non-optimization of the verification effort (e.g., repeated computation for recurrent events) results longer response time for the existing methods. iii) Furthermore, current proactive solutions (e.g., [16, 71]) fully rely on cloud tenants in providing a complete list of security critical events (i.e., events that potentially may lead to a violation) to ensure accurate proactive auditing; which lacks adaptability and may be error-prone and tedious for tenants. iv) Finally, concurrent event instances at runtime may cause malfunctions in some proactive auditing methods (e.g., [71]).

Motivating Example. Through this example, we further illustrate the above-mentioned limitations and motivate our solution. Fig. 4.1 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 the critical event, which can potentially breach a security property. Here, we consider the "no bypass" security property for the anti-spoofing mechanisms in the cloud, which can be violated by real world vulnerabilities (e.g., OpenStack vulnerability [86] [1]). We highlight the major limitations of the existing approaches and position our solution as follows.

- A typical retroactive auditing is conducted periodically (e.g., at time t_2 and t_3) within a certain interval, and such auditing usually takes several seconds (e.g., eight seconds for auditing

[1]OpenStack [89] is an open-source cloud management platform.

10,000 tenants as reported in [70, 73]); which allow attackers to exploit the vulnerable systems for a considerable amount of time with irreversible damages (e.g., DoS and leakage of sensitive information).

- A typical intercept-and-check approach overcomes the above-mentioned limitation of retroactive auditing, however, it starts the verification only after the *update port* event occurs, and thus results in a significant delay (e.g., four minutes as reported in [16]).
- To avoid such runtime delay, existing proactive approaches (e.g., [16, 88]) require a future change plan (e.g., *create port, create VM (2) and update port*) before time t_0. However, providing such a concrete future plan in advance is not always practical considering the dynamic and ad-hoc nature of cloud.
- Moreover, previous proactive solutions assume that a comprehensive list of critical events (e.g., *update port*) are known in advance. This assumption might not be practical, and it does not adapt to dynamically changing security requirements. Also, current solutions do not provide any practical method to identify such critical events (out of more than 400 cloud event types [89]); This limitation may lead to an incomplete list of critical events which later may affect the accuracy of the proactive auditing.
- Finally, current solutions would treat the two consecutive occurrences of the VM creation event (denoted as *create VM (2)* event in Fig. 4.1) as independent events and repeat almost the same pre-computation tasks.

In this work, we propose a proactive security auditing system, namely, *ProSAS*, which addresses above-mentioned limitations and hence, provides significant improvements in both efficiency and accuracy of runtime auditing. The efficiency improvement is mainly achieved by performing costly verification steps proactively, as soon as the system is a few steps ahead of a critical event. The efficiency is further improved by reducing the efforts for recurrent events with the use of a caching mechanism, which keeps track of our proactive steps on recently occurred events. Additionally, unlike existing proactive solutions, ProSAS can handle concurrent events in a sequential manner using a locking mechanism. On the other hand, the better accuracy of ProSAS is possible as we no longer only rely on initial identification of security critical events, and instead propose a new feedback module, which leverages a retroactive approach

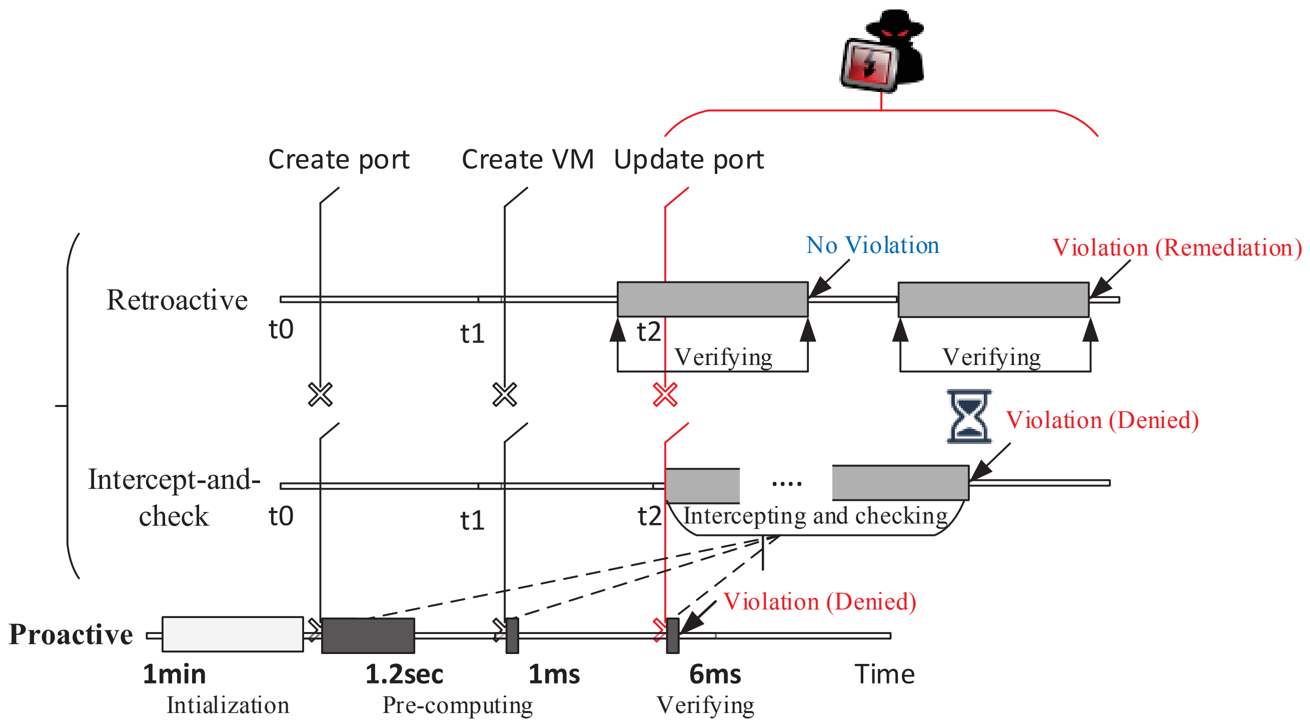


Figure 4.1: Comparison of the execution time of our solution with the typical intercept-and-check and retroactive approaches.

using cloud states (e.g., configurations) to progressively improve the list of critical events.

The main contributions of our work are as follows.

- To the best of our knowledge, this is the first pre-computation-based proactive security auditing approach for clouds. As demonstrated by our implementation and experimental results, the proposed system, ProSAS, provides an automated, efficient, and scalable solution for different cloud platforms to increase their transparency and accountability to tenants.
- ProSAS requires significantly less time than any existing proactive solutions to audit. First, our dependency models help to identify the relationship between cloud events, and to distribute the verification overhead over the chain of events to improve the efficiency of the system. Second, the ProSAS caching mechanism significantly optimizes the pre-computation and verification efforts in case of recurrent events.
- Unlike existing proactive approaches, ProSAS provides a higher rate of accuracy by using an adaptive and more comprehensive way of critical event identification. To this end, ProSAS provides a hybrid solution where a proactive auditing system is supported by a retroactive

auditing tool at the backend to more accurately identify critical events over time.

4.2 Models

This section defines our threat model and presents the dependency models.

4.2.1 Threat Model

We assume that the cloud infrastructure management systems i) may have implementation flaws, misconfigurations and vulnerabilities that can be potentially exploited to violate security properties specified by the cloud tenants, and ii) may be trusted for the integrity of the API calls, event notifications, logs 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., [10, 64, 101, 102]). Though our framework may assist to avoid any violation of specified security properties due to either misconfigurations or exploits of vulnerabilities, our focus is not to detect specific attacks or intrusions. We focus on attacks directed through the cloud management interfaces (e.g., CLI and GUI), and any violation bypassing such interfaces is beyond the scope of this work. Our proactive solution mainly targets certain security properties, which would require a sequence of operations. To make our discussions more concrete, the following shows an example of in-scope threats based on a real vulnerability.

Running Example. Real world vulnerabilities, such as the one in OpenStack [86][1], can be exploited to bypass anti-spoofing mechanisms. These mechanisms are implemented in OpenStack using firewall rules enforcing tenants' layer 3 network isolation. Fig. 4.2 shows the attack scenario to exploit this vulnerability. The exploit consists in changing the device owner (step 3 in Fig. 4.2) of an instance's port to a string starting with the word `network`, right after the instance is created (steps 1 & 2) and just before security group gets attached to it (race condition). As a result, the firewall rules of the compute node are not applied to that port, since it is treated as a network owned port. Consequently, a malicious tenant can launch IP, MAC, and DHCP spoofing attacks (step 4).

[1]OpenStack [89] is an open-source cloud infrastructure management platform.

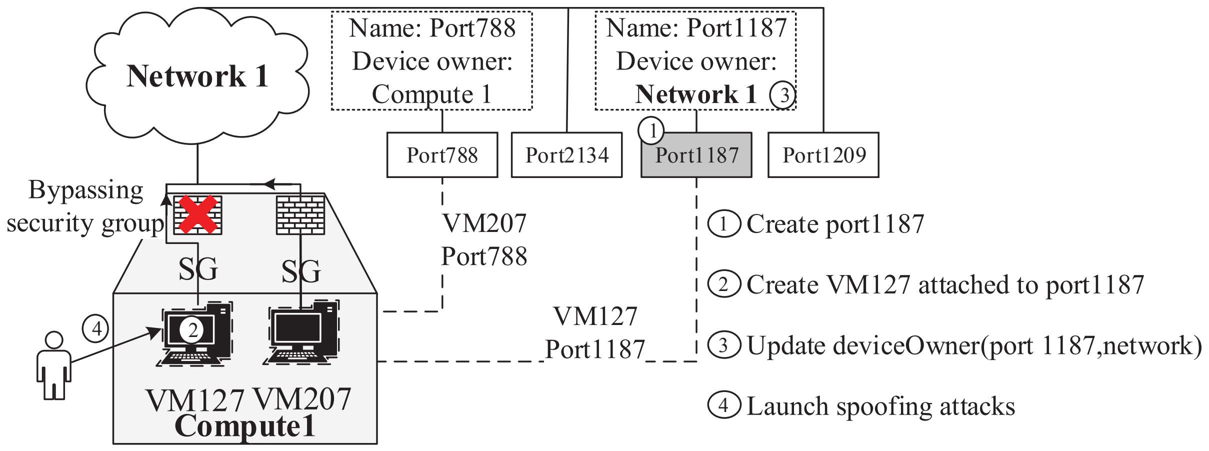


Figure 4.2: An exploit of a vulnerability in OpenStack [86], leading to bypassing the anti-spoofing mechanism.

4.2.2 Dependency Models

Figures 4.3 and 4.4 illustrate the two dependency models that we derive for an OpenStack-managed cloud covering virtual infrastructure (Fig. 4.3) and user access control (Fig. 4.4). Each dependency model can be used for proactively auditing multiple security properties. We validate these dependency models based on extensive study of OpenStack APIs [91] from different related OpenStack services (e.g., Neutron, Nova, and Keystone) and Open vSwitch [32]. For the user access control model, we are inspired by the OSAC model by Tang et al. [107]. To build intuitions of these models, we start by providing an example on how the cloud infrastructure dependency model (see Fig. 4.3) allows us to relate actual management operations or events happening in the cloud to the “no bypass” security property presented in Section 4.2.1.

Example 8 *According to the attack scenario presented in Fig. 4.2, the critical management operation that leads to the violation of the “no bypass” security property is* `update port`*. The model in Fig. 4.3 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. 4.3). As it can be seen in Fig. 4.3,* `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`).

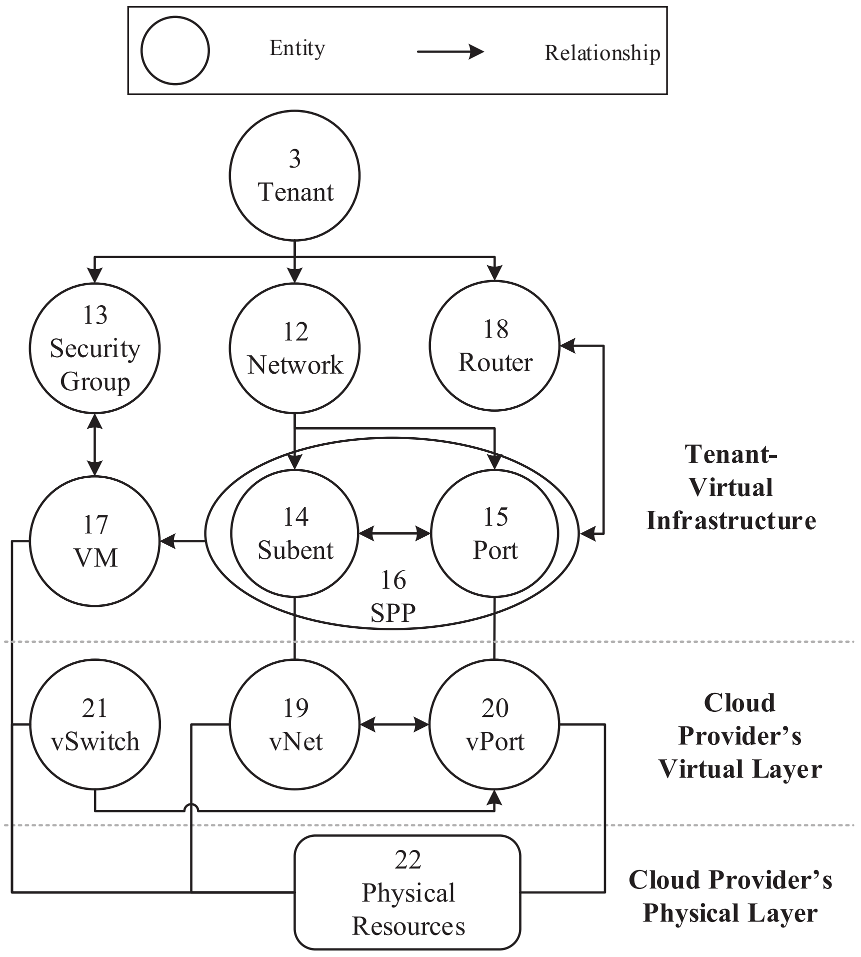


Figure 4.3: Dependency model of cloud infrastructure

As the `create port` *and* `create VM` *operations are closely related to the actual critical operation (*`update port`*), our model captures this dependency relationship and aids to avoid the security violation by starting preparation from the* `create port` *operation. Furthermore, these operations in turn depend on the existence or creation of a subnet, a network and a tenant. This induces a chain of dependencies between a set of events that could be related to this security property.*

Formally, the dependency model is a graph, $G = (V,E)$, where vertices V_i are individual cloud entities (e.g., user, role, tenant, port, VM, etc.) or groups of entities (e.g., (port, subnet) pair) and edges E_{ij} are dependency relationships between connected vertices. These relationships are activated by events/operations in the cloud (e.g., create/delete port, attach VM to a

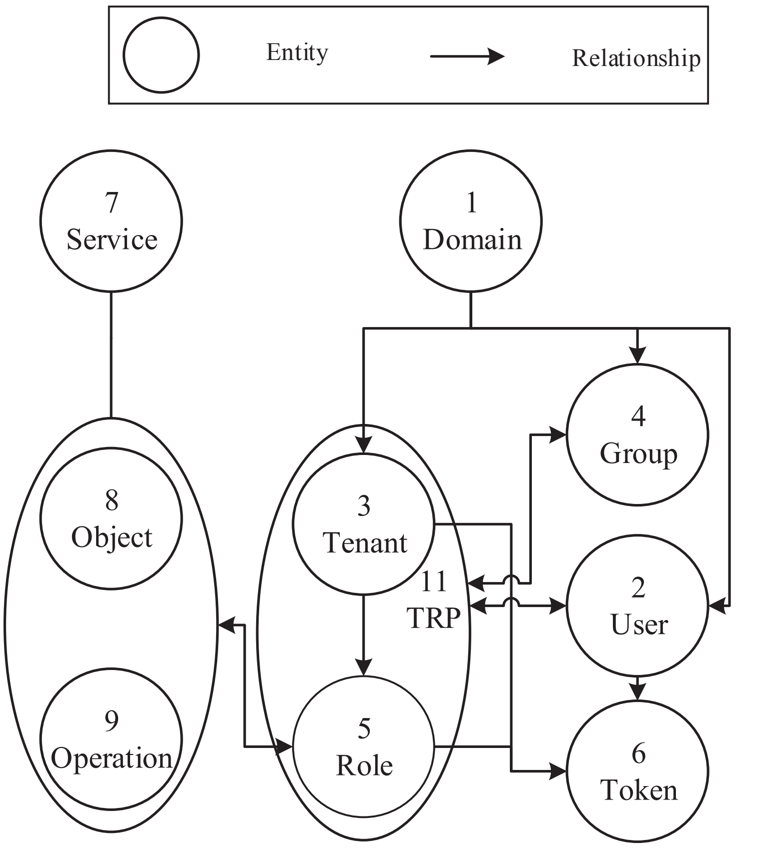


Figure 4.4: Dependency model of access control management

(port, subnet) pair, etc.). We use edges' attributes to store information on which security property are associated with the events that are related to the edge and on the type of this event per property. We define four kinds of relationships. We use different types of edges (unidirectional, bidirectional, or non-directional) to differentiate the following relationships based on their semantics.

- *Precedence* relation, represented by a unidirectional edge, such that $E_{ij} = (V_i, V_j)$ denotes that the entity V_i must exist before creating entity V_j within V_i.
- *Association* relation, represented by a bidirectional edge, such that $E_{ij} = (V_i, V_j)$ denotes that entities V_i and V_j should both exist (i.e., created) to be able to make any association between them.
- *Mapping* relation, represented by a non-directional edge, such that $E_{ij} = \{V_i, V_j\}$ denotes a correspondence relationship between entities V_i and V_j existing in different layers in the cloud.

- *Reflexive* relation (omitted in the graph), representing a relation from a node to itself such as updating attributes of the node.

We leverage the knowledge captured by these dependencies to appropriately identify the intercepted events, relate them to the security property, identify their roles in the context of proactive compliance verification, and determine the distance to a critical state. More details are provided in Section 4.3. It is worth noting that these models are static and do not depend on the execution context of the cloud. They consist of a relatively small set of entities and relationships. For example, for Neutron, Nova, and Keystone services, we enumerated only 86 different entities and about 400 events that are relevant to configuration changes and management.

Property	**Critical Event (CE)**	**Watchlist Event (WE)**	**Watchlist per tenant**
No bypass [18]	update port (15,15)	create VM (16,17) create port (12,15)	Ports except VM ports
Port consistency [53, 18]	create vPort (21,20)	create port (12,15)	ports at tenant layer
No abuse of resources [18]	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 [18]	attach port to a router (16,18)	create router (3,18)	router-tenant pair
Port isolation [53, 18]	add vPort to vNet (19,20)	create vNet (14,19)	vNets in a subnet
No co-residency[1] [53, 18]	create VM (16,17), migrate VM (17,22)	create VM (16,17)	Hosts with no conflicting VMs
		migrate VM (17,22)	

Table 4.1: An excerpt of the security properties supported by the cloud infrastructure model with their corresponding critical and watchlist events, and the watchlist contents.

Tables 4.1 and 4.2 enlist excerpts of the security properties supported by the cloud infrastructure dependency model. Here, we categorize events mainly into two types: critical event (CE) and watchlist event (WE). A CE (e.g., `update port`) potentially leads to the violation of the associated property. A WE corresponds to an event that impacts the content of the watchlist associated with the security property (e.g., `create port` and `create VM`). The third type of event is the trigger event (TE), which is neither critical nor watchlist-related, however is useful to determine the distance to a critical state. Note that an event may have multiple types considering different security properties. For example, `create VM` is a WE event for the *no bypass* property, but it is of type CE for the *no co-residency* property.

4.3 Proactive Security Auditing System (ProSAS)

This section details our proactive security auditing system.

4.3.1 Overview

The security auditing process of ProSAS involves following major steps. First, we intercept every cloud management event instance at runtime, and identify the type (i.e., critical or non-critical) of the intercepted event. Second, for a non-critical event, (without holding the intercepted event blocked) we consult the pre-computation cache, which stores recently occurred event types for which pre-computation is required and their corresponding actions to be performed, so that ProSAS can pre-compute the necessary conditions and build the watchlist incrementally. In case, the intercepted event type is not in the cache, the pre-computation is only performed, if the distance of this event type from a critical event is less than a user-defined threshold (how to set this threshold is discussed later). Third, for a critical event, (while holding the intercepted event blocked) ProSAS verifies the parameters of the intercepted event consulting the verification cache, which stores recently updated entries of the watchlist, and enforces the decision (e.g., allow or deny) accordingly. In case the event type is not in the cache, we check the whole watchlist and enforces the result. Fourth, we identify new critical events (if any) mainly by periodically verifying the cloud configurations using a state-based verification tool (e.g., [70, 73]) and analyzing cloud logs so that ProSAS can progressively improve its accuracy.

Fig. 4.5 illustrates an overview of ProSAS. The inputs to ProSAS are security properties and

Property	**Critical Event**	**Watchlist Event**	**Watchlist per tenant**
Common role ownership [53, 18]	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 [52, 81]	grant role (2,11)	delete role (3,5)	counter for each role
		deny role (2,11)	
		grant role (2,11)	
Role activation [53, 18]	create token (3,6)	grant role (2,11)	user-role pair
Permitted action [53, 18]	request an operation (8,9)	create token (3,6)	token-operation pair
		grant role (2,11)	
User-access validation [53, 18]	request an operation (8,9)	create token (3,6)	token-operation pair
		grant role (2,11)	

Table 4.2: An excerpt of the security properties supported by the access control management dependency model shown in Fig. 4.4 with their corresponding critical events, watchlist-related events, and the content of the watchlists.

their corresponding critical events, and the output is the decision (allow/deny/warn) for each intercepted cloud events to enforce the compliance. In ProSAS, there exist four major components: interceptor, caching manager, proactive module, and feedback manager. The interceptor

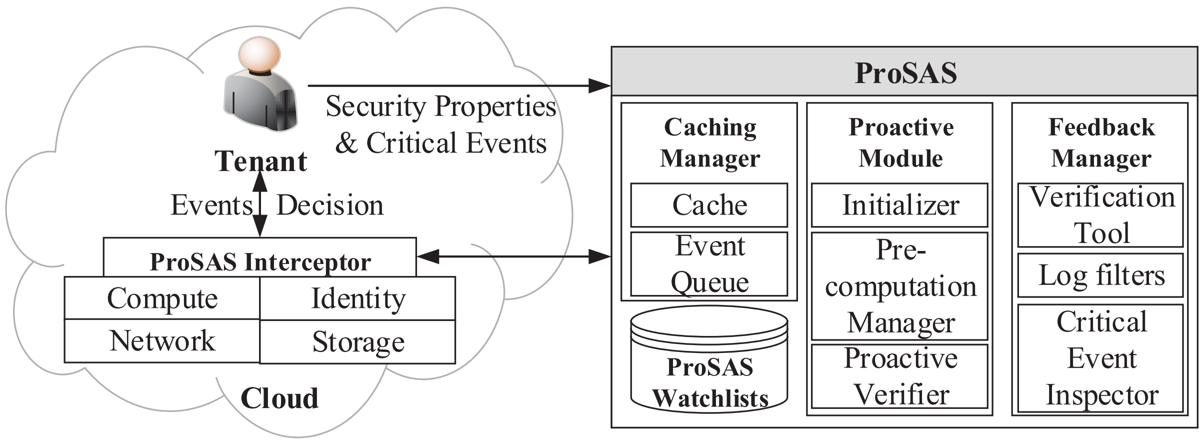


Figure 4.5: An overview of ProSAS

situates as a middleware within the cloud platform to intercept each cloud event instance for different cloud services, e.g., compute, network, storage, etc., to identify the type of the intercepted event, and after the verification process, to enforce the decision (e.g., allow or deny). The caching manager is mainly responsible to maintain two different caches for pre-computation and verification, respectively. In case the intercepted event type is found in the cache, ProSAS skips the steps in the proactive module, and responds more quickly. Otherwise, the caching manager forwards the intercepted event type and its parameters to the proactive module. Additionally, the caching manager manages a queue in cases an event occurs before the previous one is processed by ProSAS. The proactive module contains initializer, pre-computation manager and proactive verifier. First, the initializer takes a one-time effort to process the ProSAS inputs and initialize the conditions for verification. Second, the pre-computation manager incrementally updates the required conditions to preserve the security compliance. Third, the proactive verifier verifies the parameters of a critical event based on the pre-computed results and provides a decision to enforce compliance. The feedback manager is responsible to identify new critical events by leveraging existing state-based auditing methods (e.g., [70, 73]) to progressively enrich and update the initial inputs of critical events to ProSAS.

4.3.2 Interceptor

At runtime, our system intercepts all event instances performed in the cloud. The interceptor is implemented as a middleware, which is placed between the tenants and different cloud services, e.g., compute, network, storage, identity, etc. 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 4.3 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 measure the distance from a critical event.

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 4.3: Examples of OpenStack event instances and converted event types in ProSAS.

4.3.3 Caching Manager

The main responsibility of the caching manager is to prescribe necessary actions (if possible) through a caching mechanism to both proactive verifier and pre-computation manager. Maintaining the caching is performed in three steps: i) to build a cache storing actions performed by ProSAS for recent cloud events during the pre-computation and verification steps, ii) to consult the cache for the currently intercepted event, and iii) based on the cache search result, either to perform the actions (e.g., deciding verification result) by itself, or to trigger the necessary module (e.g., pre-computation manager or proactive verifier) to perform certain actions (e.g., inserting into the watchlist or verifying the watchlist). To this end, the caching manager maintains two kinds of cache: one for pre-computation and another for verification. The pre-computation cache has three attributes: recent events, the condition to check whether the distance is less than the threshold (i.e., $N\text{-}cp < N\text{-}th$), and how a watchlist update (add or remove) is performed. If the condition is false (i.e., $N\text{-}cp > N\text{-}th$), then obviously no action has been taken for the event. In case of a miss of the cache, all steps of the n-step evaluator are performed. On the other hand, the verification cache stores the recent critical events and recently added or removed watchlist contents for the corresponding critical event. A hit in the cache allows the caching manager to provide the verification result without the involvement of the proactive verifier. Otherwise, the caching manager triggers the proactive verifier to perform the verification step. The efficiency improvement due to our caching manager is evaluated in Section 4.5. Additionally, the caching manager includes a first in first out (FIFO) event queue to sequentially process concurrent events.

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

Table 4.4: An excerpt of a pre-computation cache

Example 9 Tables 4.4 and 4.5 show excerpts of pre-computation and verification caches, respectively. Table 4.4 depicts three different cases in the pre-computation caching. First, the `create VM` event is further than the threshold, and no pre-computation has been performed. Second, the `update port` event is close enough to perform an insert into the watchlist for the no bypass security property. Third, similarly for the `delete VM` event, the content from the watchlist of the no downgrade property is removed. Table 4.5 shows an excerpt of the verification cache, which stores the recently added and removed contents to/from the no_bypass and no_downgrade watchlists for the `update port`, `add/delete security group rule` events. To project the functionality of the verification cache, we depict three intercepted events: `add security group rule (2537)`,`update port (1321)` and `delete security group rule (2115)` as follows. First, LeaPS allows the `add security group rule (2537)` event just by checking the verification cache, as the VM ID 2537 is present in the recently added attribute. Second, LeaPS denies the `update port (1321)` event similarly, as the `port ID 1321` is found in the recently removed list. Third, LeaPS requires to check further in the full watchlist to verify the `delete security group rule (2115)` event, as the `VM ID 2115` is not in the cache.

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 4.5: An excerpt of a verification cache

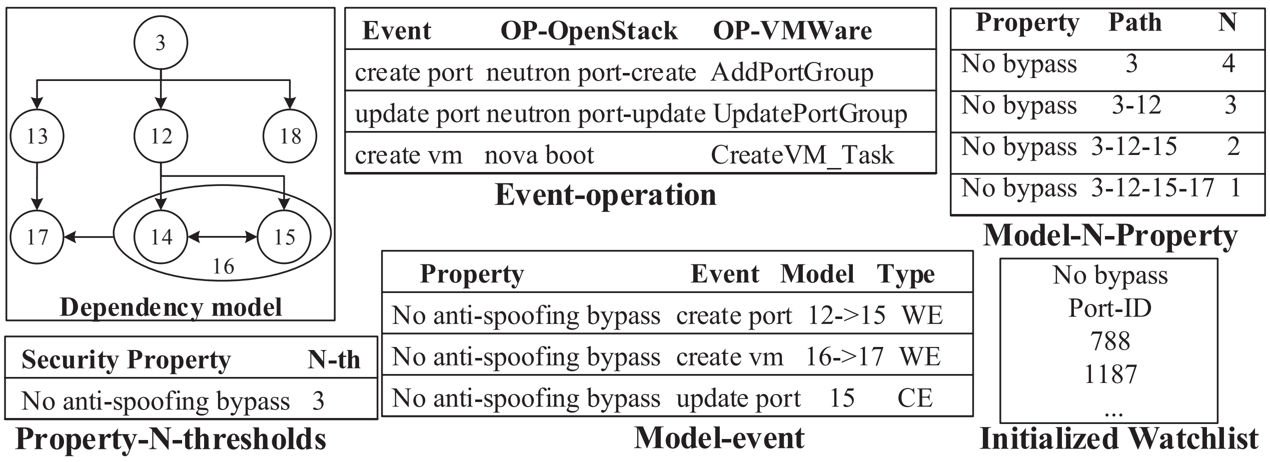
4.3.4 Proactive Module

The proactive module is mainly responsible for the proactive verification steps including initialization and incremental update of the pre-computed results (including watchlists), and verification of watchlists. In the following, we elaborate its different modules.

Initializer. The role of the initializer module is to process the ProSAS inputs, to collect necessary data from different cloud services (e.g., compute, network and storage), and to pre-process the data in order to initialize the conditions for the verification. More specifically, this module initializes following tables.

- `Event-operation:` maps event types to operations in different cloud environment to easily integrate different cloud implementations.
- `Model-event:` relates each security property with the elements of the dependency models and tenant inputs including the types of events.
- `Property-WL:` stores the specification of the contents in a watchlist for each security property.
- `Property-N-thresholds:` maps security properties and their associated thresholds (denoted as *N-th*), where thresholds are security-property-specific and inputs from the administrators. A brief guideline on choosing this threshold is provided in Section 4.6.
- `Model-N-property:` stores all possible values of *N* (denoted as *N-cp*) for each property.

Furthermore, this module initializes the watchlist content with the current cloud context.



Event-operation

Event	OP-OpenStack	OP-VMWare
create port	neutron port-create	AddPortGroup
update port	neutron port-update	UpdatePortGroup
create vm	nova boot	CreateVM_Task

Model-N-Property

Property	Path	N
No bypass	3	4
No bypass	3-12	3
No bypass	3-12-15	2
No bypass	3-12-15-17	1

Property-N-thresholds

Security Property	N-th
No anti-spoofing bypass	3

Model-event

Property	Event	Model	Type
No anti-spoofing bypass	create port	12->15	WE
No anti-spoofing bypass	create vm	16->17	WE
No anti-spoofing bypass	update port	15	CE

Initialized Watchlist

No bypass
Port-ID
788
1187
...

Figure 4.6: The output of the initializer module for the *no bypass* property.

Example 10 *Figure 4.6 shows the outcome of the initializer module for the no bypass property. The* `Event-operation` *table shows that the* `create port` *event corresponds to the* `neutron port-create` *operation in OpenStack. The* `Model-event` *table stores that the* `create port` *event is the watchlist event (WE) for the no bypass property and situates at the edge between nodes 12 and 15 in the dependency model. Also, the critical event* `update port` *for the property 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 in the figure. The minimal distance from the critical event at which our solution should react is (N-th = 3), as shown in the* `Property-N-thresholds` *table. The* `Model-N-property` *table stores all possible computed values of N taking into account the security property and the dependency model. Finally, the watchlist is initialized for the no bypass property based on data collected from the cloud. For each tenant, the watchlist is populated with the list of virtual ports that are not attached to a VM as in the* `Property-WL` *table.*

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

Example 11 *For the no bypass property, the* `Model-N-property` *table stores five entries that cover all possible values of N and the associated configuration (see Fig. 4.7). For instance, if only a tenant already exists (vertex* 3*) without yet any network, subnet, ports, and VMs, we need to observe at least five events before being able to intercept the critical event* `update port`*. If we observe an event for the creation of network within this tenant (i.e., edge* $(3,12)$*) without yet any subnet, ports, and VMs, the minimal distance to see the* `update port` *event would be* $N = 4$*. The event preceding* `update port` *is the* `create VM` *(i.e., edge* $(16,17)$*) event, and the minimal distance is one.*

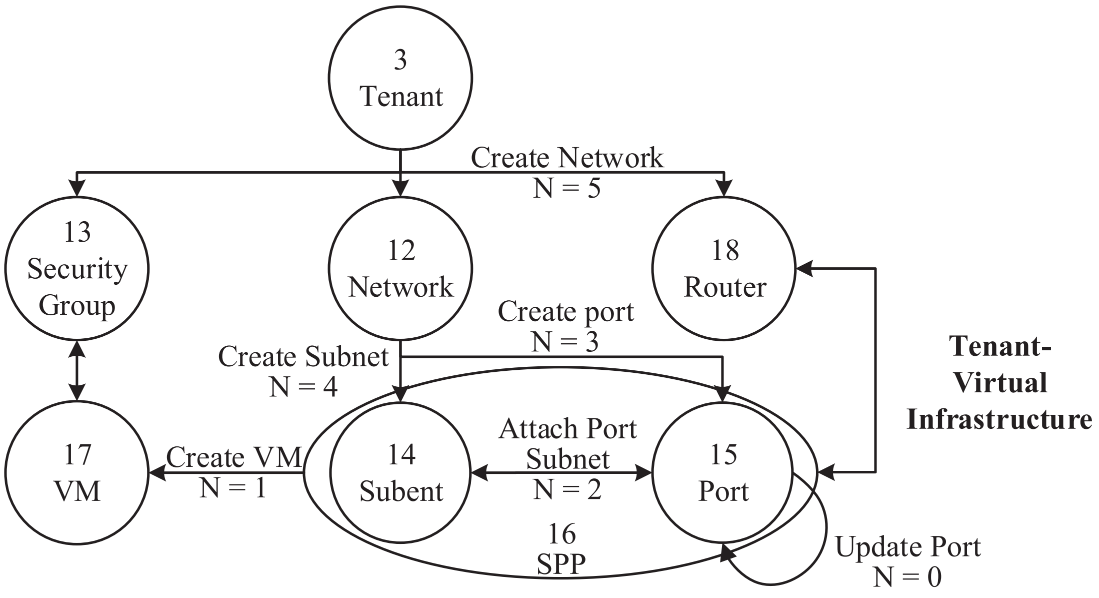


Figure 4.7: A part of the cloud infrastructure dependency model annotated with all possible values of *N* that is relevant to the *no bypass* property.

N-Step Evaluator. The N-step evaluator evaluates at runtime the value of *N-cp*, which is the estimated minimal distance from the current event to a violation, whenever the intercepted event type is a non-critical event (i.e., WE or TE) concerning a given security property. To this end, the related contextual data (e.g., corresponding tenant, network, subnet, etc.) is gathered from the cloud to determine the path to be selected from the `Model-N-property` table. Thus, we can measure the distance (i.e., *N-cp*) considering the current context. When the *N-cp* becomes equal to the threshold value (*N-th*), ProSAS starts being proactive and updates the pre-computation results corresponding to the security property with the current state of the cloud. Afterwards (i.e., *N-cp* $<$ *N-th*), whenever a WE event type is encountered, some pre-computation (e.g., updating watchlist using the values of the parameters of the intercepted event instance) is incrementally performed to ensure that the pre-computation result is up-to-date.

Proactive Verifier. The verifier is designed in a way so that by leveraging the pre-computed results it can perform the verification fast and provides the verification result in a practical time, as ProSAS halts the requested critical event. The verification of ProSAS mainly involves searching a set of values (e.g., values of the event parameters) in the corresponding watchlist. The verification decision is either to allow the operation to continue or apply the planned enforcement approach as specified by the administrator. The possible enforcement could be denying the request or requiring an approval from the admin to execute the event.

Example 12 *Figure 4.8 illustrates the runtime workflow for the no bypass property assuming that a tenant, a network and a subnet already exist. To rectify the situation described in the running example, our solution incrementally builds a watchlist with ports that are not attached to VMs, and verifies the* `update port` *operation with this watchlist. Firstly, we intercept the* `create port ID 1187` *operation, identifies the event type (which is WE), and measure the value of N (= 3), respectively, from the* `Model-event` *and* `Property-N-threshold` *tables. Since the* `create port` *event is a WE event for the no bypass property and evaluating N results in N-cp = N-th =* 3, *we add* `port ID 1187` *to the watchlist without blocking it. Secondly, we intercept* `create VM ID 127 attached to port ID 1187` *operation and measure N similarly. Then,* `port ID 1187` *is removed from the watchlist, as it is now*

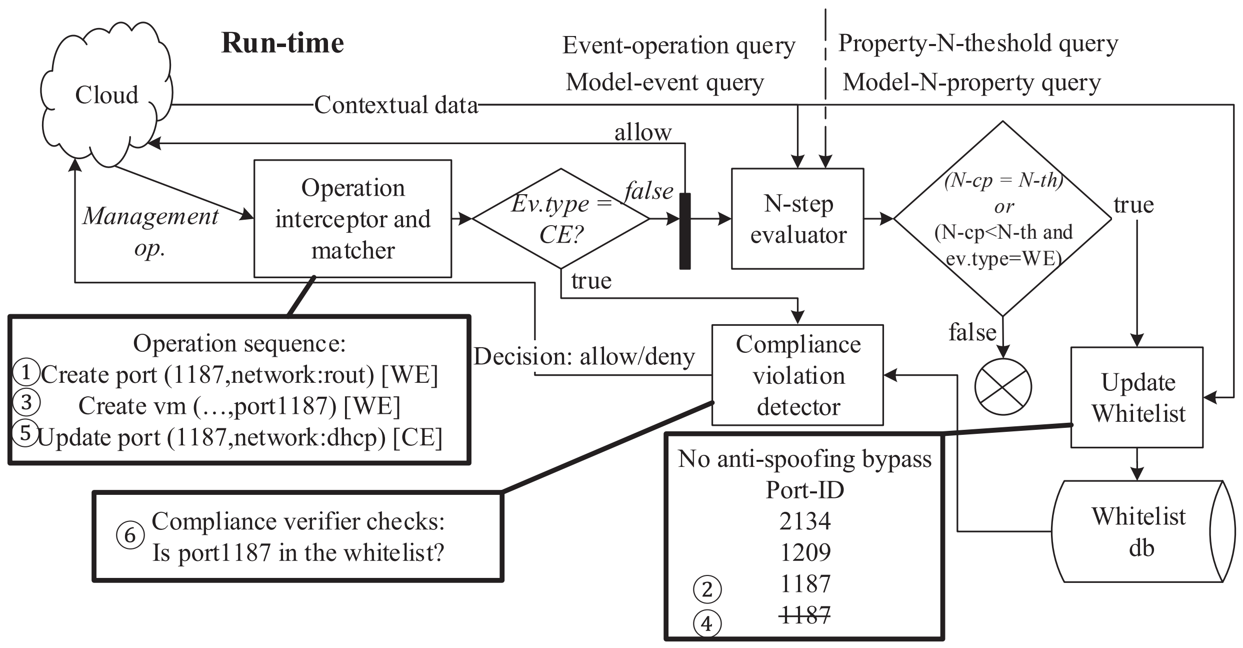

Run-time
Cloud
Contextual data
Event-operation query
Model-event query
Property-N-theshold query
Model-N-property query
allow
Management op.
Operation interceptor and matcher
Ev.type = CE?
false
true
N-step evaluator
(N-cp = N-th) or (N-cp<N-th and ev.type=WE)
true
false
Decision: allow/deny
Compliance violation detector
Update Whitelist
Whitelist db
Operation sequence:
1 Create port (1187,network:rout) [WE]
3 Create vm (…,port1187) [WE]
5 Update port (1187,network:dhcp) [CE]
6 Compliance verifier checks: Is port1187 in the whitelist?
No anti-spoofing bypass
Port-ID
2134
1209
2 1187
4 ~~1187~~

Figure 4.8: An excerpt of runtime verification of the *no bypass* property.

attached to `VM ID 127`*. Finally, after intercepting the* `update port(port ID 1187, deviceOwner, network)` *operation and measuring N, we identify that this is a CE event. Therefore, we verify with the watchlist with blocking the operation, find that* `port ID 1187` *is not in the watchlist, and hence, ProSAS recommends denial of this operation to preserve the no bypass property.*

4.3.5 Feedback Manager

The feedback manager is an interactive process in ProSAS to improve the list of critical events progressively over time; which mainly follows four major steps.

Step 1: The first step is to run a retroactive auditing tool (e.g., [70, 73]), periodically. Note that these retroactive auditing tools are state-based, and therefore, do not rely on a list of critical events. A violation detected at this step means that the corresponding critical event to the violation is not yet included to ProSAS.

Step 2: The next step is to collect data from a specific service(s) based on the detected violation of a security property in Step 1. During this step, ProSAS collects event logs from the specific cloud service(s) (e.g., network, compute and storage).

Step 3: The third step is to filter out irrelevant events from the collected logs to prepare a shortlist of candidate critical events for the violation detected in Step 1. This step first keeps only those events that occurred between last two verifications in Step 1, because the critical event must have occurred within this period. Furthermore, the irrelevant events such as generated by interfaces and not by cloud users are also eliminated from the logs. Thus, we prepare a shortlist for the next step.

Step 4: The final step is to identify the responsible critical event for the violation. This final step involves an expert, who identifies the critical event that violates a security property from the short list of Step 3 based on his/her discretion.

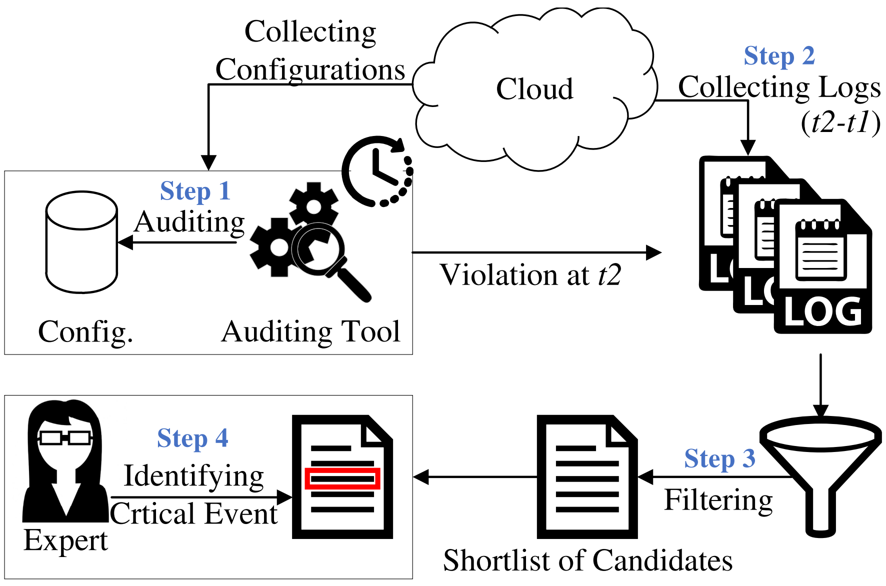


Figure 4.9: The steps of the feedback manager module

Example 13 *Figure 4.9 depicts the steps of our feedback manager. First, ProSAS periodically collects cloud snapshots (e.g., at time t_1 and t_2), and verifies the no bypass property using one of our retroactive auditing tools (e.g., [70, 73]). At time t_1, there is no violation of the property, however, at time t_2, our retroactive auditing tool finds a violation. Second, the feedback manager collects logs from the network service of the cloud for the period of $t_1 - t_2$; as we are sure that the critical event that caused this violation happens within this period. Third, ProSAS filters out all events with the* `GET` *requests, because these events are interface generated to show lists of* *different resources on the interface. 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 property.*

4.4 Implementation

This section describes how we integrate ProSAS into OpenStack.

4.4.1 Architecture

Figure 4.10 shows a high-level architecture of ProSAS. ProSAS consists of four major components: dashboard & reporting engine, interceptor, verification engine and pre-computation engine. The dashboard & reporting engine provides an interface to ProSAS users. The main users of ProSAS are cloud tenants, who provide customized security properties and other ProSAS configurations (e.g., a list of critical events and watchlist attributes) through the dashboard. ProSAS maintains a repository to provide auditing reports to its users. The ProSAS interceptor is placed within the cloud as a middleware, in between the cloud dashboard or command line

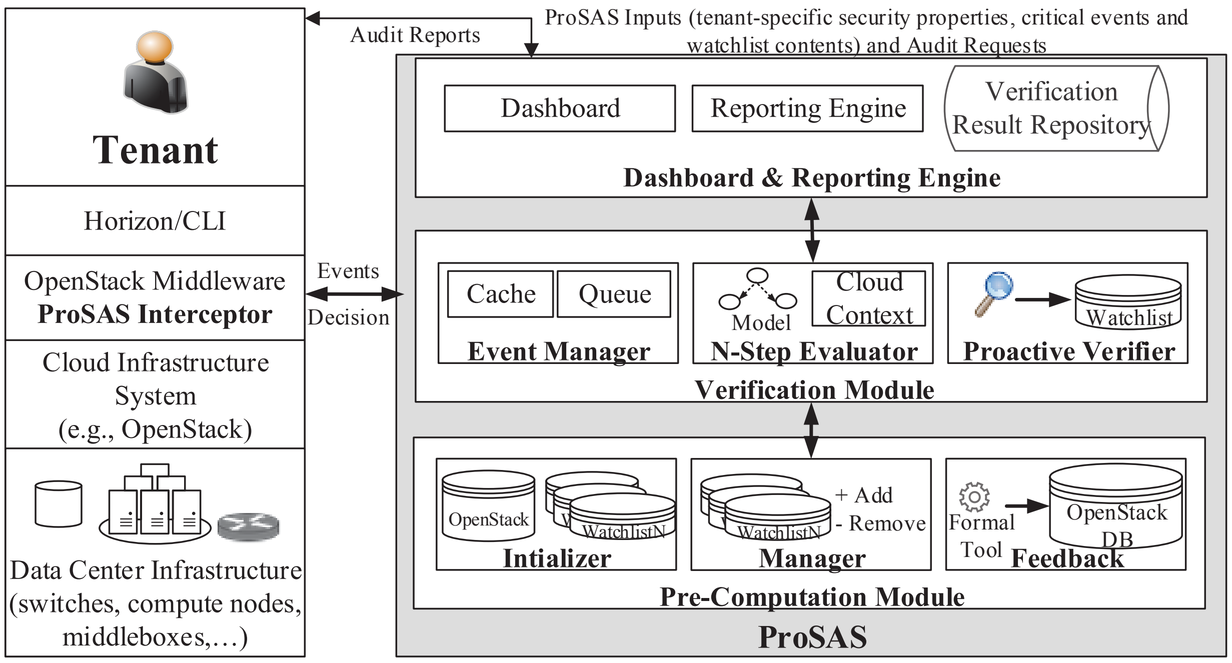


Figure 4.10: A high-level architecture of ProSAS

interface and different services (e.g., Nova, Neutron, Swift, etc. in OpenStack). This module intercepts all tenant initiated events and forwards them to ProSAS for a runtime verification, and enforces the verification results (e.g., allow or deny). In the verification engine, the caching manager contains caches and event queue, *N*-step evaluator measures *N* for each critical event from the intercepted event using cloud context, which is actually populated from the current cloud configurations (e.g., OpenStack database), and the proactive verifier queries watchlist databases to verify the parameters of the intercepted events. The pre-computation engine first initializes all watchlist databases for different databases storing the cloud configurations (e.g., OpenStack databases), then incrementally updates those watchlists based on the parameters of intercepted events, and also progressively learns new critical events by executing a formal verification tool (e.g., Sugar [106]) on cloud configurations (e.g., OpenStack databases).

4.4.2 Integration into OpenStack

Background. OpenStack [89] is an open-source cloud infrastructure management platform that is being used almost in half of private clouds and significant portions of the public clouds (see [21] for detailed statistics). Keystone [89] is the OpenStack identity service for authentication and authorization. Keystone implements the RBAC model [100]. Neutron [89] provides tenants with capabilities to build networking topologies through the exposed APIs. Nova [89] is the OpenStack project designed to provide on-demand access to compute resources, and relies on VMs.

Interceptor Middleware. The interceptor module, which is implemented in Python, intercepts operations based on the existing intercepting methods (e.g., audit middleware [90]) supported in OpenStack. We intercept event instances requested to the Nova service as they are passed through 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 [91] corresponding to the events that are relevant to the monitored security properties. Finally, the interceptor determines the criticality of the current event, and forwards the intercepted event details (e.g., type and parameters) to the caching manager.

Caching Manager. The caching manager, which is mainly implemented in Python, consults any of the pre-computation and verification caches. 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 form of key value pairs in which data is stored in value, and key is the hash value. If it is a hit in the cache, then we either perform the verification based on the cache entry or at least obtain information about the pre-computation to skip the *N*-step evaluator, and directly conduct the pre-computation (if necessary). To handle the concurrent events, we leverage the Python library *EventQueue*[1] so that events are handled sequentially (in case). Algorithm 2 shows the steps of the caching 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 cache-type is “verification” 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, Properties)
    else if cache-type is “precompute” then
        if event.type in cache & N-cp > N-th then
            return
        else if event.type in cache & N-cp <= N-th then
            perform actions mentioned in cache
        else
            Pre-Compute-Update(WL, Properties, event.params)
```

Proactive Verification Engine. Our pre-computation is mainly implemented in Python, and our pre-computed results and tenant-specific watchlists are in a MySQL database, which allows us to efficiently query OpenStack cloud data. The initializer module first populates all

[1]https://m7i.org/tutorials/python-event-queue-concurrency-modeling/

watchlist tables from Neutron, Nova and Keystone databases; this step allows to capture the initial configurations into the watchlists. Our Python scripts derive the association between the model provided in Fig. 4.3 and the security properties, and populate our database by adding the dependency information and the values of the pre-computed N. After one-time initialization, ProSAS incrementally updates these watchlists through the pre-computation manager, which is triggered by the runtime modules, such as N-step evaluator and inserts/deletes watchlist contents based on the parameters of the intercepted event. Algorithm 2 further explains the steps during the pre-computation phase.

The N-step evaluator is implemented as MySQL stored procedures to measure the distances from each critical event. Based on the outcome of both the caching manager and N-step evaluator modules, any of the following is performed: i) the pre-computation manager is triggered to update the watchlists, or ii) the proactive verifier searches the current values of the parameter(s) in the corresponding watchlist, and accordingly takes a decision (e.g., allow or deny) through the interceptor. Algorithm 3 further details the steps of the verification engine.

Feedback Engine. The feedback manager periodically invokes the formal verification tool (e.g., [70, 73]) to verify OpenStack configurations for the requested security properties. Whenever, the verification tool finds any violation, the feedback manager collects event logs from the corresponding OpenStack service (e.g., Neutron, Nova and Swift). Finally, the feedback 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. Algorithm 4 shows the steps of the feedback engine.

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, CCM V3.0.1, NIST 800-53, etc.). Afterwards, security properties under the selected standards can be chosen. Additionally, admins can select any of the following verification options: i) proactive verification ii) runtime verification, or iii) retroactive verification. Apart from the proactive enforcement of compliance through the interceptor, the reporting engine of ProSAS provides a detailed report on recent

Algorithm 3 Proactive Verification

```
1: procedure INITIALIZE($DependencyModels$, $SecPropertyNth$, $CloudOS$)
2:     Events=GetEvents($DependencyModels$)
3:     Mapping=ReadEventOperationMapping($Events$, $CloudOS$)
4:     for each event $e \in$ Events do
5:         Populate Event-operation
6:     for each property $p \in$ SecPropertyNth do
7:         Populate Property-N-thresholds
8:     for each property $p \in$ SecPropertyNth do
9:         criticalRelation = getCriticalEdge(p, DependencyModels)
10:        Allpaths = ComputeN-allpaths(DependencyModels, criticalRelation)
11:        for each $path$ in Allpaths.paths do
12:            Populate Model-N-property
13:        Edges = getEdges(DependencyModels)
14:        for each $ed \in$ Edges do
15:            (event, model, type) =ReadAtributes($ed$, DependencyModels)
16:            Populate Model-event from DependencyModels
```

```
17: procedure PRE-COMPUTE-INITIALIZE($CloudOS$, $Property\text{-}WL$)
18:     for each property $p_i \in Properties$ do
19:         $WL_i$= initializeWatchlist($p_i$, $Property\text{-}WL$, $CloudOS$)
20: procedure PRE-COMPUTE-UPDATE($WL$, $property$, $parameters$)
21:     updateWatchlist($WL$, $property$, $parameters$)
```

```
22: procedure EVALUATENSTEP(Event e, Property p, Params opparams)
23:     Find N-th for $p$ from Property-N-thresholds
24:     Find entities in the model related to $p$
25:     context = CollectCloudData(entities)
26:     Find N-cp for p and context from Model-N-property
27:     if N-cp=N-th then
28:         updateWatchlist(p, opparams)
29:     else if N-cp $<$ N-th and e.type=WE then
30:         updateWatchlist(p, opparams)
```

```
31: procedure PROACTIVEVERIFY($Event$, $Properties$)
32:     for each property $p_i \in Properties$ do
33:         if $Event.parameters$ in $p_i.watchlist$ then
34:             Allow $Event$ in the cloud
35:         else
36:             Deny $Event$ in the cloud
```

Algorithm 4 Feedback Engine

```
procedure FEEDBACK($Properties$, $interval$)
    collectData($CloudOS$)
    while $true$ do
        for each property $p_i \in Properties$ do
            $results$= verifyOffline($CloudOS$, $p_i$)
            if $results$= "Violated" then
                $deltaLog$= collectLogs($CloudOS$, ($currentTime$-$interval$), $p_i$)
                $Feedback_i$= filterLogs($deltaLog$)
                consultExpert($Feedback_i$)
        Wait($interval$)
```

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. Figures 4.11 and 4.12 show screenshots of the ProSAS dashboard.

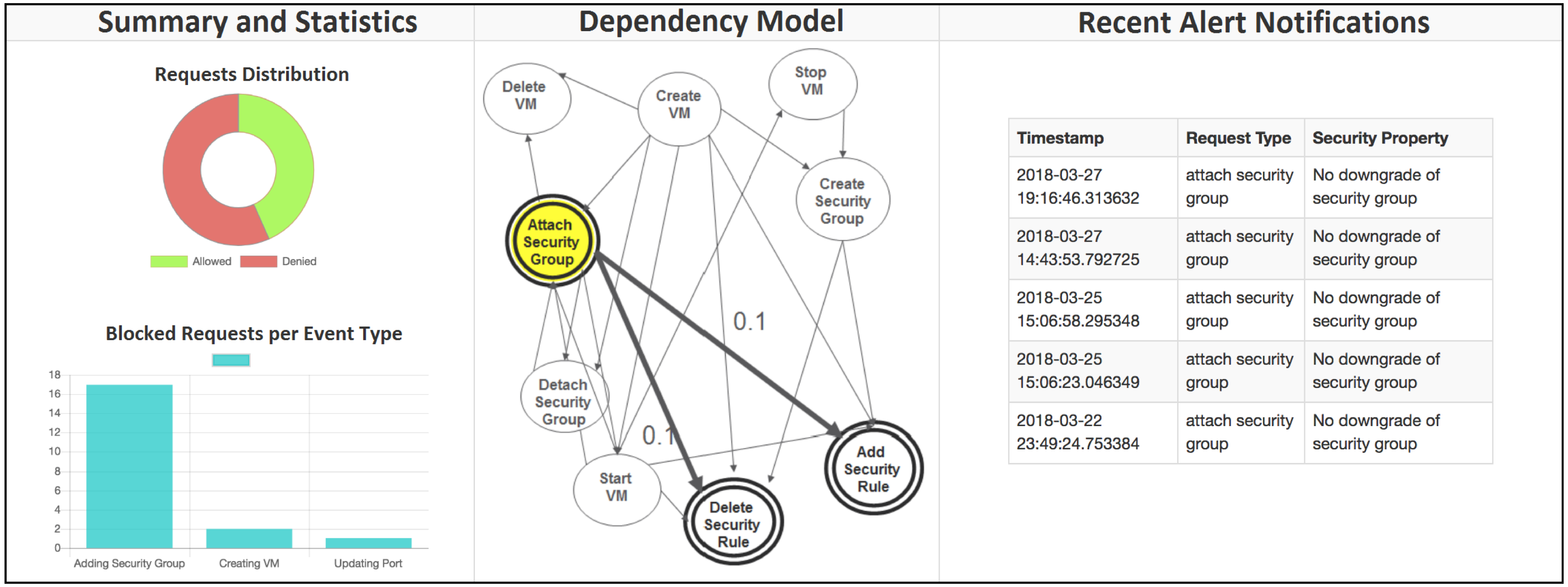
4.5 Experimental Results

In this section, we first describe the experiment settings, and then present ProSAS experimental results with both synthetic and real data.

4.5.1 Experiment Settings

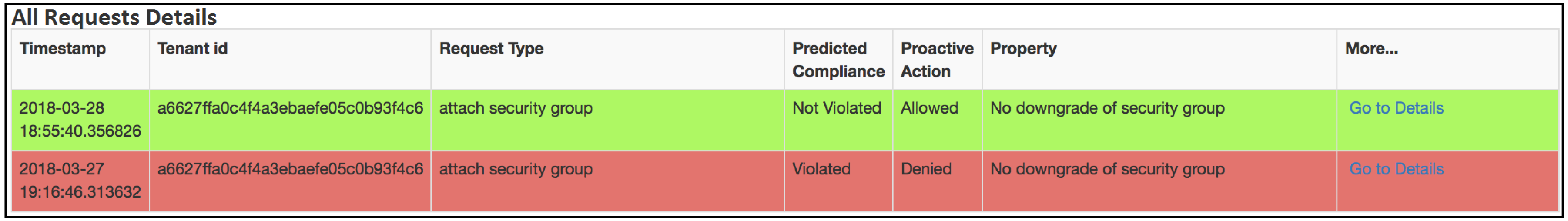
Our conducted experiments on ProSAS involve datasets collected from both our testbed and the real cloud. In the following, we describe both environmental settings.

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 running Ubuntu 16.04 server. Based on a recent survey [92] 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* property, both the number of ports (from 10,000 to 100,000 with the gap of 10,000) and the number of tenants (from 1,000



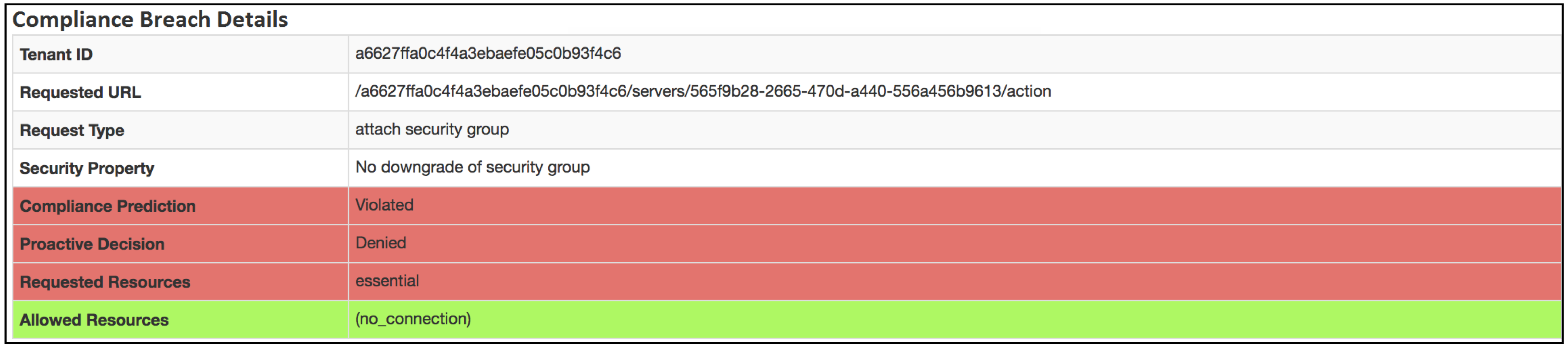
Timestamp	Request Type	Security Property
2018-03-27 19:16:46.313632	attach security group	No downgrade of security group
2018-03-27 14:43:53.792725	attach security group	No downgrade of security group
2018-03-25 15:06:58.295348	attach security group	No downgrade of security group
2018-03-25 15:06:23.046349	attach security group	No downgrade of security group
2018-03-22 23:49:24.753384	attach security group	No downgrade of security group

Figure 4.11: Screenshots of the ProSAS monitoring dashboard.



All Requests Details

Timestamp	Tenant id	Request Type	Predicted Compliance	Proactive Action	Property	More...
2018-03-28 18:55:40.356826	a6627ffa0c4f4a3ebaefe05c0b93f4c6	attach security group	Not Violated	Allowed	No downgrade of security group	Go to Details
2018-03-27 19:16:46.313632	a6627ffa0c4f4a3ebaefe05c0b93f4c6	attach security group	Violated	Denied	No downgrade of security group	Go to Details



Compliance Breach Details

Tenant ID	a6627ffa0c4f4a3ebaefe05c0b93f4c6
Requested URL	/a6627ffa0c4f4a3ebaefe05c0b93f4c6/servers/565f9b28-2665-470d-a440-556a456b9613/action
Request Type	attach security group
Security Property	No downgrade of security group
Compliance Prediction	Violated
Proactive Decision	Denied
Requested Resources	essential
Allowed Resources	(no_connection)

Figure 4.12: Screenshots of the ProSAS audit report dashboard.

to 10,000 with the gap of 1,000) are varied, as the watchlist related to our example security property contains a list of ports belonging to different tenants. For the *common ownership*[1] property, the number of tenants is varied from 1,000 to 10,000 with the gap of 1,000 having 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 is not configured in this cloud, we utilize Nova and Neutron logs that increases the log processing efforts.

4.5.2 Experimental Results with Testbed Clouds

The objective of the first set of experiments is to measure the effect of the ProSAS caching system. Figures 4.13, 4.14 and 4.15 show the hit ratio (i.e., $\frac{\text{number of hits}}{\text{total number of tries}}$) and the effects of our caching system applying two different caching mechanisms, i.e., least recent update (LRU) and most recent update (MRU), by varying the size of the cache. Figure 4.13 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. Figure 4.14 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. Figure 4.15 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 Figure 4.14, 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

[1]This property allows users to hold only the roles that are defined within their domains [53, 18].

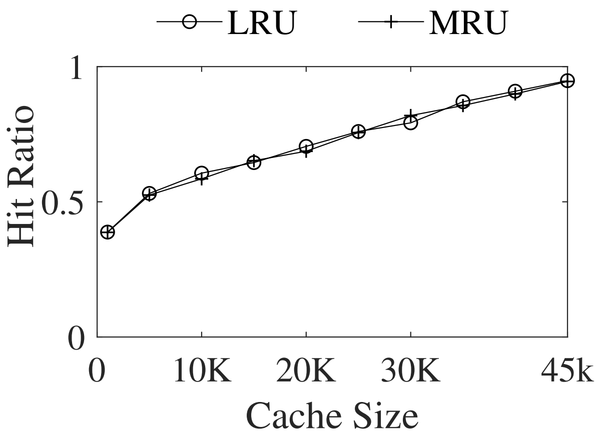


Figure 4.13: Evaluation of caching: hit ratio ($\frac{\text{number of hits}}{\text{total number of tries}}$) for both least recently used (LRU) and most recently used (MRU) caches. In all cases, we vary cache size (in number of entries) from 1,000 to 45,000, and verify the no bypass security property.

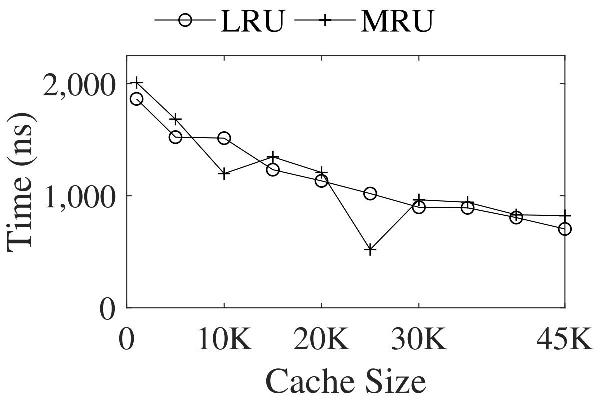


Figure 4.14: Evaluation of caching: average response time (in nanoseconds) by ProSAS when the intercepted event is found in the cache for both LRU and MRU caches. In all cases, we vary cache size (in number of entries) from 1,000 to 45,000, and verify the no bypass security property.

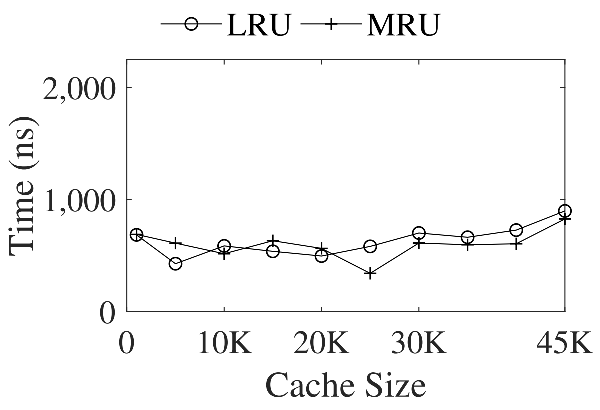


Figure 4.15: Evaluation of caching: delay (in nanoseconds) caused by a miss for both LRU and MRU caches. In all cases, we vary cache size (in number of entries) from 1,000 to 45,000, and verify the no bypass security property.

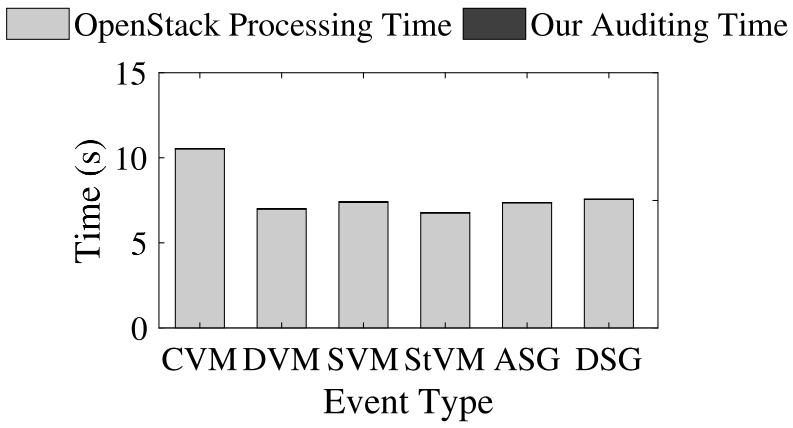


Figure 4.16: Time (in seconds) required to process different requests by OpenStack and ProSAS.

individually by OpenStack and ProSAS. Figure 4.16 shows the time (in seconds) to process different event 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, Figure 4.16 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.

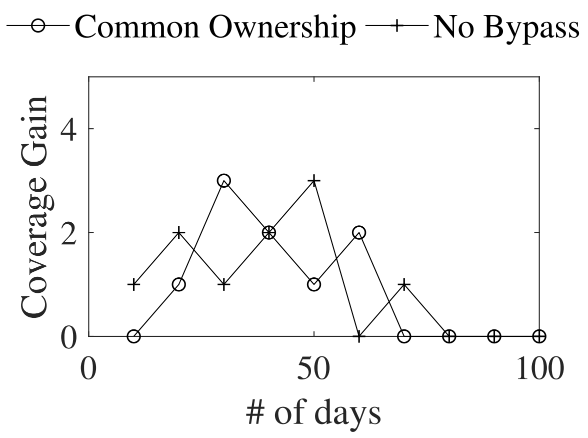


Figure 4.17: The coverage gain (measured in terms of number of new violations detected) by ProSAS feedback module over 100 days with the 10,000 tenants for the common ownership and no bypass security properties.

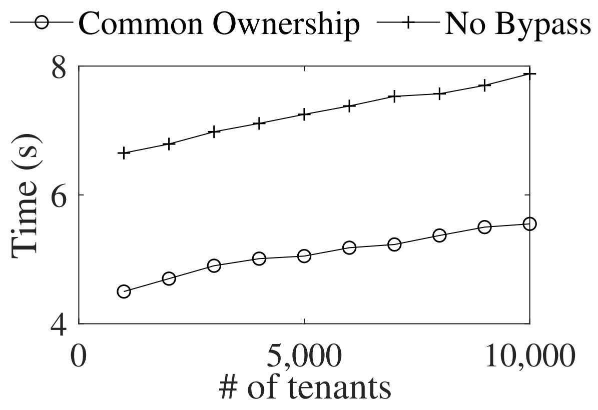


Figure 4.18: Time (in seconds) required to prepare feedbacks to improve the list of critical events while varying the number of events for the common ownership and no bypass security properties.

The objective of the third set of the experiments is to measure the coverage gain and time requirement of our feedback manager. Figure 4.17 shows the number of new violations ProSAS catches over 100 days after introducing the feedback manager in it for both common ownership and no bypass security properties. During the first two months, we observe the highest gain. In the last 20 days, there is no new security violation. Figure 4.18 measures the time (in seconds) to prepare the feedback for the common ownership and no bypass security properties while varying the number of tenants up to 10,000. Note that the reported time only includes the time to perform automatic steps (e.g., executing verification tool and filtering logs). The feedback preparation for the common ownership and no bypass properties takes maximum 5.55 seconds and 7.88 seconds, respectively, for our largest dataset.

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. 4.19). Moreover, calculating N-step (NSE in Fig. 4.19) completes in constant time (i.e., 0.133 ms) for the *no bypass* (NB) property, and in quasi constant time (varying from 0.773 ms to 0.794 ms) for the *common ownership* (CO) property. The violation detector blocks only critical operations for a maximum extra delay of 8.2 ms (VD in Fig. 4.20) for the largest dataset. Fig. 4.21 shows that pre-computing the watchlists for both *no bypass* and *common ownership* properties 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. 4.21 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 eight milliseconds per critical operation call at runtime.

In the fifth part of the experiments, we measure the memory cost for the watchlists. Fig. 4.22 depicts that the memory requirement increases quasilinearly 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 property, and only port IDs for the no bypass property.

Finally, Tables 4.6 and 4.7 compare the execution time of ProSAS and our alternative implementation of *intercept-and-check*, in which after detecting a critical event we collect data from the cloud and start verifying security properties using a SAT solver (e.g., Sugar [106]). We observe that verifying with the *intercept-and-check* approach including data collection takes 15

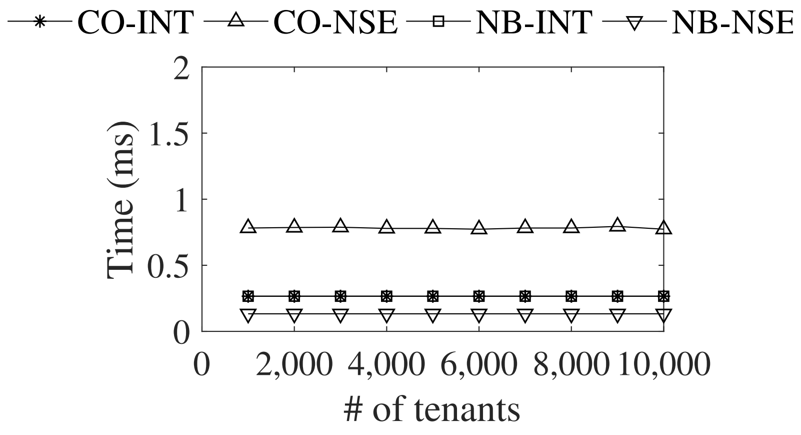


Figure 4.19: Time duration (in ms) for different modules (INT: Interceptor and NSE: N-step evaluator) of ProSAS for the *common ownership* (CO) and *no bypass* (NB) security properties by varying the number of tenants. The number of ports is also varied from 10,000 to 100,000, and each tenant contains five roles. Time required for the steps: intercepting operations and evaluating N-step.

Number of Ports	**10,000**	**20,000**	**30,000**	**40,000**	**50,000**
Intercept-and-check	60,200	107,209	184,230	237,245	317,252
ProSAS	5.928	6.09	6.916	7.016	7.496

Table 4.6: Comparing execution time (in ms) between ProSAS and our alternative implementation of *intercept-and-check* for the common ownership and no bypass security properties for the first set of dataset.

seconds (for common ownership) to 8 minutes (for no bypass) for our largest dataset. Therefore, each critical operation would experience long response time. In contrast, LeaPS experiences maximum response time of 8.5 ms. Our solution only permits allowed actions, hence any further accuracy evaluation is irrelevant.

Number of Ports	**60,000**	**70,000**	**80,000**	**90,000**	**100,000**
Intercept-and-check	357,261	407,268	437,271	455,276	480,277
ProSAS	7.815	8.024	8.14	8.453	8.501

Table 4.7: Comparing execution time (in ms) between ProSAS and our alternative implementation of *intercept-and-check* for the common ownership and no bypass security properties for the second set of dataset.

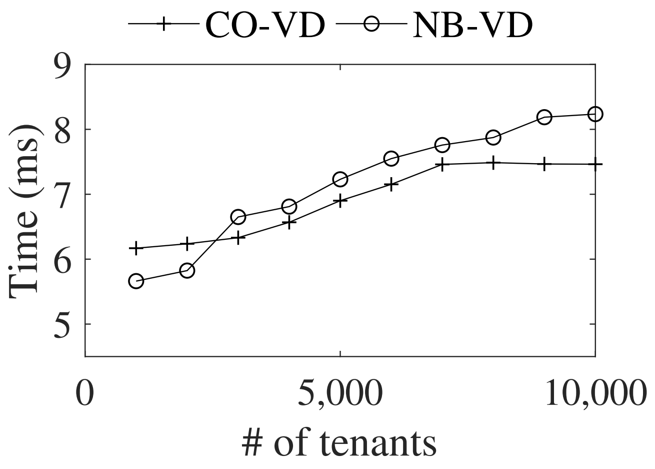


Figure 4.20: Time duration (in ms) for the Violation detector (VD) of ProSAS for the *common ownership* (CO) and *no bypass* (NB) security properties by varying the number of tenants. The number of ports is also varied from 10,000 to 100,000, and each tenant contains five roles. Time required for detecting violations.

Properties	**Hit Ratio**	**Pre-Compute**	**Feedback**	**Verification (W)**	**Verification (C)**	**Delay**
No bypass	0.71	2500ms	5270ms	6.2ms	1250ns	890ns
Common ownership	0.721	1700ms	3150ms	5.5ms	1000ns	810ns

Table 4.8: Summary of the experimental results with real data. The reported delay is the additional time required in LeaPS 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.

4.5.3 Experimental Results with Real Clouds

Table 4.8 summarizes the obtained results for the real cloud dataset, which logs of total 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.

4.6 Discussions

In this section, we discuss different aspects of ProSAS.

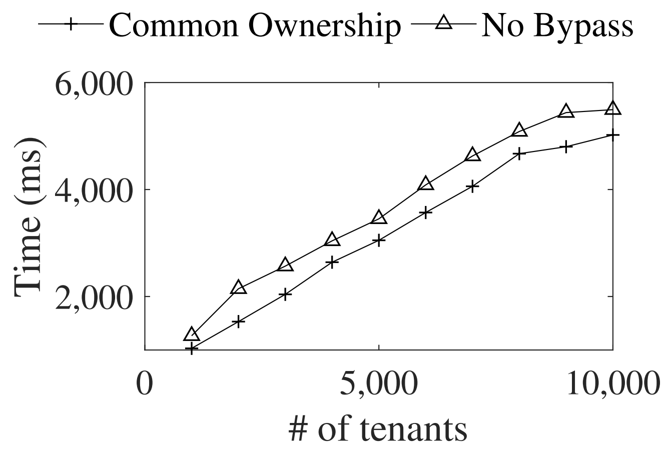


Figure 4.21: (a) Time required (in ms) for preparing watchlist for different properties varying the number of tenants at the initialization step. The number of ports is also varied from 10,000 to 100,000 , and each tenant contains five roles.

Effects of Change in Cloud Design. As our experiment results shown in Section 4.5, ProSAS can verify security properties for large size cloud in only few seconds 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 extending verification to new security properties, the ProSAS initialization must be repeated. Even though the initialization can take several minutes, this task can be executed in parallel with run time verification and the pre-computed information updated instantly to minimize the impact on verifications at runtime. Note that there are few cases where the pre-computation needs to be repeated and those cases regarding management API changes in the cloud are by nature not frequent.

Supporting Operational Properties. In this work, we cover structural properties involving cloud management operations (e.g., creating a tenant, granting a role, assigning instances to physical hosts and configuring virtualization mechanisms). The properties involving session/-context specific data are not yet considered. In our running example, if the malicious tenant can somehow successfully bypass the firewall rules and launch a spoofing attack, our solution cannot yet detect such spoofing attacks. As our solution relies on the information reported through the management interface, any verification by extracting the information from the actual infrastructure components (e.g., virtual or hardware) is not covered in this work and considered as a

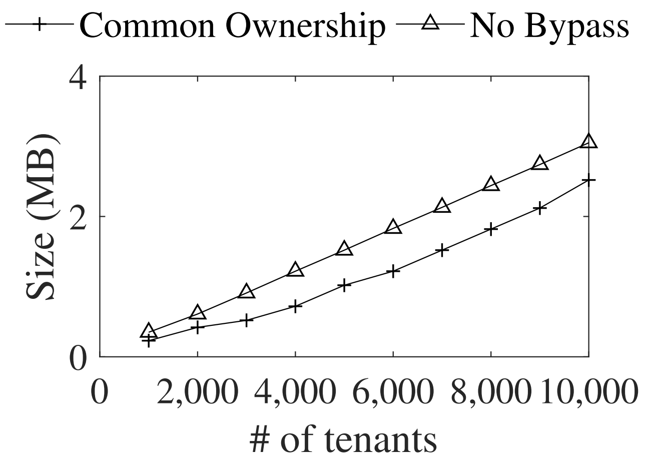


Figure 4.22: Time required (in ms) for preparing watchlist for different properties varying the number of tenants at the initialization step. The number of ports is also varied from 10,000 to 100,000 , and each tenant contains five roles.

Cloud Platform	**Interception Support**
OpenStack	*WSGI Middleware* [113]
Amazon EC2-VPC	*AWS Lambda Function* [5]
Google GCP	*GCP Metrics* [38]
Microsoft Azure	*Azure Event Grid* [77]

Table 4.9: Interception supports in major cloud platforms

potential future work.

Adapting to Other Cloud Platforms. ProSAS is designed to work with most popular cloud platforms (e.g., OpenStack [89], Amazon EC2 [5], Google GCP [38] and Microsoft Azure [77]) with a one-time effort for implementing a platform-specific interface. More specifically, ProSAS interacts with the cloud platform (e.g., while collecting logs and intercepting runtime events) through two modules: log processor and interceptor. These two modules require to interpret implementation specific event instances and intercept runtime events. First, to interpret platform-specific event instances to generic event types, we currently maintain a mapping of the APIs from different platforms. Table 4.10 enlists some examples of such mappings. Second, the interception mechanism may require to be implemented for each cloud platform. In OpenStack, we leverage 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

ProSAS Event Type	OpenStack [89]	Amazon EC2-VPC [5]	Google GCP [38]	Microsoft Azure [77]
create VM	`POST /servers`	`aws opsworks -region create-instance`	`gcloud compute instances create`	`az vm create l`
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 4.10: Mapping event APIs from different cloud platforms to ProSAS event types.

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 ProSAS 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 is quite similar as our interception mechanism. Therefore, we believe that ProSAS can be an extension of the Azure Event Grid to proactively audit cloud events. Table 4.9 summarizes the interception support in these cloud platforms. The rest modules of ProSAS deal with the platform-independent data, and hence, the next steps in ProSAS are platform-agnostic.

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 properties 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 4.20, and may be detected by performing all 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 4.21);

which is still faster than any other existing works (which respond 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 internal tasks 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 elaborate those two ways of tackling single-step violations in our future work.

Choosing the Value of $N-th$. As described in Section 5.5, ProSAS schedules the pre-computaiton based on a threshold value ($N-th$), and the distance ($N-cp$) from the current intercepted event to a critical event. The ProSAS users (e.g., tenants) choose the value of $N-th$. The possible values of $N-th$ are in the range of $\{N_{max}:1\}$, where N_{max} is the longest possible distance between two events in the dependency model. Within this range, choosing a larger value may allow ProSAS more time to perform the pre-computation. However, at the same time it might not be much effective, as it requires more watchlist updates over the time. On the other hand, choosing a smaller $N-th$ may allow a precise watchlist content. However, in that case, ProSAS may not get enough time to perform necessary pre-computation steps before the critical event occurs.

4.7 Related Work

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

4.7.1 Comparison between Related Works

Table 5.13 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

Proposals	Methods	Coverage		Features						Supporting Platforms			
		Environment	Cloud Layer	Proactive	Enforcive	Caching	Queuing	Expressive	Self-Reliant	OpenStack	Azure	VMware	Adaptable
Doelitzscher et al. [25]	Custom Algorithm	Cloud	Virtual Infr.	-	-	N/A	N/A	-	●	●	-	-	●
Ullah et al. [108]	Custom Algorithm	Cloud	Virtual Infr.	-	-	N/A	N/A	-	●	●	-	-	-
Majumdar et al. [73]	CSP Solver	Cloud	User-level	-	-	N/A	N/A	●	●	●	-	-	-
Madi et al. [70]	CSP Solver	Cloud	Virtual Infr.	-	-	N/A	N/A	●	●	●	-	-	-
Majumdar et al. [74]	CSP Solver	Cloud	User-level	-	●	-	-	●	●	●	-	-	-
Ligatti et al. [66]	Model Checking	Non-Cloud	N/A	●	●	-	-	●	●	N/A	N/A	N/A	N/A
PVSC [71]	Custom Algorithm	Cloud	Both	●	●	-	-	-	-	●	-	-	-
LeaPS [72]	Custom + Bayesian	Cloud	Both	●	●	-	-	-	-	●	-	-	●
Weatherman [16]	Graph-theoretic	Cloud	Virtual Infr.	●	-	N/A	N/A	-	-	-	-	●	-
Congress [88]	Datalog	Cloud	Both	●	-	N/A	N/A	●	-	●	-	-	-
Patron [69]	Custom Algorithm	Cloud	User-level	-	●	●	-	●	●	●	-	-	-
ProSAS	Custom Algorithm	Cloud	Both	●	●	●	●	-	○	●	-	-	●

Table 4.11: Comparing existing solutions with LeaPS. The symbols (●), (○)[a], (-) and N/A mean fully supported, partially supported, not supported and not applicable, respectively.

[a]This symbol is used when a work supports a feature partially, but less then other works supporting the same feature fully.

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 to the cloud at runtime, we check the enforcive feature. The caching feature is checked, when a work optimizes its verification computation by storing previous results. The queuing feature refers to handling concurrent events for an enforcive solution. We mark both caching and queuing features as ‘N/A’ for the works that do not support the enforcive feature. The expressive feature is checked for the works, which utilize well-known expressive policy languages (e.g., first order logic) to express security properties. By the self-reliant feature, we mean the works which only depend on the user-provided security properties for the accuracy of the verification. In the last four columns of the table, we compare the works based on their supporting cloud platforms. The adaptable field is checked for those works which support multiple cloud platforms or describe how their works 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 captures dependencies among cloud events. Secondly, ProSAS is the only proactive auditing solution, which supports caching of recent verification computations and results to more efficiently (e.g., in nanoseconds) audit clouds. Thirdly, unlike other proactive solutions, ProSAS can handle concurrent events by maintaining an event queue. Fourthly, even though ProSAS is not as self reliant as most retroactive approaches (e.g., [25, 108, 73]), ProSAS improves the current state-of-the-art for proactive solutions by adding a feedback loop to progressively reduce the reliance on a list of critical events. 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 relies 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.

4.7.2 Cloud Security Auditing

The existing solutions in cloud security auditing can be categorized into three major approaches: retroactive, intercept-and-check, and proactive. We discuss related works under these categories as follows.

Retroactive Auditing Approach. Auditing security compliance in the cloud has recently been explored. For instance, Solonas et al. [104] detect illegal activities in the cloud only based on collected billing data in order to preserve privacy. In [73, 70], 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 [61] and NetPlumber [59] monitor network events and check network properties and policies at runtime to capture bugs before or as soon as they occur. They rely on incremental calculations to achieve the runtime verification. These works focus on operational network properties (e.g., black holes and forwarding loops) in traditional networks, whereas our effort is oriented toward preserving compliance with structural security properties that impact isolation in cloud virtualized infrastructures.

Various mechanisms and concepts for designing security service-level-agreement-based cloud monitoring services have been discussed in [96]. CloudSec [50] and CloudMonatt [116] propose VM security monitoring. In contrast, our work covers a larger spectrum of properties (beyond the scope of VMs) that require collecting data from various sources. In addition, unlike intercepting security measurements, we intercept multiple kinds of events and assess their impact on the cloud system before applying them. In [93], a host-based secure active monitoring mechanism, where protected hooks into untrusted VMs are installed to intercept malicious events, is proposed. Once a malicious action is intercepted, the control is transferred to security tools running on a trusted VM. They detect unwanted operations initiated by malicious softwares; whereas, our contribution is at a higher level covering events initiated by potentially untrusted users.

Intercept-and-Check Approach. Existing intercept-and-check approaches (e.g., [16, 88]) 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., [61, 59]) monitoring network events and checking network policies at runtime. Weatherman [16] and OpenStack Congress [88] offer security verification of virtual infrastructure using the intercept-and-check approach. These works focus on operational network properties (e.g., black holes and forwarding loops) in traditional networks, whereas our effort is oriented toward preserving compliance with structural security properties that impact isolation in cloud virtualized infrastructures. Cloud monitoring services based on security service-level agreements are discussed in [96].

Proactive Auditing Approach. Weatherman [16] is the most closely related work to ours. Aiming at mitigating misconfigurations and enforcing security policies in a virtualized infrastructure, Weatherman has both online and offline approaches. Their online approach intercepts management operations for analysis, and relays them to the management hosts, only if Weatherman confirms no security violation caused by those operations. Otherwise, they are rejected with an error signal to the requester. The work defines a realization model, that captures the virtualized infrastructure configuration and topology in a graph-based model. The latter is synchronized with the actual infrastructure using the approach in [14]. Two major limitations of this proposition are: i) the model capturing the whole infrastructure causes a scalability issue for the solution, and ii) the time consuming operation-checking that should be performed on the emergence of each event, makes security enforcement not feasible for large size data centers. Our work overcomes these limitations by proposing a proactive auditing approach by leveraging the dependency relationships among cloud events.

Congress [88] is an OpenStack project offering both online and offline policy enforcement approaches. The offline approach requires submitting a future change plan to Congress, so that the changes can be simulated and the impacts of those changes can be verified against specific properties. In the online approach, Congress first applies the operation to the cloud, then checks its impacts. In case of a violation, the operation is reverted. However, the time elapsed before reverting the operation can be critical to perform some illicit actions, for instance, transferring sensitive files before loosing the assigned role. Foley et al. [31] provide an algebra to assess the effect of security policies replacement and composition in OpenStack. Their solution can be considered as a proactive approach for checking operational properties violations, whereas our work targets the runtime verification of structural security property violations.

4.7.3 Other Proactive Security Approaches

Proactive security analysis has been explored for software security enforcement through monitoring programs' behaviors and taking specific actions (e.g., warning) in case security policies are violated. Many state-based formal models are proposed for those program monitors over the last two decades. First, Schneider [103] models program monitors using an infinite-state-automata model to enforce safety properties. Those automata recognize invalid behaviors and halt the target application before the violation occurs. Ligatti et al. [65] build on Schneider's model and defines a more general program monitors model based on the so called edit/security automata. Rather than just recognizing executions, edit automata-based monitors are able to suppress bad and/or insert new actions, transforming hence invalid executions into valid ones. Mandatory Result Automata (MRA) is another model proposed by Ligatti et al. [66, 26] that can transform both actions and results. Narain [80] proactively generates correct network configurations using the model finder Alloy. Our work further expands the proactive monitoring approach into cloud environments differing in scope and approach.

4.8 Conclusion

The continuous auditing with scalability and practical response time is important to both cloud providers and their tenants. In this work, 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 built a dependency model of the relationships between cloud events so that the auditing process can be launched proactively by leveraging this model. Second, we intercepted each runtime event instances and built the watchlists for each security property incrementally. Third, when a critical event occurs, we performed a light-weight verification by simply checking the intercepted event parameters with the watchlists to decide whether to allow or deny the critical event. Fourth, ProSAS stored the recent results in the cache from the verification and pre-computation steps to further improve the response time on average. Finally, we prepared a feedback to progressively improve the list of critical events, which directly impacts the accuracy of our solution, by using a state-based verification tool. We integrated ProSAS to OpenStack, one of the most popular cloud management platforms, and provided guidelines to port it to other cloud platforms. Furthermore, we evaluated the efficiency and accuracy of our method, and showed that the response time is reduced to a practical level (e.g., 1,041 nanoseconds and 8.2 milliseconds to audit 10,000 tenants with and without caching, respectively), and the accuracy is improved (19 new violations detected).

However, there exist several limitations in ProSAS, which we consider as future works. First, the current method of improving the critical events involves manual inspection, which could be error-prone. We intend to automate this step by using machine learning techniques to build a comprehensive critical event list. Second, single-step violations are not yet efficiently handled in ProSAS. An efficient runtime approach might help to address this concern. Third, concurrent critical management operations may affect the performance of ProSAS. A parallel or distributed approach might reduce the effect of this situation.

Chapter 5

Learning Probabilistic Dependencies among Events for Proactive Security Auditing in Clouds

5.1 Introduction

Security threats such as isolation breach in multi-tenant clouds cause persistent fear among tenants while adopting clouds [98]. To this end, security auditing in clouds can possibly ensure the accountability and transparency of a cloud provider to its tenants. However, the traditional approach of auditing, a.k.a. *retroactive auditing*, becomes ineffective with the unique nature (e.g., dynamics and elasticity) of clouds, which means the configurations of a cloud is frequently changed and hence, invalidates the auditing results. To address this limitation and offer continuous auditing, the *intercept-and-check* approach verifies each cloud event at runtime. However, the sheer size of the cloud (e.g., 1,000 tenants and 100,000 users in a decent-sized cloud [92]), can usually render the *intercept-and-check* approach expensive and non-scalable (e.g., over four minutes for a mid-sized cloud [16]). Since the number of critical events (i.e., events that may potentially breach security properties) to verify usually grows with the number of security properties supported by an auditing system, auditing larger clouds could incur prohibitive costs.

To this end, the proactive approach (e.g., [71]) is a promising solution and specifically designed to ensure a practical response time. Such an approach prepares for critical events in advance based on the so-called dependency models that indicate which events lead to the critical events [117, 71]. However, a key limitation of existing proactive approaches (including our previous work [71]) is that their dependency models are typically established through manual efforts based on expert knowledge or user experiences, which can be error-prone and tedious especially for large clouds. Moreover, existing dependency models are typically static in nature in the sense that the captured dependencies do not reflect runtime patterns. A possible solution is to automatically learn probabilistic dependencies from the historical data (e.g., cloud logs). However, the log formats in current cloud platforms (especially, in OpenStack [89], which is one of the most popular cloud management platforms) are unstructured and not ready to be fed into different learning tools. Furthermore, due to the diverse formats of logs in different versions of the cloud platform, the log processing task becomes more difficult. Therefore, to enable log analysis (e.g., learning dependency models for proactive auditing), the need of a log processing approach addressing different real-world challenges (which are discussed in Section 5.3.2) and preparing raw logs for different learning tools is evident.

To address those limitations, our key idea is to design a log processor, which prepares the inputs for different learning techniques, to learn probabilistic (instead of deterministic) dependencies, and to automatically extract such a model from processed logs. Specifically, we first conduct case studies on cloud log formats in different OpenStack deployments including a real community cloud, and enumerate all challenges related to raw log processing to automate different learning mechanisms. Second, we design a log processor that addresses all challenges identified in our investigation, and provides inputs for different learning techniques (e.g., Bayesian network and sequence pattern mining). Third, we propose a new approach to automatically generate the probabilistic dependency models from the processed logs. Fourth, we provide detailed methodology and algorithms for our learning-based proactive security auditing system, namely, *LeaPS*, including the log processor, learning component and proactive verification component. We describe our implementation of the proposed system based on OpenStack [89], and demonstrate how the system may be ported to other cloud platforms (e.g., Amazon EC2 [5] and Google

GCP [38]). Finally, we evaluate our solution through extensive experiments with both synthetic and real data. The results confirm our solution can achieve practical response time (e.g., 6ms to audit a cloud of 100,000 VMs) and significant improvement over existing proactive approaches (e.g., about 50% faster), and our log processor can be adopted by different learning techniques efficiently (e.g., only 18ms to execute different sequence pattern mining algorithms for 50,000 events).

In summary, our main contributions are threefold.

- To the best of our knowledge, this is the first approach for processing OpenStack logs for identifying event sequences to learn dependencies. First, our study investigates cloud logs from both real and testbed clouds, and enumerates all challenges in log processing. Second, our log processing technique addresses these challenges, and supports different learning techniques (e.g., Bayesian network and sequence pattern mining).

- We propose an automated learning-based proactive auditing system, namely, *LeaPS*, which automatically learns probabilistic dependencies using the proposed log processor to allow handling the uncertainty that is inherent to runtime events, and hence, addresses the major limitations of existing proactive solutions. As demonstrated by our implementation and experimental results, LeaPS provides an automated, efficient, and scalable solution for different cloud platforms to increase their transparency and accountability to tenants.

- Unlike most learning-based security solutions, since we are not relying on learning techniques to detect abnormal behaviors, we avoid the well-known limitations of high false positive rates; any inaccuracy in the learning phase would only affect the efficiency, as will be demonstrated through experiments later in this chapter. We believe this idea of leveraging learning for efficiency, instead of security, may be adapted to benefit other security solutions.

5.2 LeaPS Overview

In this section, we present a motivating example, describe the threat model, and provide an overview of our proposed solution.

5.2.1 Motivating Example

The upper part of Figure 5.1 depicts several sequences of events in a cloud (from Session N to Session $N+M$). The critical events, which can potentially breach some security properties, are shown shaded (e.g., $E2$, $E5$ and $E7$). The lower part of the figure illustrates two different auditing approaches of such events. We discuss their limitations below to motivate our solution.

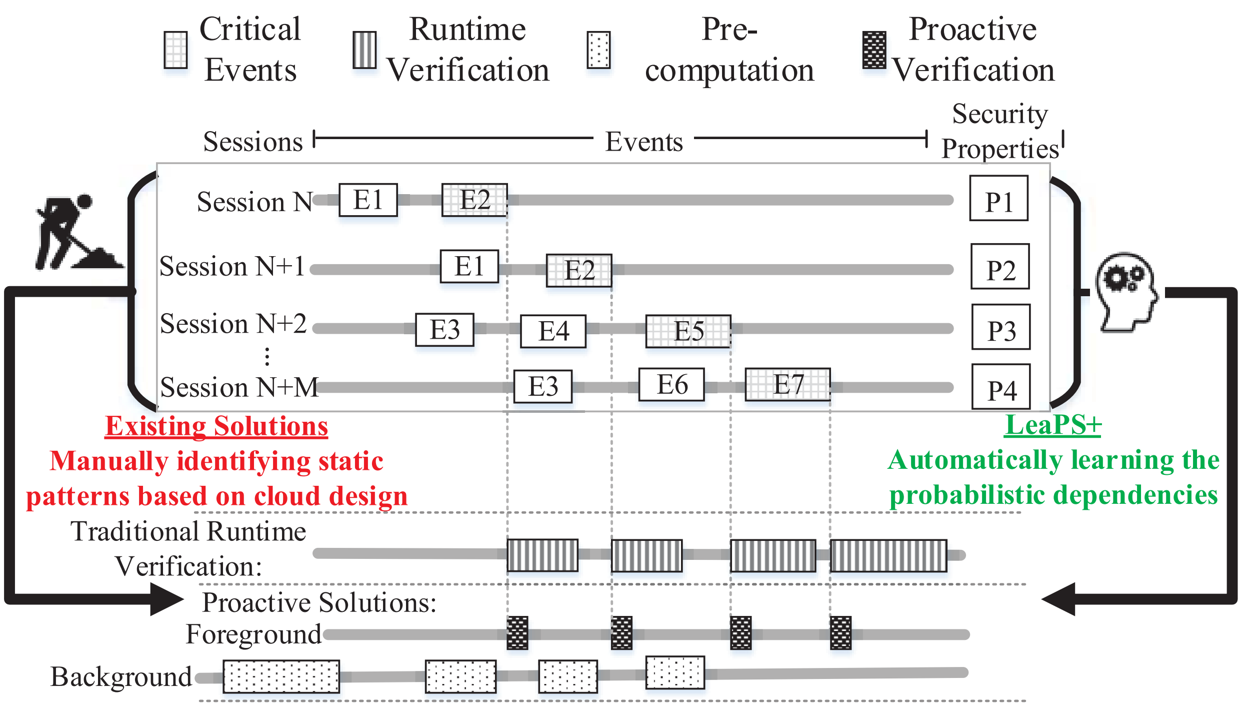


Figure 5.1: Identifying the main limitations of both traditional runtime verification and existing proactive solutions, and positioning our solution.

- With a traditional runtime verification approach, most of the verification effort (depicted as boxes filled with vertical lines) is performed after the occurrence of the critical events, while holding the related operations blocked until a decision is made; consequently, such solutions may cause significant delays to operations.

- In contrast, a proactive solution will pre-compute most of the expensive verification tasks well ahead of the critical events in order to minimize the response time. However, this means such a solution would need to first identify patterns of event dependencies, e.g., $E1$ may lead to a critical event ($E2$), such that it may pre-compute as soon as $E1$ happens.

- Manually identifying patterns of event dependencies for a large cloud is likely expensive and non-scalable. Indeed, a typical cloud platform allows more than 400 types of operations [89], which implies 160,000 potential dependency relationship pairs may need to be examined by human experts.

- Furthermore, this only covers the static dependency relationships implied by the cloud design, whereas runtime patterns, e.g., those caused by business routines and user habits, cannot be captured in this way.

- Another critical limitation is that existing dependency models are deterministic in the sense that every event can only lead to a unique subsequent event. Therefore, the case demonstrated in the last two sessions ($N+2$, $N+M$) where the same event ($E3$) may lead to several others ($E4$ or $E6$) will not be captured by such models.

5.2.2 Threat Model

We assume that the cloud infrastructure management systems i) may have implementation flaws, misconfigurations and vulnerabilities that can be potentially exploited to violate security properties specified by the cloud tenants, and ii) may be trusted for the integrity of the API calls, event notifications, logs and database records (existing techniques on trusted computing may be applied to establish a chain of trust from TPM chips embedded inside the cloud hardware, e.g., [10, 64]). Though our framework may assist to avoid any violation of specified security properties due to either misconfigurations or exploits of vulnerabilities, our focus is not to detect specific attacks or intrusions. We focus on attacks directed through the cloud management interfaces (e.g., CLI, GUI), and any violation bypassing such interfaces is beyond the scope of this work. We assume a comprehensive list of critical events are provided upon which the accuracy of our auditing solution depends (however, we provide a guideline on identifying critical events in Section 5.8). Our proactive solution mainly targets certain security properties which would require a sequence of operations. To make our discussions more concrete, the following shows an example of in-scope threats based on a real vulnerability.

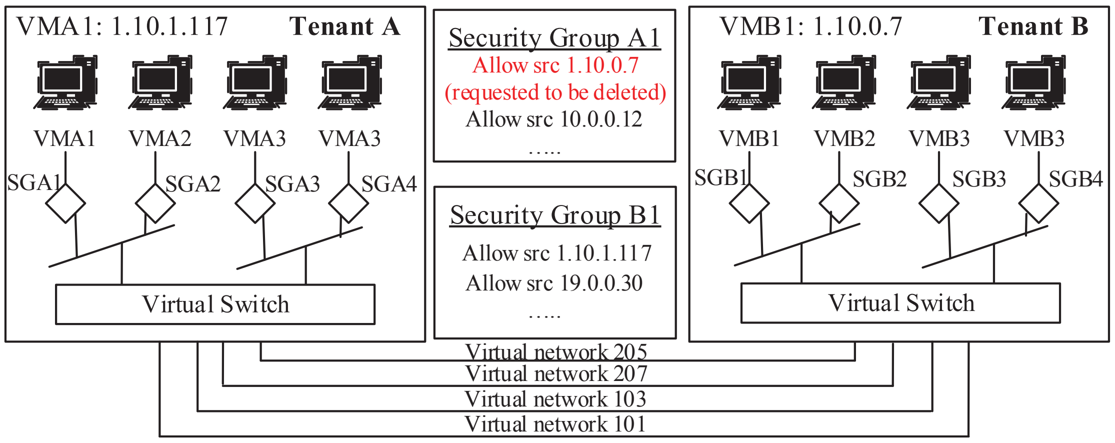


Figure 5.2: An exploit of a vulnerability in OpenStack [87], leading to bypassing the security group mechanism.

Running Example. A real world vulnerability in OpenStack[1], CVE-2015-7713 [87], can be exploited to bypass security group rules (which are fine-grained, distributed security mechanisms in several cloud platforms including Amazon EC2, Microsoft Azure and OpenStack to ensure isolation between instances). Figure 5.2 shows a potential deployment configuration which might be exploited using this vulnerability. The pre-requisite steps of this scenario are to create VMA1 and VMB1 (*step 1*), create security groups A1 and B1 with two rules (i.e., *allow 1.10.0.7* and *allow 1.10.1.117*) (*step 2*), and start those VMs (*step 3*). Next, when Tenant A tries to delete one of the security rules (e.g., *allow 1.10.0.7*) (*step 4*), the rule is not removed from the security group of the active VMA1 due to the afore-mentioned vulnerability. As a result, VMB1 is still able to reach VMA1 even though Tenant A intends to filter out that traffic. According to the vulnerability description, the security group bypass violation occurs only if this specific sequence of event instances (steps 1-4) happens in the mentioned order (namely, *event sequence*). In the next section, we present an overview of our approach and show how we automatically capture probabilistic dependencies among cloud events for proactive security

[1]OpenStack [89] is a popular open-source cloud infrastructure management platform.

auditing.

5.2.3 Approach Overview

In the following, we briefly describe our learning-based proactive auditing techniques used by LeaPS.

- First, it parses raw cloud logs into a structured format after marking each field of log entries so that log processing in the next step can be efficient.
- Next, it processes these parsed logs to interpret event types, aggregate log entries from different services (e.g., compute and network), and prepare inputs (as event sequences) for learning techniques.
- Then, it learns probabilistic dependencies between different event types captured as a Bayesian network from sequences of events processed from different cloud logs.
- Afterwards, based on the decreasing order of critical events' conditional probabilities in these dependency models, LeaPS incrementally prepares the ground for the runtime verification.
- Finally, once one of these critical events is about to occur, we simply verify the parameters associated with its event instance with respect to the pre-computed conditions of that event, and enforce the security policy according to the verification result.

Figure 5.3 shows an overview of LeaPS. It consists of three major modules: log processor, learning system and proactive verification system. The log processor is related to processing the unstructured and incomplete raw log files, which will be detailed in Section 5.3.1, and prepares the data to be used by the learning system. Our log processor consists of four major parts. The parser is responsible to identify fields for each log entries and parse them into a structured format. The filter extracts the relevant log entries and groups them based on tenant IDs. The interpreter is to mark event types for each log entry. Finally, the sequence identifier is responsible to extract the sequence out of those log entries and prepare inputs for various learning techniques. The learning system is dedicated for learning probabilistic dependencies for

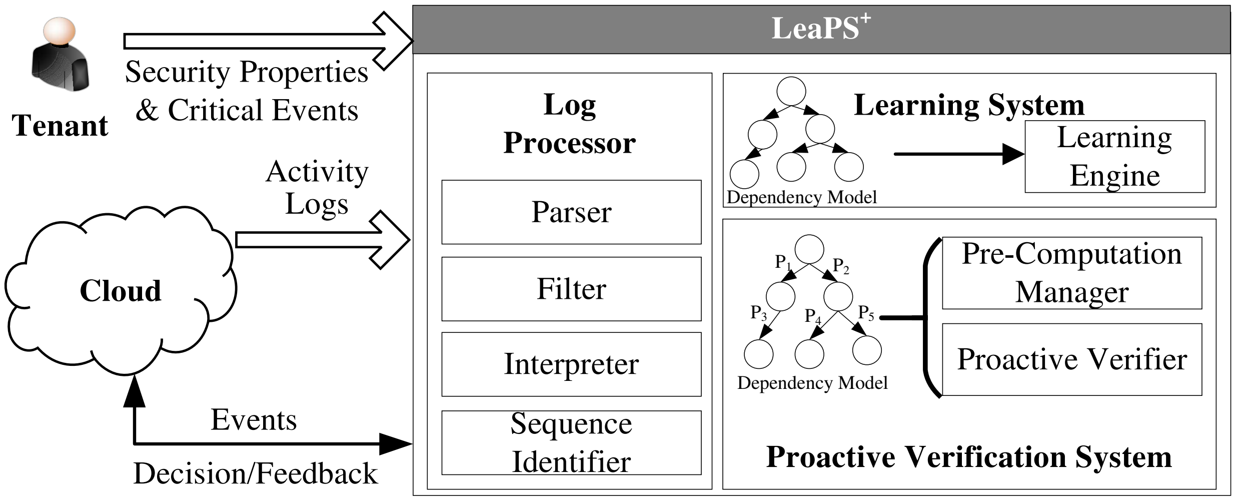


Figure 5.3: An overview of LeaPS log processing, learning and auditing mechanisms.

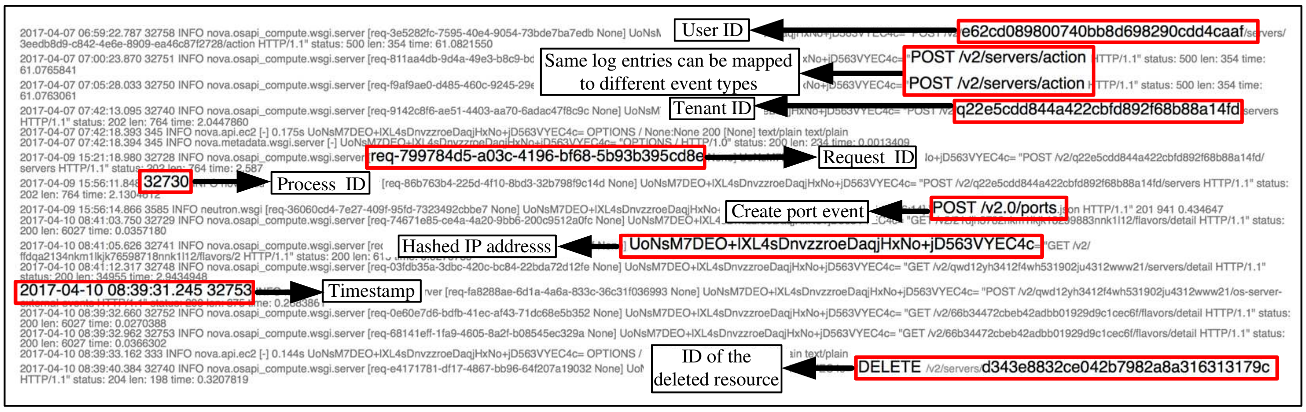
the model. The proactive verification system consists of two major parts. The pre-computation manager prepares the ground for the runtime verification. At runtime, a light-weight verification tool (e.g., proactive verifier [71]), which basically executes queries in the pre-computed results, is leveraged for the verification purpose. Based on the verification result, LeaPS provides a decision on the intercepted critical event instance.

5.3 Case Studies and Log Processing

In this section, we detail our approach for processing unstructured raw cloud logs and present the challenges and lessons learned from the analysis of the formats of real world cloud logs in OpenStack [89].

5.3.1 Case Studies on Real-World Cloud Logs

As a first step, we conducted an investigation on how the executed operations at the cloud management level are logged in OpenStack [89], one of the major cloud management systems in today's cloud environments. To this end, we used two different OpenStack deployments; a real-life community cloud hosted at a real data center of a large telecommunication vendor and a cloud testbed managed within our institution. All sensitive information in the logs is anonymized based on the data owner's policies.



```
2017-04-07 06:59:22.787 32758 INFO nova.osapi_compute.wsgi.server [req-3e5282fc-7595-40e4-9054-73bde7ba7edb None] UoNsM  e62cd089800740bb8d698290cdd4caaf/servers/
3eedb8d9-c842-4e6e-8909-ea46c87f2728/action HTTP/1.1" status: 500 len: 354 time: 61.0821550
2017-04-07 07:00:23.870 32751 INFO nova.osapi_compute.wsgi.server [req-811aa4db-9d4a-49e3-b8c9-bd  xNo+jD563VYEC4c= "POST /v2/servers/action HTTP/1.1" status: 500 len: 354 time:
61.0765841
2017-04-07 07:05:28.033 32750 INFO nova.osapi_compute.wsgi.server [req-f9af9ae0-d485-460c-9245-29e  xNo+jD563VYEC4c= "POST /v2/servers/action HTTP/1.1" status: 500 len: 354 time:
61.0763061
2017-04-07 07:42:13.095 32740 INFO nova.osapi_compute.wsgi.server [req-9142c8f6-ae51-4403-aa70-6adac47f8c9c None] UoNsM  q22e5cdd844a422cbfd892f68b88a14fd/servers
HTTP/1.1" status: 202 len: 764 time: 2.0447860
2017-04-07 07:42:18.393 345 INFO nova.api.ec2 [-] 0.175s UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= OPTIONS / None:None 200 [None] text/plain text/plain
2017-04-07 07:42:18.394 345 INFO nova.metadata.wsgi.server [-] UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "OPTIONS / HTTP/1.0" status: 200 len: 234 time: 0.0013409
2017-04-09 15:21:18.980 32728 INFO nova.osapi_compute.wsgi.server req-799784d5-a03c-4196-bf68-5b93b395cd8e  lo+jD563VYEC4c= "POST /v2/q22e5cdd844a422cbfd892f68b88a14fd/
servers HTTP/1.1" status: 202 len: 764 time: 2.587
2017-04-09 15:56:11.848 32730  [req-86b763b4-225d-4f10-8bd3-32b798f9c14d None] UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "POST /v2/q22e5cdd844a422cbfd892f68b88a14fd/servers HTTP/1.1" status:
202 len: 764 time: 2.1304612
2017-04-09 15:56:14.866 3585 INFO neutron.wsgi [req-36060cd4-7e27-409f-95fd-7323492cbbe7 None] UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo  POST /v2.0/ports.json HTTP/1.1" 201 941 0.434647
2017-04-10 08:41:03.750 32729 INFO nova.osapi_compute.wsgi.server [req-74671e85-ce4a-4a20-9bb6-200c9512a0fc None] UoNsM7DEO+lXL4  9883nnk1l12/flavors/detail HTTP/1.1" status:
200 len: 6027 time: 0.0357180
2017-04-10 08:41:05.626 32741 INFO nova.osapi_compute.wsgi.server [req  UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "GET /v2/
ffdqa2134nkm1lkjk76598718nnk1l12/flavors/2 HTTP/1.1" status: 200 len: 61
2017-04-10 08:41:12.317 32748 INFO nova.osapi_compute.wsgi.server [req-03fdb35a-3dbc-420c-bc84-22bda72d12fe None] UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "GET /v2/qwd12yh3412f4wh531902ju4312www21/servers/detail HTTP/1.1"
status: 200 len: 34955 time: 2.9434948
2017-04-10 08:39:31.245 32753  rver [req-fa8288ae-6d1a-4a6a-833c-36c31f036993 None] UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "POST /v2/qwd12yh3412f4wh531902ju4312www21/os-server-
 me: 0.2683861
2017-04-10 08:39:32.660 32752 INFO nova.osapi_compute.wsgi.server [req-0e60e7d6-bdfb-41ec-af43-71dc68e5b352 None] UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "GET /v2/66b34472cbeb42adbb01929d9c1cec6f/flavors/detail HTTP/1.1" status:
200 len: 6027 time: 0.0270388
2017-04-10 08:39:32.962 32753 INFO nova.osapi_compute.wsgi.server [req-68141eff-1fa9-4605-8a2f-b08545ec329a None] UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "GET /v2/66b34472cbeb42adbb01929d9c1cec6f/flavors/detail HTTP/1.1" status:
200 len: 6027 time: 0.0366302
2017-04-10 08:39:33.162 333 INFO nova.api.ec2 [-] 0.144s UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= OPTIONS /  ain text/plain
2017-04-10 08:39:40.384 32740 INFO nova.osapi_compute.wsgi.server [req-e4171781-df17-4867-bb96-64f207a19032 None] UoN  DELETE /v2/servers/d343e8832ce042b7982a8a316313179c
HTTP/1.1" status: 204 len: 198 time: 0.3207819
```

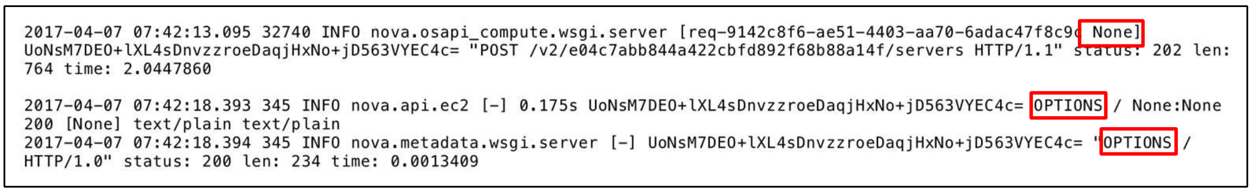
User ID
Same log entries can be mapped to different event types
Tenant ID
Request ID
Process ID
Create port event
Hashed IP addresss
Timestamp
ID of the deleted resource

Figure 5.4: Identification of useful information in one of the logs collected from the real cloud.

Background on OpenStack Logging. Logging systems can provide an essential resource for both troubleshooting and behavior analysis of the underlying clouds. To this end, tracing back log entries to identify the root cause of a problem and subsequent actions, respectively, are natural solutions which motivate the processing of logs. Furthermore, the high complexity and ever-growing nature of cloud environments further increase the need for processing logs in a cloud. To this end, OpenStack [89] logs different user and system actions performed within the cloud. The most commonly used log format in OpenStack services starts with the timestamp, process ID, log level, the program generating the log, an ID, followed by a message field, which might be divisible into smaller informative segments such as request method and URL Path. Furthermore, different OpenStack services write their log files to their corresponding subdirectory of the /var/log directory in the node they are installed in. For example, the log location of Nova is /var/log/nova. Figure 5.4 depicts an excerpt of the logs collected from a real cloud highlighting the useful information stored in these logs.

Investigated Factors. In the following, we describe different factors in the logs that are relevant to automate the learning of dependency models in LeaPS.

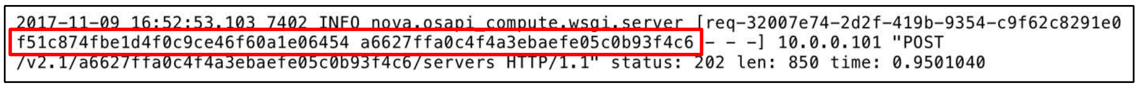
i) **Layouts.** The first factor that we investigated is the general layout of logs. While comparing the layouts of the logs, we found that there are different attributes in each log entry, and those attributes vary based on the version and the logging service of OpenStack. This was exasperated by the lack of detailed documentation describing the meaning of these attributes. Through our study, we identified 11 fields that are used in OpenStack logging: `timestamp`, `process-ID`, `log-type`, `method`, `request-ID`, `user-ID`, `tenant-ID`, `IP address`, `API URL`, `HTTP response` and `request-length`. Apart from common layout issues, we also observed discrepancies in layouts of the logs collected from the two different cloud deployments as both environments were managed by different versions of OpenStack[1]. The following example depicts the latter observation.



```
2017-04-07 07:42:13.095 32740 INFO nova.osapi_compute.wsgi.server [req-9142c8f6-ae51-4403-aa70-6adac47f8c9d None]
UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "POST /v2/e04c7abb844a422cbfd892f68b88a14f/servers HTTP/1.1" status: 202 len:
764 time: 2.0447860

2017-04-07 07:42:18.393 345 INFO nova.api.ec2 [-] 0.175s UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= OPTIONS / None:None
200 [None] text/plain text/plain
2017-04-07 07:42:18.394 345 INFO nova.metadata.wsgi.server [-] UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "OPTIONS /
HTTP/1.0" status: 200 len: 234 time: 0.0013409
```

(a)



```
2017-11-09 16:52:53.103 7402 INFO nova.osapi_compute.wsgi.server [req-32007e74-2d2f-419b-9354-c9f62c8291e0
f51c874fbe1d4f0c9ce46f60a1e06454 a6627ffa0c4f4a3ebaefe05c0b93f4c6 - - -] 10.0.0.101 "POST
/v2.1/a6627ffa0c4f4a3ebaefe05c0b93f4c6/servers HTTP/1.1" status: 202 len: 850 time: 0.9501040
```

(b)

Figure 5.5: Excerpts of unique fields and entries found only in either studied version of OpenStack in (a) real cloud or (b) testbed cloud.

Figure 5.5 shows three examples of fields that are only present in one of the studied versions[2] of OpenStack: i) The logs from the real cloud does not have any user IDs; instead they store `none`; ii) the real cloud logs have entries starting with `OPTIONS`; and iii) the testbed log entries contain `user-ID` and `tenant ID`.

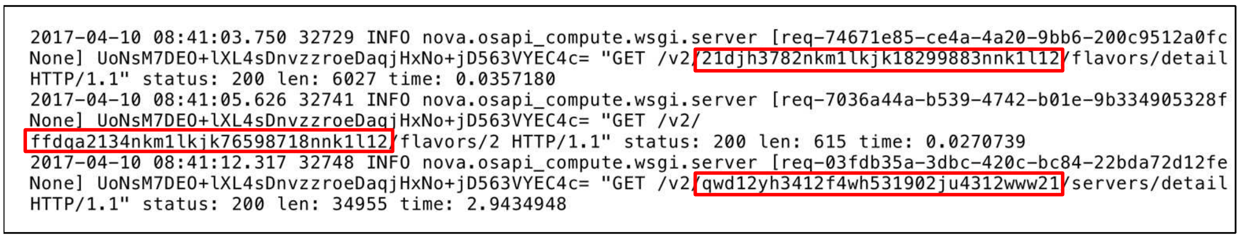
ii) **Log Entries.** After identifying these differences in the layout, we scrutinize each log entry to enable the understanding of the meaning of these entries and their related attributes. We observe that OpenStack logs a wide-range of system-initiated events related to the coordination between different cloud services (e.g., compute, network and storage). Such events are usually logged with a special tenant ID (`tenant-service`). The first row of Table 5.1 shows an example of such a log entry, where the ID of the `tenant-service`

[1]We avoid disclosing the exact version details for the sake of security

[2]Despite that the studied versions are directly consecutive, there are multiple differences in the logging system

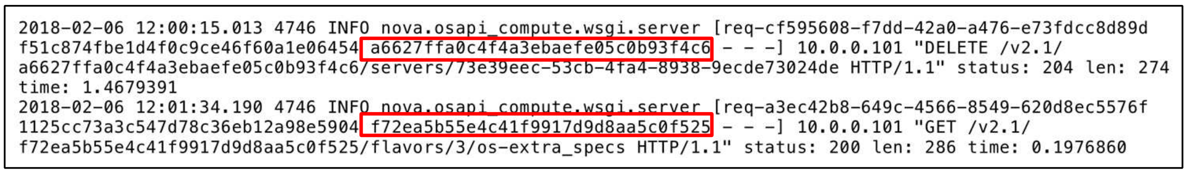
is `dsfre23de8111214321def6e1e834re31`. Moreover, there are requests to list resources or their corresponding details, which are made with GET in their method field. For instance, the second row of Table 5.1 shows a logged event for the tenant ID (`77c433dsf43123edcc12349d9c16fcec`) to render the Flavors (the component to show different resource consumptions by a VM). Both resource rendering and system-initiated logged events has no effect on changing cloud configurations, and therefore, these events are not useful for the auditing purpose.

Additionally, we notice that user-generated requests made under different tenants, are jointly logged into the same log file. Furthermore, their log entries could be distinguished from each other based on the associated tenant ID field identifying the tenant initiating the request. Figure 5.6 highlights the different tenant IDs present in some entries within the log files of both real and testbed clouds.



```
2017-04-10 08:41:03.750 32729 INFO nova.osapi_compute.wsgi.server [req-74671e85-ce4a-4a20-9bb6-200c9512a0fc
None] UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "GET /v2/21djh3782nkm1lkjk18299883nnk1l12/flavors/detail
HTTP/1.1" status: 200 len: 6027 time: 0.0357180
2017-04-10 08:41:05.626 32741 INFO nova.osapi_compute.wsgi.server [req-7036a44a-b539-4742-b01e-9b334905328f
None] UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "GET /v2/
ffdqa2134nkm1lkjk76598718nnk1l12/flavors/2 HTTP/1.1" status: 200 len: 615 time: 0.0270739
2017-04-10 08:41:12.317 32748 INFO nova.osapi_compute.wsgi.server [req-03fdb35a-3dbc-420c-bc84-22bda72d12fe
None] UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "GET /v2/qwd12yh3412f4wh531902ju4312www21/servers/detail
HTTP/1.1" status: 200 len: 34955 time: 2.9434948
```

(a)



```
2018-02-06 12:00:15.013 4746 INFO nova.osapi_compute.wsgi.server [req-cf595608-f7dd-42a0-a476-e73fdcc8d89d
f51c874fbe1d4f0c9ce46f60a1e06454 a6627ffa0c4f4a3ebaefe05c0b93f4c6 - - -] 10.0.0.101 "DELETE /v2.1/
a6627ffa0c4f4a3ebaefe05c0b93f4c6/servers/73e39eec-53cb-4fa4-8938-9ecde73024de HTTP/1.1" status: 204 len: 274
time: 1.4679391
2018-02-06 12:01:34.190 4746 INFO nova.osapi_compute.wsgi.server [req-a3ec42b8-649c-4566-8549-620d8ec5576f
1125cc73a3c547d78c36eb12a98e5904 f72ea5b55e4c41f9917d9d8aa5c0f525 - - -] 10.0.0.101 "GET /v2.1/
f72ea5b55e4c41f9917d9d8aa5c0f525/flavors/3/os-extra_specs HTTP/1.1" status: 200 len: 286 time: 0.1976860
```

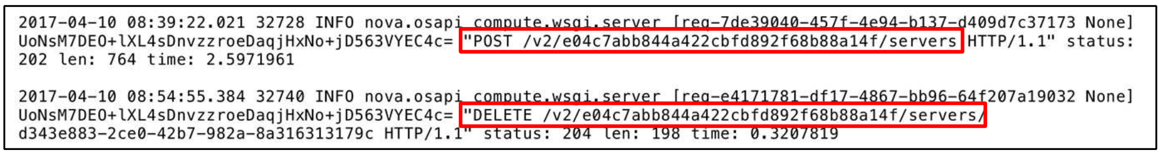
(b)

Figure 5.6: Parts of log entries belonging to different tenants (highlighting corresponding tenant IDs) in (a) real cloud or (b) testbed cloud.

OpenStack Log Entry
`"...POST /v2/dsfre23de8111214321def6e1e834re31/ os-server-external-events HTTP/1.1..."`
`"...GET /v2/77c433dsf43123edcc12349d9c16fcec/ flavors/detail HTTP/1.1..."`

Table 5.1: An excerpt of the log entries corresponding to system initiated events in the real cloud.

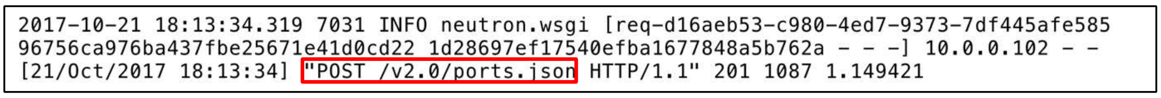
iii) **Type of Events.** In this part of the case studies, we investigate the process of identifying event types from user-generated requests. Usually, OpenStack user requests are transmitted to the server as REST API calls. Thus, our next step is to obtain the event type from each log entry. However, relying only on the REST methods (e.g., POST, GET, PUT, etc.) does not help as it does not uniquely map to a specific event type. Therefore, we study the API documentation of OpenStack to identify specific path information along with the REST methods (called `URL path`) to pinpoint each corresponding event type. Figure 5.7 shows examples of log entries highlighting URL paths corresponding to different event types. The URL paths in the figure actually refer to the event types *create VM*, *delete VM* and *create port*, respectively. However, there are event types for which we observe the same `URL paths`. For instance, Table 5.2 shows three examples of such URL paths from some log entries in the real cloud. Even though the three rows in the table correspond to three different events, their URL paths look identical except the `VM ID` field, which indicates that these are VM related events, but does not help to uniquely identify the event type. As a result, using only those `URL paths` to identify their corresponding event types is insufficient.



```
2017-04-10 08:39:22.021 32728 INFO nova.osapi_compute.wsgi.server [req-7de39040-457f-4e94-b137-d409d7c37173 None]
UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "POST /v2/e04c7abb844a422cbfd892f68b88a14f/servers HTTP/1.1" status:
202 len: 764 time: 2.5971961

2017-04-10 08:54:55.384 32740 INFO nova.osapi_compute.wsgi.server [req-e4171781-df17-4867-bb96-64f207a19032 None]
UoNsM7DEO+lXL4sDnvzzroeDaqjHxNo+jD563VYEC4c= "DELETE /v2/e04c7abb844a422cbfd892f68b88a14f/servers/
d343e883-2ce0-42b7-982a-8a316313179c HTTP/1.1" status: 204 len: 198 time: 0.3207819
```

(a)



```
2017-10-21 18:13:34.319 7031 INFO neutron.wsgi [req-d16aeb53-c980-4ed7-9373-7df445afe585
96756ca976ba437fbe25671e41d0cd22 1d28697ef17540efba1677848a5b762a - - -] 10.0.0.102 - -
[21/Oct/2017 18:13:34] "POST /v2.0/ports.json HTTP/1.1" 201 1087 1.149421
```

(b)

Figure 5.7: The format (highlighted URL paths as REST APIs) of OpenStack logs collected from (a) real cloud and (b) testbed cloud, to show different cloud events (e.g., delete VM and create VM).

iv) **Correlations of Events.** Once event types of different log entries are identified, we need to investigate the relationships between these events and how they correlate. We find out that multiple log entries in different log files of different services correspond to the same user request; which implies that to complete certain user-requests, OpenStack internally

OpenStack Log Entry	Event Name
"POST /v2.1/a6627ffa0c4f4a3ebaefe05c0b93f4c6/ servers/f6128951-0c48-4a11-8b8b-5e96da77b698/"	Stop VM
"POST /v2.1/a6627ffa0c4f4a3ebaefe05c0b93f4c6/ servers/1223d052-bc35-485a-9237-1830bca80fd7/"	Start VM
"POST /v2.1/a6627ffa0c4f4a3ebaefe05c0b93f4c6/ servers/4c886192-43ad-4f98-90dd-34e24c84fcd0/"	Add security group

Table 5.2: Examples of similar URL paths corresponding to different cloud event types.

calls multiple APIs involving different services, thus generating multiple logged events. Table 5.3 shows an example, where the actual user request is to create a VM. However, we observe at least two entries (`create VM` and `create port`) in `nova-api.log` and `neutron-server.log` log files, respectively, to complete the related request. Additionally, we notice that there are log entries in nova-api.log and neutron-server.log with the same `timestamps` (`2017-11-01 18:17:16.345`) corresponding to two different events (`Add security group` and `Create port`). Thus, distinguishing the right precedence relations between events cannot only rely on logged timestamps.

OpenStack Log Entry	Log File
2017-04-09 15:56:14.866 ... "POST /v2/e04c7abb844a422cbfd892f68b88a14f/servers HTTP/1.1"...	nova-api.log
2017-04-09 15:56:11.848 ... "POST /v2.0/ports.json HTTP/1.1"...	neutron-server.log

Table 5.3: Multiple entries in different logs corresponding to the same user request (i.e., create VM).

v) **Session Identification.** Finally, we need to split the log files into groups of events mainly based on the contexts (e.g., same user events) or session. The main intention behind this step is to prepare inputs for different learning techniques, many of which accept inputs as a sequence of events. However, in OpenStack logs, we observe that there exists no session specific information. Moreover, most log entries do not include the requestor ID, which could have been useful to identify the context.

5.3.2 Real-World Challenges to Log Processing

In this section, we summarize the main challenges in processing cloud logs, from the above-mentioned study.

- **Heterogeneous Formats:** Our study shows that the cloud logs may have heterogeneous formats. The log formats vary within the same cloud platforms as well as for different versions of the management system; which includes varieties of different specifications (e.g., fields or attributes). In most cases, the logged information (fields) is not explicitly mentioned in the log file. As a result, it is non-trivial to interpret the log entries. Additionally, the log entries are not systematically generated. Therefore, processing such logs efficiently and systematically is challenging. To handle this challenge, we parse logs and store them in a structured manner by marking each attribute of logs. We will design a method for parsing in Section 5.3.3.2.

- **Ungrouped and Irrelevant Log Entries:** Log entries of OpenStack jointly log all tenants' requests, and contain system initiated entries which are typically irrelevant to the auditing system. However, to detect / prevent security concerns at the tenant-level, it is essential to separately analyze the log entries related to each tenant and hence, current log formats are not appropriate for this purpose. Additionally, cloud generates many internal events to process a user request and store log entries corresponding to those system-initiated events in the same log files. As a result, the analyses requiring to distinguish user actions from system actions can be hampered. This challenge will be tackled by grouping tenant-specific log entries and eliminating entries related to system-initiated events. More details are in Section 5.3.3.3.

- **Difficulties in the Identification of Event Types:** Identifying event types corresponding to each log entry is non-trivial due to the following reasons. First, while most clouds support REST APIs to request different operations (e.g., *create VM*, *create port* and *update port*), the event types are not obvious from the API and require to check the whole URL paths. Second, in some cases, URL paths for different events are the same, and hence, such paths are not sufficient to identify these events uniquely. We call this kind of events *ambiguous* event. Ambiguous events will be properly identified by leveraging a special log option called `request body` in OpenStack. We detail the solutions to these challenges in Section 5.3.3.4.

- **Distributed Logging for Different Services:** Activities in different services (e.g., compute, network and storage) are logged separately. However, for a log analysis, which involves different services combinedly such as ours, merging logs from these services is essential, but not trivial. There are certain requests that result in multiple log entries distributed over different log files by different services. Additionally, due to time synchronization issues between the services, multiple events may be logged with the same timestamp, which hinders analysis tools to extract the right precedence relationships between them. However, we noticed that the synchronization issue does not concern events related to the same tenant but concerns different concurrent tenants behaviors. As we are tackling tenants separately, this turns out not affecting our analysis. Other challenges in this finding will be addressed by merging logs from different services and eliminating duplicate entries corresponding to the same user request. We provide more details of the solution in Section 5.3.3.5.

- **Need for a Sequence Identification Solution:** There exists no session specific information in the logs. Also, the requester ID for a user request is missing in most log entries. Therefore, there is a need for a solution to identify sequences of events from these specific formats of logs fulfilling all the requirements (e.g., preserving transitions and their relative order) for a specific or a group of analyses purposes. To this end, we have already requested to the OpenStack community to include requestor ID (at least) within each log entry to facilitate log analysis. Till then, we propose a custom algorithm to identify sequences of events, in which all transitions and their relative orders of the actual log entries are preserved. More details on this solution are presented in Section 5.3.3.6.

5.3.3 Our Solution: LeaPS Log Processing

In this section, we discuss our log processing approach, which addresses all of the above-mentioned challenges, and provides more structured and meaningful processed logs for different analyses. A high-level algorithm of our log processor is shown in Algorithm 5.

Algorithm 5 High-Level Algorithm of LeaPS Log Processing

```
Input: Predefined parsing and matching rules
Output: Sequences of events to different log analysis tools (e.g., LeaPS and sequence pattern
    mining)
 1: procedure PROCESSLOGS(CloudOS)
 2:     for each component ∈ CloudOS do
 3:         Parse the raw logs;
 4:         Group parsed logs based on tenant IDs;
 5:         Prune irrelevant log entries (system-initiated and UI rendering);
 6:         Mark event types based on information in URL path and request body;
 7:     Combine logs from different services (e.g., compute and network)
 8:     for each log entry ∈ combinedLogs do
 9:         Identify Sequences from the combined logs
10:     return Sequences
```

In the following, we briefly describe main steps of the log processing algorithm.

- **Line 3:** parses raw logs into a structured format. This step extracts identified fields in the log entries and uses them together with a set of pre-defined rules to parse the raw log into a structured log (e.g., CSV) file. This allows handling the heterogeneity of log formats.

- **Line 4:** groups parsed logs based on tenant IDs. The latter, easily identified in the obtained structured log file, allows grouping log entries based on the tenants under which the events are being logged. This tackles the issue of ungrouped events.

- **Line 5:** prunes irrelevant log entries. System-initiated log entries can be grouped and discarded easily based on the system tenant ID (i.e., tenant-service) present in each of the related log entries. Those related to the UI rendering actions are identified by inspecting the method used in the URL (e.g., GET) in the entries related to the logged API calls.

- **Line 6:** identifies the type of events for each log entry. We first identify event types from the method and path information available from Line 3. However, there are several event types (a.k.a. ambiguous events), which have the same method and path information. To tackle this, we further check the request body, which contains detailed information for each log entry, mainly by tuning logging options to include the missing information.

- **Line 7:** combines logs from different services (e.g., compute and network) based on different attributes (e.g., tenant id and request id) and timestamps. This step draws the correlation among events logged in different services so that it can handle the challenges mentioned in Section 5.3.2.

- **Lines 8-10:** construct the event sequences based on the occurrences of events in the actual log fulfilling the requirements mentioned in Section 5.3.2. Our log processor provides these event sequences as outputs, which can be later used by different analysis methods (e.g., LeaPS learning system in Section 5.4 and sequence pattern mining algorithms in Section 5.7.2).

Figure 5.8 illustrates an example of the outputs of each of these steps. In the following, we first describe the inputs to our log processing and then provide more details on each processing step.

5.3.3.1 Inputs to LeaPS Log Processing

Apart from raw cloud logs, our log processing algorithm requires two inputs: parsing rules to handle different log entries into a structured format and matching rules to identify the event types. Building these inputs is a one-time effort obtained from our investigations in the aforementioned case study.

Building Parsing Rules. To build the parsing rules, we first study different formats of logs and their corresponding structure. Next, we obtain different fields (e.g., timestamp, process ID and tenant ID) in the log entries and their relative positions in the logs. Finally, we build rules based on the fields and their corresponding orders in logs to support the parsing in Section 5.3.3.2.

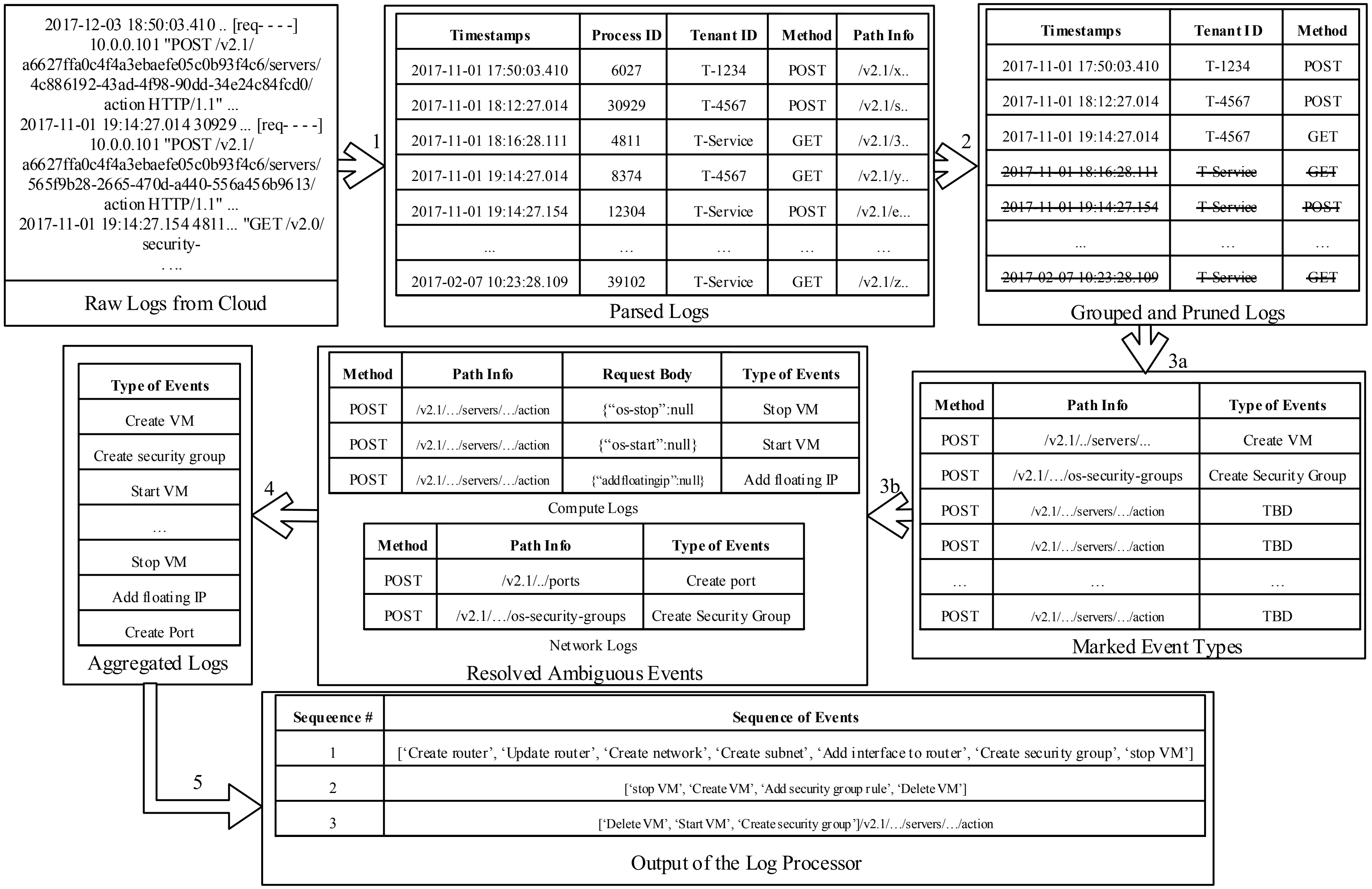
Building Matching Rules. To identify the matching rules, we study the API documentation of OpenStack along with the log formats. To build a relationship between those fields in the logs and their corresponding event types. However, there exist several events for which all these fields are identical and hence, the event types of those events cannot be identified using this procedure. To tackle this, we leverage the request body, which contains detailed information for each log entry. Finally, we provide a complete mapping to identify different event types in Section 5.3.3.4.

5.3.3.2 Parsing Logs

The main purposes of this step are to mark all useful fields of different cloud logs and store them in a more structured way to enhance log analysis effort in terms of efficiency. We achieve these purposes, by parsing logs based on the pre-defined rules so that each identified field is marked with a meaningful name, and storing the logs in a more structured manner (e.g., in CSV) converting from a text-based file. For example, as shown in Step 1 of Figure 5.8, we collect OpenStack logs from compute and network services. The first log entry `2017-12-03 18:50:03.410 .. [req- - - -] 10.0.0.101 "POST /v2.1/a6627ffa0c4f4a3ebaefe05c0b93f4c6/ servers/4c886192-43ad-4f98-90dd-34e24c84fcd0/ action HTTP/1.1"` contains the fields `timestamp`, `process ID`, `request ID`, `IP address`, `method`, `path info`, `tenant ID`, respectively. The next step is to parse each entry of the logs based on these fields and their relative positions, and to store them in a table (*Parsed Logs* in Figure 5.8).

5.3.3.3 Grouping and Pruning of Parsed Logs

Extracting the contextual information, and separating both system and user initiated activities from the logs are the main goals of this step. First, we identify different contextual information such as tenant ID and accessed resource IDs (e.g., port ID, VM ID and subnet ID). Then, we group log entries based on the tenant ID so that we can obtain tenant specific activities together. Next, we extract the accessed resource IDs by the activities of all log entries. Finally, we separate log entries of user initiated activities from that of system initiated activities. Note that, the tenant ID field of the log entries for system-initiated events contains a special value (i.e.,



```
2017-12-03 18:50:03.410 .. [req- - - -]
10.0.0.101 "POST /v2.1/
a6627ffa0c4f4a3ebaefe05c0b93f4c6/servers/
4c886192-43ad-4f98-90dd-34e24c84fcd0/
action HTTP/1.1" ...
2017-11-01 19:14:27.014 30929 ... [req- - - -]
10.0.0.101 "POST /v2.1/
a6627ffa0c4f4a3ebaefe05c0b93f4c6/servers/
565f9b28-2665-470d-a440-556a456b9613/
action HTTP/1.1" ...
2017-11-01 19:14:27.154 4811... "GET /v2.0/
security-
....
```

Timestamps	Process ID	Tenant ID	Method	Path Info
2017-11-01 17:50:03.410	6027	T-1234	POST	/v2.1/x..
2017-11-01 18:12:27.014	30929	T-4567	POST	/v2.1/s..
2017-11-01 18:16:28.111	4811	T-Service	GET	/v2.1/3..
2017-11-01 19:14:27.014	8374	T-4567	GET	/v2.1/y..
2017-11-01 19:14:27.154	12304	T-Service	POST	/v2.1/e...
...	…	…	…	…
2017-02-07 10:23:28.109	39102	T-Service	GET	/v2.1/z..

Timestamps	Tenant ID	Method
2017-11-01 17:50:03.410	T-1234	POST
2017-11-01 18:12:27.014	T-4567	POST
2017-11-01 19:14:27.014	T-4567	GET
~~2017-11-01 18:16:28.111~~	~~T-Service~~	~~GET~~
~~2017-11-01 19:14:27.154~~	~~T-Service~~	~~POST~~
...	…	…
~~2017-02-07 10:23:28.109~~	~~T-Service~~	~~GET~~

Method	Path Info	Type of Events
POST	/v2.1/../servers/...	Create VM
POST	/v2.1/…/os-security-groups	Create Security Group
POST	/v2.1/…/servers/…/action	TBD
POST	/v2.1/…/servers/…/action	TBD
…	…	…
POST	/v2.1/…/servers/…/action	TBD

Method	Path Info	Request Body	Type of Events
POST	/v2.1/…/servers/…/action	{"os-stop":null	Stop VM
POST	/v2.1/…/servers/…/action	{"os-start":null}	Start VM
POST	/v2.1/…/servers/…/action	{"addfloatingip":null}	Add floating IP

Method	Path Info	Type of Events
POST	/v2.1/../ports	Create port
POST	/v2.1/…/os-security-groups	Create Security Group

Type of Events
Create VM
Create security group
Start VM
…
Stop VM
Add floating IP
Create Port

Sequence #	Sequence of Events
1	['Create router', 'Update router', 'Create network', 'Create subnet', 'Add interface to router', 'Create security group', 'stop VM']
2	['stop VM', 'Create VM', 'Add security group rule', 'Delete VM']
3	['Delete VM', 'Start VM', 'Create security group']/v2.1/…/servers/…/action

Figure 5.8: An excerpt of outputs after each step of our log processor.

tenant-service) in OpenStack. However, there exist exceptions where system initiated events are stored under an existing tenant. Therefore, we maintain a list of such exceptions and match them with the log entries while separating user-initiated events.

For instance, the *Parsed Logs* in Figure 5.8 contain entries from tenants `T-1234, T-4567, T-6789` and `T-Service`. After Step 2, in the *Grouped and Pruned Logs*, we first store all entries from the `T-1234` tenant, and then, similarly store entries for `T-4567, T-6789` and `T-Service`. Afterwards, we identify all log entries with the tenant ID `T-Service` and store them separately.

5.3.3.4 Marking Event Types

Marking the corresponding event types using the pre-defined matching rules (as shown) for each log entry is the main objective of this step. Based on the set of fields used in marking, there are two categories of event types. For the first category, we use the URL path (which includes `method`, `resources`, etc.). For example, from the *Grouped Logs* in Figure 5.8, we identify `method`, `resources`, `resource ID` and `action` as the potential fields which together may provide unique information about each event type. Based on this assumption, we build the matching rules as shown in Table 5.4. The first entry in the table shows that the fields `method: POST` and `resources: ports` indicate the `create port` event type. Whereas, to identify the `add interface to router` event type, we require the `action: add_router_interface` field along with the `method`, `resources` and `resource ID` fields. However, for the last entry in the table, these fields are not sufficient to obtain the event type, as they have the same values for `method`, `resources`, `resource ID` and `action` fields for multiple event types. These event types are considered as the second category and marked in the following manner.

For the second category of event types, we identify event type of a log entry, we match the request ID in the log file and request body, and look into the matching rules for that particular request body. For instance, Table 5.5 shows examples of matching rules between request body and event types. The first and second rows of the table show that the request body values `{"os-start": null}` and `{"os-stop": null}` ensure that the requested

Method	Resources	Resoruce ID	Action	Event Type
`POST`	ports	NaN	NaN	`Create port`
`PUT`	ports	port ID	NaN	`Update port`
`DELETE`	ports	port ID	NaN	`Delete port`
`PUT`	routers	router ID	add_router _interface	`Add interface to router`
`PUT`	routers	router ID	remove_router _interface	`Remove interface to router`
`POST`	floating-IPs	NaN	NaN	`Create floating IP`
`PUT`	floating-IPs	VM ID	NaN	`Associate floating IP`
`POST`	servers	VM ID	actions	`TBD`

Table 5.4: An excerpt of the mapping to obtain the event types from URL-method and path_info. Note that, the term 'NaN' is used by OpenStack to indicate the unrepresentable value for the specific fields, and the term 'TBD' is to indicate that the corresponding event type is not conclusive based on the identified fields.

Request Body	Event Type
`req_body: {"os-start": null}`	`Start VM`
`req_body: {"os-stop": null}`	`Stop VM`
`req_body: {"addSecurityGroup": {"name": "essential"}}`	`Add security group`
`req_body: {"addFloatingIp": {"name": "leapsVM"}}`	`Add floating IP`

Table 5.5: Examples of identifying event types from request bodies for the event types as the last row of Table 5.4.

event types are `start VM` and `stop VM`, respectively. The third and fourth rows provide event types (`Add security group` and `Add floating IP`) along with their involved security group name (`essential`) and instance name (`leapsVM`), respectively.

In summary, we utilize the method, resource, resource id, action and request body fields to mark all event types.

5.3.3.5 Aggregating Logs

Merging logs from different services (e.g., compute, network and storage) is the main goal of this step. To this end, we first combine multiple log files and sort them based on timestamp. Next, we identify entries with the same timestamp (if any), and mark them specially to later identify that they occurred at the same time in different services (if that helps any log

OpenStack Log Entry	Event Type	Initiated by
`"POST /v2.1/a6627ffa0c4f4a3ebaefe05c0b93f4c6/servers HTTP/1.1"`	`Create VM`	`User`
`"POST /v2.0/ports.json HTTP/1.1"`	`Create Port`	`System`

Table 5.6: Showing part of multiple log entries in compute and network services referring to one user request (Create VM).

analysis mechanism). Also, we identify any duplicate entry in different logs (as mentioned in Section 5.3.1 that the same event might be logged in multiple services), and only keep the corresponding log to the actual user request.

For example, Stage 5 in Table 5.6 shows two entries each from the compute service (`Nova`) and network service (`Neutron`), respectively. The first row of the table is for the `Create VM` event, which is actually initiated by a cloud user. On the other hand, the `Create port` event is a system initiated event as a result of the user request. In other words, OpenStack creates a port by itself while creating a VM.

5.3.3.6 Generating Outputs

Our log processor provides outputs as sequences of events. In this work, we mainly observe the following three requirements for identifying events sequences. First, we preserve all transitions that are present in the actual logs. Second, we maintain the relative order between events. Third, in each sequence, we avoid cycles (by starting a new sequence when there is a repetition) to facilitate capturing relationships between events (e.g., dependencies in our model), flowing from top to bottom. To validate our approach, we use the generated events sequences to build a Bayesian network and to perform sequence pattern mining in Sections 5.4 and 5.7.2, respectively.

To generate the final output (i.e., sequences of events), the log processor performs the following steps:

- Read event types sorted by timestamp in the *Aggregated Logs*, and group event types in a sequence till any event type is observed for the second time. In other words, a sequence Seq_i contains all event types from $Event_m$ to $Event_{n-1}$, where the $Event_n$ (the successor of $Event_{n-1}$) has already been observed in the sequence Seq_i. Thus, no sequence contains any repeated event types and hence, we avoid cycles in sequences.

- Start the next sequence from the last element of the previous sequence so that all transitions within the sequences are preserved. In other words, the following sequence, Seq_{i+1}, starts with the $Event_{n-1}$, which is the last event of the Seq_i sequence.

Aggregated Log Content
{`Create router`, `Update router`, `Create network`, `Create VM`, `Add interface to router`, `Create security group`, `Start VM`, `Create VM`, `Add security group rule`, `Delete VM`, `Create VM`, `Create security group`, `Start VM`}

Output
Seq_1 = {`Create router`, `Update router`, `Create network`, `Create VM`, `Add interface to router`, `Create security group`, `Start VM`}
Seq_2 = {`Start VM`, `Create VM`, `Add security group rule`, `Delete VM`}
Seq_3 = {`Delete VM`, `Create VM`, `Create security group`, `Start VM`}

Table 5.7: An excerpt of outputs from LeaPS log processor.

In summary, our log processing approach addresses all challenges discussed in Section 5.3.2. As an example, it provides sequences of events as shown in Table 5.7. Later, these sequences will be utilized by our learning system to learn the dependency model presented in the next section. The implementation details including the algorithms for our log processing are presented in Section 5.6.3. Also, the performance evaluation of our log processor is shown in Section 5.7.2.

5.4 LeaPS Learning System

This section first describes the dependency model and then, presents the steps to learn probabilistic dependencies for this model.

5.4.1 The Dependency Model

We first demonstrate our dependency model through an example and then formally define the model. The model will be the foundation of our proactive auditing solution (detailed in Section 5.5).

Figure 5.9 shows an example of a dependency model, where nodes represent different event types in a cloud and edges represent transitions between event types. For example, nodes, *create VM* and *create security group*, represent the corresponding event types, and the edge from *create VM* to *create security group* indicates the likely order of occurrence of those event types. The label of this edge, 0.625, means 62.5% of the times an instance of the *create VM* event type will be immediately followed by an instance of the *create security group* event type.

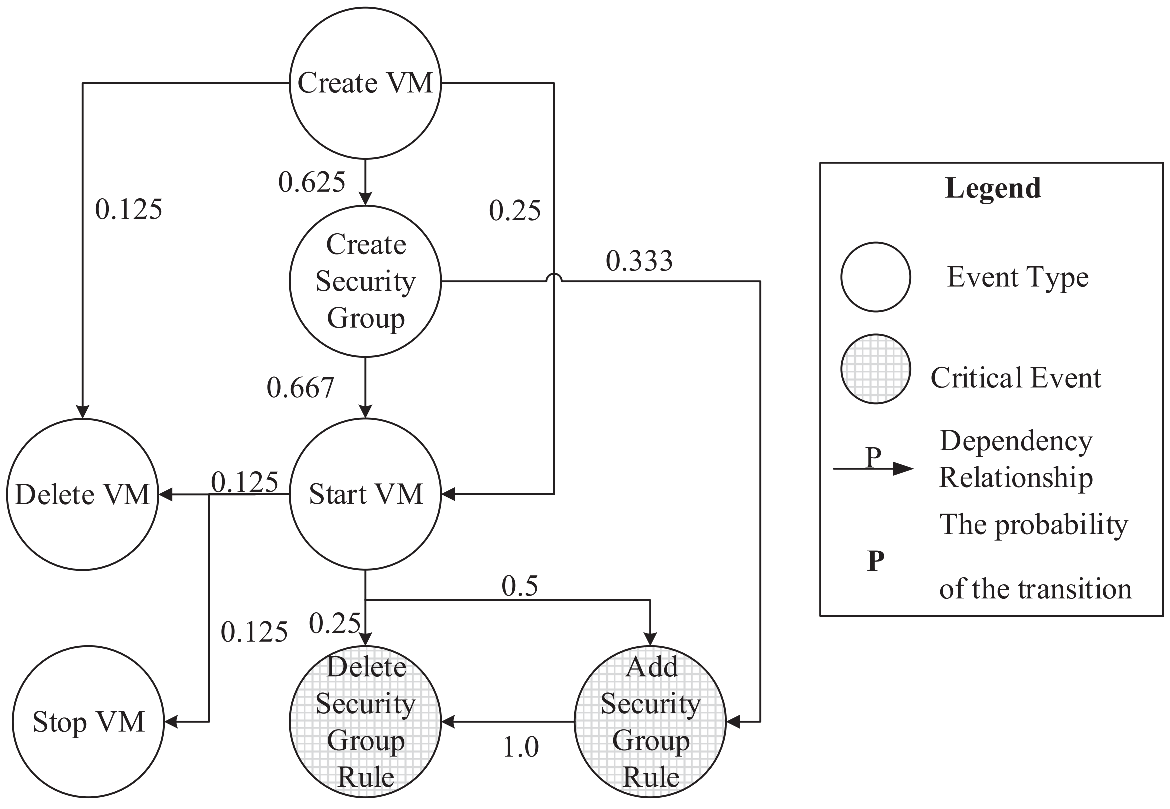


Figure 5.9: An example dependency model represented as a Bayesian network.

Our objective is to automatically construct such a model from logs in clouds. As an example, the following shows an excerpt of the event types *event-type* and historical event sequences *hist* for four days related to the running example of Section 5.2.2.

- *event-type* = {*create VM (CV), create security group (CSG), start VM (SV), delete security group rule (DSG)*}; and
- *hist* = {*day* 1 : *CV*, *CSG*, *SV*; *day* 2 : *CSG*, *SV*; *day* 3 : *CSG*, *DSG*; *day* 4 : *CV*, *DSG*}, where the order of event instances in a sequence indicates the actual order of occurrences.

The dependency model shown in Figure 5.9 may be extracted from such data (note above we only show an excerpt of the data needed to construct the complete model). For instance, in *hist*, *CV* has three immediate successors (i.e., *CSG*, *SV*, *DSG*), and their probabilities can be calculated as $P(CSG|CV) = 0.5$, $P(SV|CV) = 0.5$ and $P(DSG|CV) = 0.5$.

As demonstrated in the above example, Bayesian network [94] suits our needs for capturing probabilistic patterns of dependencies between events types. A Bayesian network is a probabilistic graphical model that represents a set of random variables as nodes and their conditional dependencies in the form of a directed acyclic graph. We choose Bayesian network to represent our dependency model for the following reasons. Firstly, the event types in cloud and their precedence dependencies can naturally be represented as nodes (random variables) and edges (conditional dependencies) of a Bayesian network. Secondly, the need of our approach for learning the conditional dependencies can be easily implemented as parameter learning in Bayesian network. For instance, in Figure 5.9, using the Bayes' theorem we can calculate the probability for an instance of *add security group rule* to occur after observing an instance of *create VM* to be 0.52. More formally, the following defines our dependency model.

Given a list of event types *event-type* and the log of historical events *hist*, the *dependency model* is defined as a Bayesian network $B = (G, \theta)$, where G is a DAG in which each node corresponds to an event type in *event-type*, and each directed edge between two nodes indicates the first node would immediately precede the other in some event sequences in *hist* whose probability is part of the list of parameters, θ.

We say a *dependency* exists between any two event types if their corresponding nodes are connected by an edge in the dependency model, and we say they are not dependent, otherwise. We assume a subset of the leaf nodes in the dependency model is given as *critical events* that might breach some given *security properties*.

5.4.2 Learning Engine

The next step is to learn the probabilistic dependency model from the sequences of event instances in the processed logs. To this end, we choose the parameter learning technique in Bayesian network [79, 44, 94] (this choice has been justified in Section 5.4.1). We now first demonstrate the learning steps through an example, and then provide further details.

Figure 5.10 shows the dependency model of Figure 5.9 with the outcomes of different learning steps as the labels of edges. The first learning step is to define the priori, where the nodes represent the set of event types received as input, and the edges represent possible transitions from an event type, e.g., from the *create VM* event to the *delete VM*, *start VM* and *create security group* events. Then, $P(DV|CV)$, $P(CSG|CV)$, $P(SV|CV)$ and other conditional probabilities (between immediately adjacent nodes in the model) are the parameters; all parameters are initialized with equal probabilities. For instance, we use 0.33 to label each of the three outgoing edges from the *create VM* node. The second learning step is to use the historical data to train the model. For instance, the second values in the labels of the edges of Figure 5.10 are learned from the processed logs obtained from the log processor. The third values in the labels of Figure 5.10 represent an incremental update of the learned model using the feedback from a sequence of runtime events.

This learning mechanism mainly takes two inputs: the structure of the model with its parameters, and the historical data. The structure of the model, meaning the nodes and edges in a Bayesian network, is first derived from the set of event types received as input. To this end, we provide a guideline on identifying such a set of event types in Section 5.8. Initially, the system considers every possible edge between nodes (and eventually deletes the edges with probability 0), and conditional probabilities between immediately adjacent nodes (measured as the conditional probability) are chosen as the parameters of the model. We further sparse the structure into smaller groups based on different security properties (the structure in Figure 5.10 is one of the examples). The processed logs containing sequences of event instances serve as the input data to the learning engine for learning the parameters. Finally, the parameter learning in Bayesian network is performed as follows: i) defining a priori (with the structure and

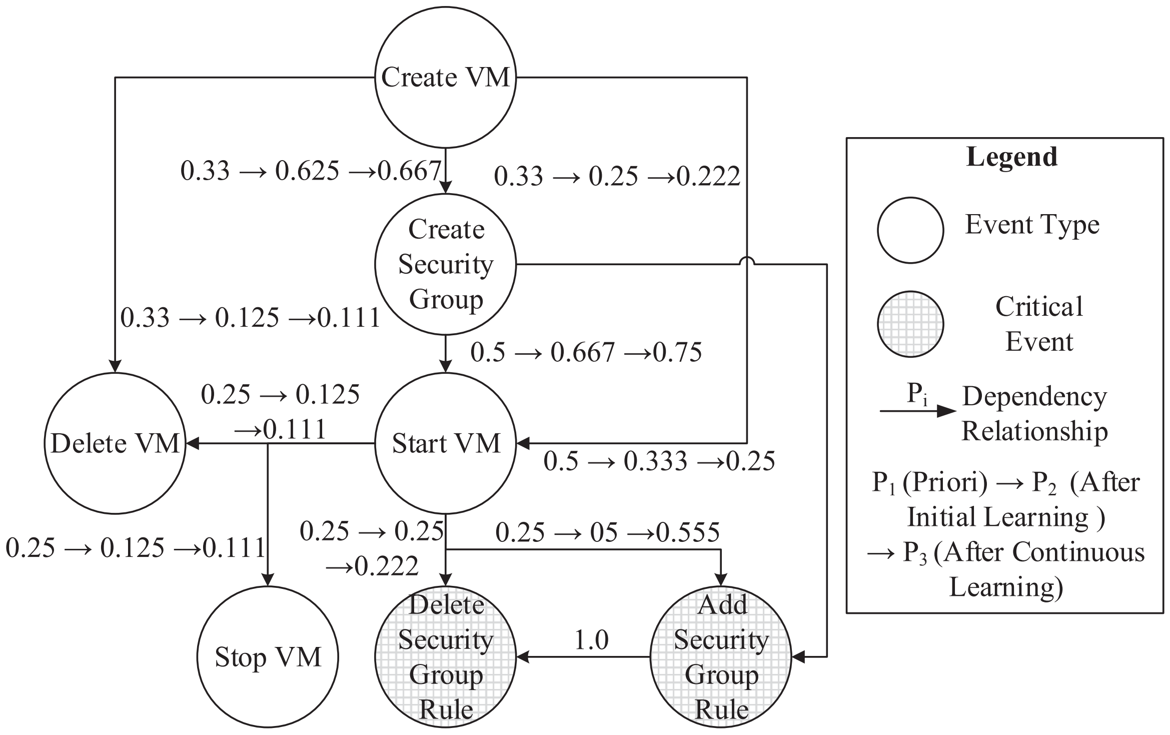


Figure 5.10: The outcomes of three learning steps for the dependency model.

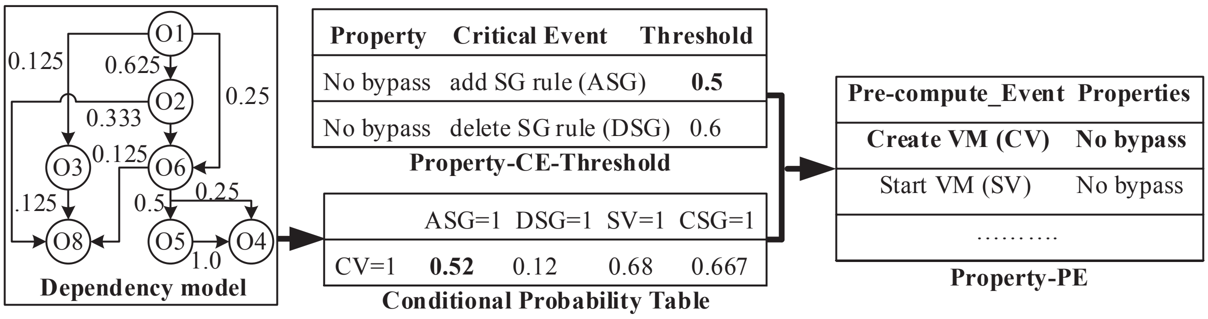
initialized parameters of the model), ii) training the initial model based on the historical data, and iii) continuously updating the learned model based on incremental feedbacks.

5.5 LeaPS Proactive Verification System

This section presents our learning-based proactive verification system.

5.5.1 Likelihood Evaluator

The likelihood evaluator is mainly responsible for triggering the pre-computation. To this end, the evaluator first takes the learned dependency model as input, and derives offline all indirect dependency relationships for each node. Based on these dependency relationships, the evaluator identifies the event types for which an immediate pre-computation is required. Additionally, at runtime the evaluator matches the intercepted event instance with the event type, and decides whether to trigger a pre-computation or verification request.[2] The data manipulated by the likelihood evaluator based on the dependency model will be described using the following example.



Property	Critical Event	Threshold
No bypass	add SG rule (ASG)	**0.5**
No bypass	delete SG rule (DSG)	0.6

Property-CE-Threshold

	ASG=1	DSG=1	SV=1	CSG=1
CV=1	**0.52**	0.12	0.68	0.667

Conditional Probability Table

Pre-compute_Event	Properties
Create VM (CV)	**No bypass**
Start VM (SV)	No bypass
……….	

Property-PE

Figure 5.11: An excerpt of the likelihood evaluator steps and their outputs.

Figure 5.11 shows an excerpt of the steps and their outputs in the likelihood evaluator module. In this figure, the *Property-CE-Threshold* table maps the *no bypass of security group* property [18] with its critical events (i.e., *add security group rule* and *delete security group rule*) and corresponding thresholds (i.e., 0.5 and 0.6). Then, from the conditional probability in the model, the evaluator infers conditional probabilities of all possible successors (both direct and indirect), and stores them in the *Conditional-Probability* table. The conditional probability for *ASG* having *CV* (i.e., *P(ASG/CV)*) is 0.52 in the *Conditional-Probability* table in Figure 5.11. Next, this value is compared with the thresholds of the *no bypass* property in the *Property-CE-threshold* table. As the reported probability is higher, the *CV* event type is stored in the *Property-PE* table so that for the next *CV* event instance, the evaluator triggers a pre-computation.

5.5.2 Pre-Computing Module

The purpose of the pre-computing module is to prepare the ground for the runtime verification. In this work, we mainly discuss watchlist-based pre-computation [71]; where watchlist is a list containing all allowed parameters for different critical events. The specification of contents in a watchlist is defined by the cloud tenant, and is stored in the *Property-WL* table. We assume

[2]This is not to respond to the event as in incident response, but to prepare for the auditing, and the incident response following an auditing result is out of the scope of this work.

that at the time LeaPS is launched, we initialize several tables based on the cloud context and tenant inputs. For instance, inputs including the list of security properties, their corresponding critical events, and the specification of contents in watchlists are first stored in the *Property-WL* and *Property-CE-Threshold* tables. The watchlists are also populated from the current cloud context. We maintain a watchlist for each security property. Afterwards, each time the pre-computation is triggered by the likelihood evaluator, this module incrementally updates the watchlist based on the changes applied to the cloud in the meantime. The main functionality of the pre-computing module is described using the following example.

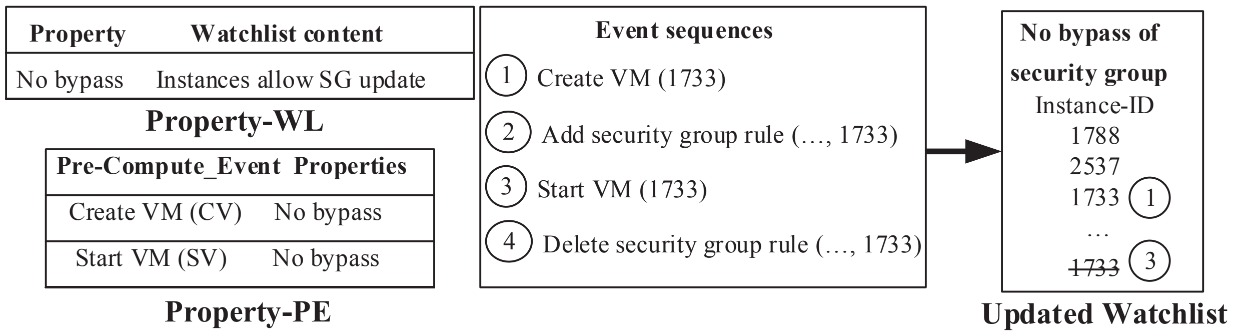


Figure 5.12: Showing steps of the updating watchlist for a sample event sequences.

Left side of Figure 5.12 shows two inputs (*Property-WL* and *Property-PE* tables) to the pre-computing module. We now simulate a sequence of intercepted events (shown in the middle box of the figure) and depict the evolution of a watchlist for the *no bypass* property (right side box of the figure). (1) We intercept the *create VM 1733* event instance, identify the event in the *Property-PE* table, and add VM 1733 to the watchlist without blocking it. (2) After intercepting the *add security group rule (..., 1733)* event instance, we identify that this is a critical event. Therefore, we verify with the watchlist keeping the operation blocked, find that VM 1733 is in the watchlist, and hence, we recommend to allow this operation. (3) We intercept the *start VM 1733* operation and identify the event in the *Property-PE* table. VM 1733 is then removed from the watchlist, as the VM is active. (4) After intercepting the *delete security group rule (..., 1733)* event instance, we identify that this is a critical event. Therefore, we verify with the watchlist keeping the event instance blocked, find that VM 1733 is not in the watchlist, and hence, identify the current situation as a violation of the *no bypass* property.

5.5.3 Feedback Module

The main purposes of the feedback module are: i) to provide feedback to the learning engine, and ii) to provide feedback to the tenant on thresholds for different properties. These purposes are achieved by three steps: storing verification results in the repository, analyzing the results, and providing the necessary feedback to corresponding modules.

Firstly, the feedback module stores the verification results in the repository. Additionally, this module stores the verification result as hit or miss after each critical event, where the hit means the requested parameter is present in the watchlist (meaning no violation), and the miss means the requested parameter is not found in the watchlist (meaning a violation). Additionally, we store the sequence of events for a particular time period (e.g., one day) in a similar format as the processed log described in the learning module. In the next step, we analyze these results along with the models to prepare a feedback for different modules. From the sequence of events, the analyzer identifies whether the pattern is already observed or is a new trend, and accordingly the updater prepares a feedback for the learning engine either to fine-tune the parameter or to capture a new trend. From the verification results, the analyzer counts the number of misses for different properties to provide a feedback to the user on their choice of thresholds (stored in the *Property-CE-Threshold* table) for different properties. For more frequently violated properties, the threshold might be set to a lower probability to trigger the pre-computation earlier.

5.6 Implementation

In this section, we detail the LeaPS implementation and its integration into OpenStack along with the architecture of LeaPS (Figure 5.13) and a detailed algorithm.

5.6.1 Background

OpenStack [89] is an open-source cloud management platform that is being used almost in half of private clouds and significant portions of the public clouds (see [92] for detailed statistics). Neutron is its network component, Nova is its compute component, and Ceilometer is its telemetry for receiving event histories from other components. Each component of OpenStack generates notifications, which are triggered by predefined activities such as VM creation, security group addition, and are sent to Ceilometer for monitoring purposes. Ceilometer extracts the information from the notifications, and transforms them into events.

5.6.2 LeaPS Architecture

Figure 5.13 shows an architecture of LeaPS. It has four main components: log processor, learning system, verification system and dashboard & reporting engine.

- The first component, namely, the log processor, obtains sequences of events from the retrieved raw cloud logs. To this end, the parser module first processes the raw logs to retrieve identified fields in each log entry and systematically generates structured content stored as CSV files. Then, the filter module groups the log entries tenant-wise and separates log entries corresponding to the user initiated events. The interpreter module consults the mapping of URL paths and request body to identify the corresponding event type of a log entry. The merger module combines logs from different services (e.g., Neutron and Nova) of OpenStack. The sequence builder generates the sequences of events from the logs.
- The second component, namely, the learning system, is responsible for learning the probabilistic dependencies using Bayesian network from the output of the log processing component. To this end, the appropriate input formats of the learning engine are obtained from the log processor. Then, the learning engine, which is a Bayesian network learning tool, learns the probabilistic dependencies from the sequences of events.
- The third component, namely, the proactive verification system, incrementally prepares for the verification and verifies the preconditions of the security critical events that are about to occur. To this end, the likelihood evaluator consists of three modules. The interceptor intercepts runtime event instances, the event matcher obtains the event type of the intercepted event instances, and the critical event identifier detects the critical events from the intercepted event type. Triggered by the likelihood evaluator, the pre-computation manager is to initialize (by the initializer) and update (by the updater) watchlists. LeaPS leverages a proactive

verification tool [71] to perform the runtime verification utilizing the pre-computed results. The feedback module is to analyze the previous verification results and to provide feedback to update the probabilities in the model.

- The fourth component, namely, the dashboard & reporting engine, is to provide an interface to LeaPS users to interact with the system and to observe different verification results.

In the following, we describe the implementation details of different components of LeaPS.

5.6.3 Log Processor

The log processor first automatically collects logs from different OpenStack components, e.g., Nova, Neutron, Ceilometer, etc. We use Logstash [27], a popular open-source data processing tool, for transforming un-structured and semi-structured logs into CSV format and available for further processing. To enable Logstash transformation, we use the parsing rules that we build for OpenStack logs in our case study. Afterwards, we implement filters in Python to group and eliminate log entries. Then, we build a mapping between URL paths with request body and event types, and consult this map to identify event type of each log entry. Next, we merge Neutron and Nova logs based on the timestamps while handling conflicting issues. For example, while a user requests to create a VM, the event (i.e., create port) happening at Neutron is done by the `tenant-service`, and is removed while dividing events into different tenant groups. Finally, to prepare the logs to be used in the LeaPS learning system and for other log analysis purposes, we run a custom algorithm, which preserves all transitions in the actual logs, implemented in Python to identify sequences in combined logs. The *processLogs* procedure in Algorithm 6 implements all above-mentioned steps of our log processor.

5.6.4 Learning System

For learning, we leverage SMILE & GeNIe [9], which is a popular tool for modeling and learning with Bayesian network. SMILE & GeNIe uses the EM algorithm [22, 63] for parameter learning. The learning module is responsible for preparing inputs to GeNIe, and conducting the learning process using GeNIe. The sequences obtained from the log processor are further

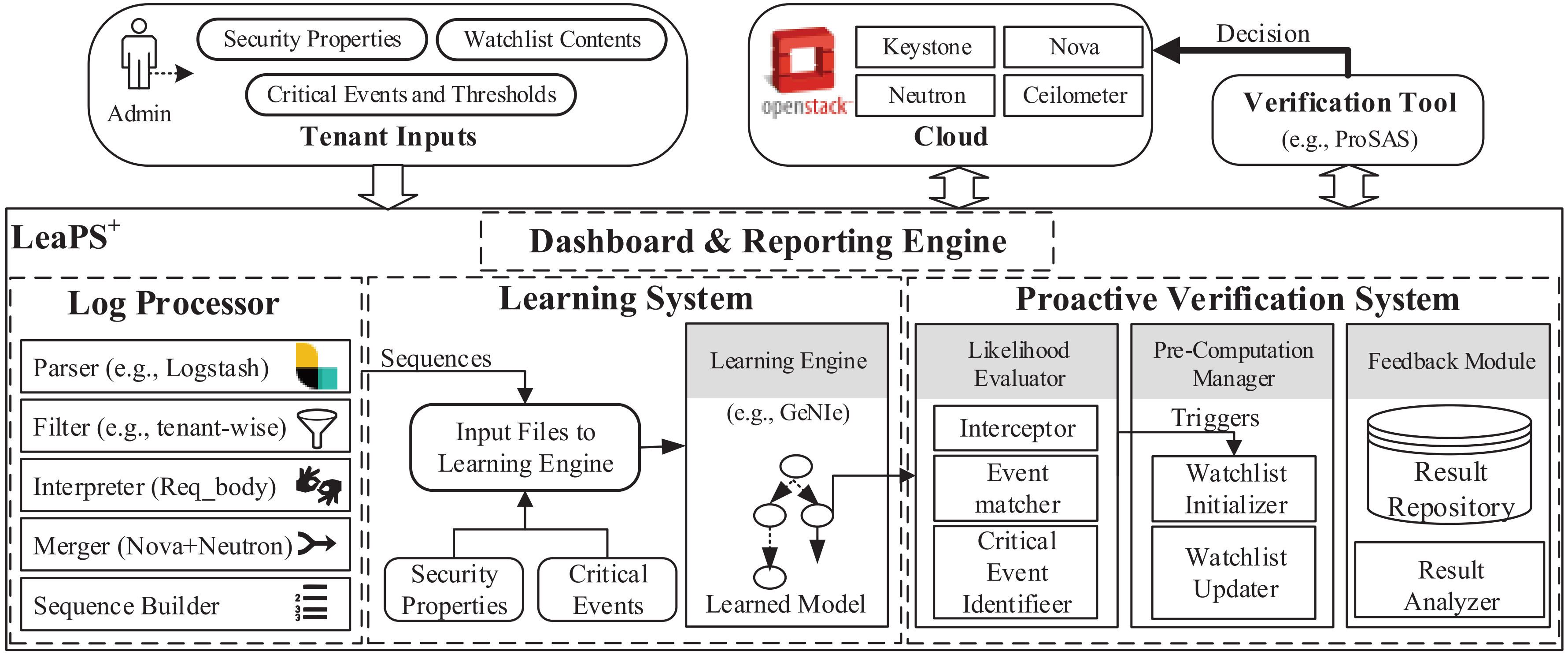


Figure 5.13: An architecture of LeaPS auditing system.

Algorithm 6 Log Processing ()

```
Input: $CloudOS$, $parsing\text{-}rules$, $matching\text{-}rules$
Output: $sequence[]$
 1: procedure ProcessLogs($CloudOS$)
 2:     for each component $c_i \in CloudOS$ do
 3:         $rawLogs$ = collectLog($c_i$)
 4:         $Fields[]$ = identifyLogFields($rawLogs$, $parsing\text{-}rules$)
 5:         $parsedLogs[c_i]$ = parseLogs($rawLogs$, $Fields[]$, $parsing\text{-}rules$)
 6:         $systemEvents[]$ = identifySystemEvents($CloudOS$)
 7:         $prunedLogs[c_i]$ = pruneLogs($parsedLogs[c_i]$, $systemEvents[]$)
 8:         $groupedLogs[c_i]$ = groupLogs($prunedLogs[c_i]$, $CloudOS.tenants[]$)
 9:         $markedEvents[c_i]$ = markEvents($groupedLogs[c_i]$,$matching\text{-}rules$)
10:     $combinedLogs$ = combineLogs($markedEvents[]$)
11:     for each log entry $entry_i \in combinedLogs$ do
12:         if $entry_i$ is not already in $sequence[j]$ then
13:             $sequence[j] = sequence[j] + entry_i$
14:         else
15:             $j$++;
16:             $sequence[j] = sequence[j] + entry_i$
17:     return $sequence[]$
```

processed to convert them into the input format (in .dat) of GeNIe. Additionally, the structure of the Bayesian network and its parameters are provided to GeNIe. Furthermore, we choose the uniform option, where the assumption is that all parameters in the network form the uniform distribution with confidence is equal to one. Finally, GeNIe provides an estimation of the parameters, which are basically probabilities of different transitions in the dependency model. Additionally, to execute sequence pattern mining algorithms with log processor outputs, we leverage SPMF [33], which is a popular open-source data mining library. The *Learn* procedure in Algorithm 7 implements the learning steps of LeaPS.

5.6.5 Proactive Verification System

We intercept all requests to the Nova service as they are passed through the Nova pipeline, having the LeaPS 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. Next, the pre-computing module stores the result of inspection in a MySQL database. The feedback module is implemented in Python. Those modules work together to support the methodology described in Section 5.5, as detailed in Algorithm 7.

Algorithm 7 Learning-Based Proactive Verification ()

```
Input: $CloudOS$, $Properties$, $structure$, $sequence[]$
Output: $decision$
 1: procedure LEARN($sequence[]$, $structure$, $Properties$)
 2:     for each property $p_i \in Properties$ do
 3:         $learnedParameters$ = learnModel($structure$, $sequence[]$)
 4:         $dependencyModel$ = buildModel($structure$, $learnedParameters$, $p_i.critical\text{-}events$)
        return $dependencyModels$
    ------------------------------------------------
 5: procedure EVALUATE-LIKELIHOOD($CloudOS$, $Properties$, $dependencyModels$)
 6:     for each event type $e_i \in CloudOS.event$ do
 7:         $Conditional\text{-}Probability\text{-}Table$ = inferLikelihood($e_i$, $dependencyModels$)
 8:         if checkThreshold($Conditional\text{-}Probability\text{-}Table$,$Property\text{-}CE\text{-}Threshold$) then
 9:             insertProperty-PE($e_i$,$Property\text{-}CE\text{-}Threshold.property$)
10:     $interceptedEvent$ = intercept-and-match($CloudOS$, $Event\text{-}Operation$)
11:     if $interceptedEvent \in Properties.critical\text{-}events$ then
12:         $decision$ = verifyWL($Properties.WL$, $interceptedEvent.params$)
13:         return $decision$
14:     else if $interceptedEvent \in Property\text{-}PE$ then
15:         Pre-compute-update($Properties$, $interceptedEvent.params$)
    ------------------------------------------------
16: procedure PRE-COMPUTE-INITIALIZE($CloudOS$, $Properties$)
17:     for each property $p_i \in Properties$ do
18:         $WL_i$= initializeWatchlist($p_i.WL$, $CloudOS$)
19: procedure PRE-COMPUTE-UPDATE($Properties$, $parameters$)
20:     updateWatchlist($Properties.WL$, $parameters$)
    ------------------------------------------------
21: procedure FEEDBACK($Result$, $dependencyModels$, $Properties$)
22:     storeResults($Result$, $dependencyModels$)
23:     if analyzeSequence($Result.seq$) = "new-trend" then
24:         updateModel($Result.seq$,'new')
25:     else
26:         updateModel($Result.seq$,'old')
27:     for each property $p_i \in Properties$ do
28:         $change\text{-}in\text{-}threshold[i]$ = analyzeDecision($Result.decision$, $p_i$)
```

5.6.6 Dashboard & Reporting Engine

LeaPS users interact with the system through a dashboard, which is implemented using a web interface in PHP. Through this dashboard, users can enable proactive auditing so that LeaPS starts intercepting cloud events and verify them. In the dashboard, tenant admins can initially select security properties from different standards (e.g., ISO 27017, CCM V3.0.1, NIST 800-53, etc.). Through the monitoring panel, LeaPS continuously updates the summary of the verification results. Furthermore, the details of any violation with a list of evidence are also provided. Moreover, our reporting engine archives all the verification reports for a pre-defined period.

5.7 Experimental Results

In this section, we first describe the experiment settings, and then present LeaPS experimental results with both synthetic and real data.

5.7.1 Experimental Settings

Both experiments on LeaPS log processor and proactive verification system involve datasets collected from our testbed and the real cloud. In the following, we describe both environmental settings.

Testbed Cloud Settings. Our testbed cloud is based on OpenStack version Mitaka. There are one controller node and up to 80 compute nodes, each having Intel i7 dual core CPU and 2GB memory running Ubuntu 16.04 server. Based on a recent survey [92] on OpenStack, we simulate an environment with maximum 1,000 tenants and 100,000 VMs. We conduct the experiment for 10 different datasets varying the number of tenants from 100 to 1,000 while keeping the number of VMs fixed to 1,000 per tenant. For Bayesian network learning, we use GeNIe academic version 2.1. For sequential pattern mining, we use SPMF v.2.20. Table 5.8 describes the datasets for experiments on log processing. We repeat each experiment 100 times.

Dataset	**Nova**	**Neutron**
DS1	9,997	7,998
DS2	20,000	15,998
DS3	29,998	23,999
DS4	39,998	32,000
DS5	48,995	40,293

Table 5.8: The number of events in both Neutron and Nova logs for different datasets generated in our testbed cloud.

Real Cloud Settings. We further test LeaPS using data collected from a real community cloud hosted at one of the largest telecommunication vendors. To this end, we analyze the management logs (sized 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 is not configured in this cloud, we utilize Nova and Neutron logs which increases the log processing efforts significantly.

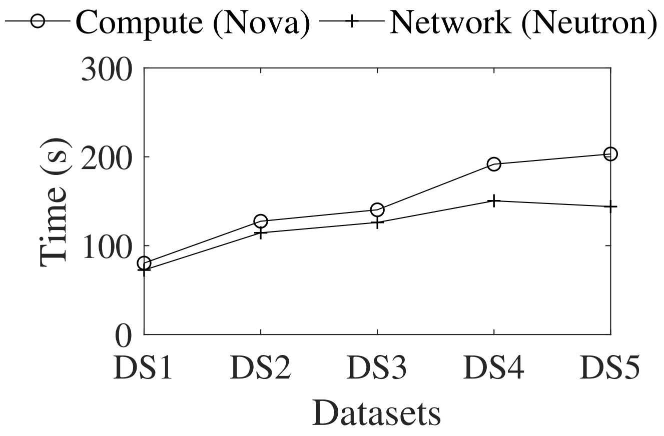


Figure 5.14: Time (in Seconds) required for parsing raw logs while varying the number of events provided in different datasets.

5.7.2 Results on Log Processor

In the following, we present obtained experiment results for our log processor both in testbed and real clouds.

Experiments with Testbed Cloud. The objective of the first set of the experiments is to measure the efficiency of our log processor for two different cloud services, e.g., compute (Nova) and network (Neutron). Figure 5.14 shows the time required (in seconds) to parse logs of different datasets. The results show that the parsing is the most time consuming step in log processing, as this step parses text-based logs and stores them into CSV files. The parsing of our largest dataset (DS5) requires around 3 minutes and around 2 minutes for Nova and Neutron logs, respectively. Figure 5.16 shows the time required (in seconds) to group the log entries based on tenant IDs and to eliminate system-initiated entries (from `tenant-service`). For the largest dataset of Nova, the required time remains within 80 milliseconds. Grouping Neutron

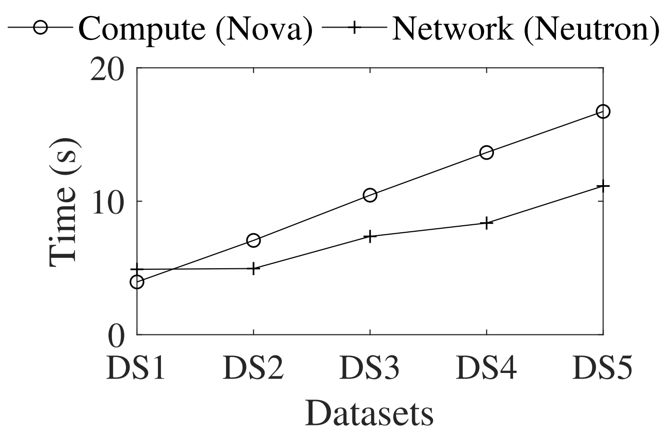


Figure 5.15: Time (in Seconds) required for interpreting event types while varying the number of events provided in different datasets.

logs, which is comparatively smaller in size, requires maximum 55 milliseconds. Figure 5.15 presents the results (in seconds) to interpret event types of all entries from the grouped logs for Nova and Neutron. The trend for both services shows almost a linear increase while varying the number of log entries. Interpreting event types for the largest dataset takes 16.72 seconds and 11.14 seconds for Nova and Neutron, respectively.

The second set of experiments is to measure the efficiency of aggregating logs from different services and generating outputs by our log processor. Figure 5.17 shows the time in seconds to aggregate logs from Nova and Neutron, and to generate inputs to the sequence pattern mining algorithms implemented in the SPMF library [33]. The time to aggregate logs remains within 112 milliseconds for the largest dataset. The time for the input generation remains within two milliseconds for the largest dataset and shows a linear increase. Figure 5.18 depicts the time required for the steps, i.e., eliminating and counting repeated entries, identifying sequences as the outputs of the log processor, performed on the aggregated logs. In both steps, the larger datasets show less increase in the required time. The time required for identifying sequences remains within 2.6 seconds, and the eliminating repeated entries step takes maximum 4.2 seconds for our largest dataset.

Applying Alternative Learning Techniques. To demonstrate the applicability of our log processor, we further apply its outputs to run three popular sequence pattern mining (which is a

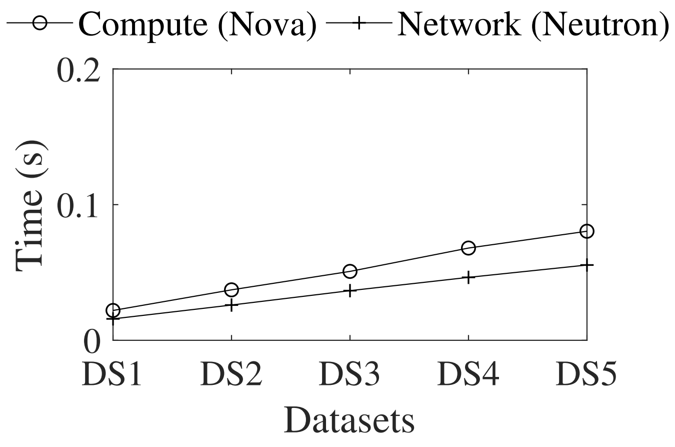


Figure 5.16: Time (in Seconds) required for grouping log entries based on tenant IDs while varying the number of events provided in different datasets.

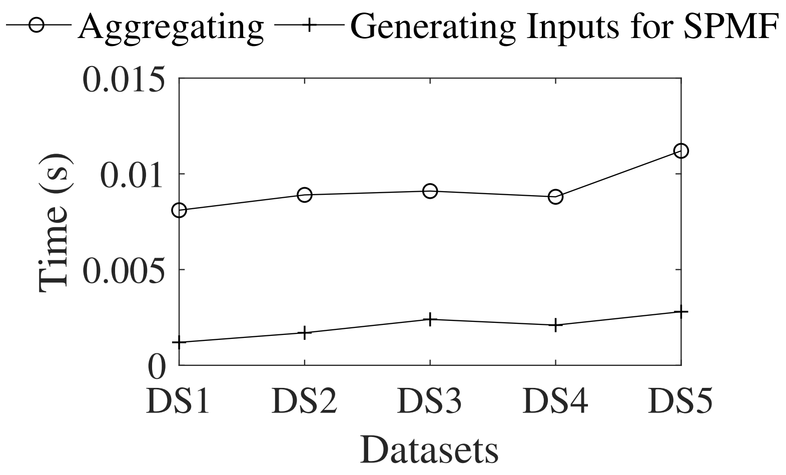


Figure 5.17: Time required (in seconds) for merging nova_api and neutron_server logs, and generating inputs for the pattern mining library (SPMF).

data mining approach with broad range of applications) algorithms. Specifically, we first run the PrefixSpan [95] algorithm, which mines the complete set of patterns. One potential application could be to identify structures of dependencies among cloud events using this algorithm to further enhance the proactive security solutions. Second, we execute the MaxSP [34] algorithm, which mines maximal sequential patterns. Using this algorithm, we can easily identify the most common patterns, which potentially can facilitate anomaly-based security solutions. Third, we run the ClaSP [37] algorithm, which identifies the largest pattern with a minimum frequency. This algorithm might be useful to identify unique patterns to profile a security violation or a

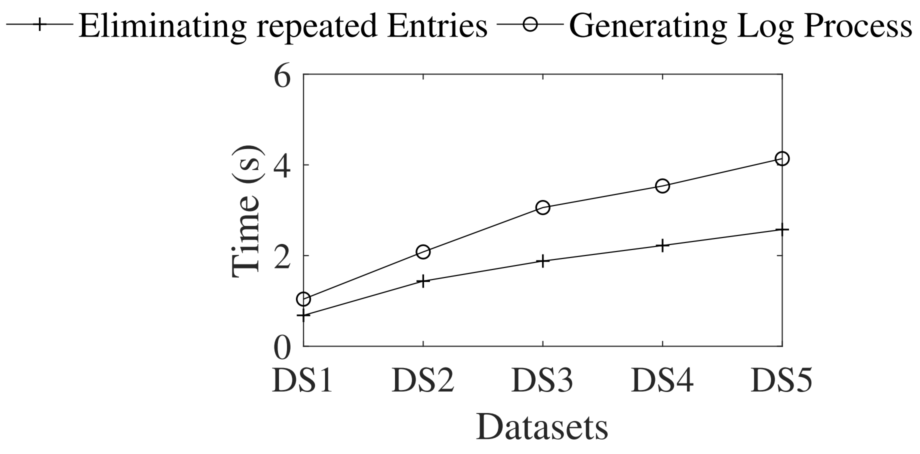


Figure 5.18: Time required (in seconds) for eliminating repeated entries and generating log processor output.

legitimate use. To this end, we generate inputs to SPMF, which is a sequence pattern mining tool, and report the input generation time in Figure 5.18. We also report the efficiency results to run these algorithms with the outputs of our log processor in Figure 5.19. We observe constant time (one millisecond) while running ClaSP algorithm for different datasets. The MaxSP and PrefixSpan algorithms take around 18 milliseconds and 6.5 milliseconds, respectively, for our largest dataset.

Experiments with Real Cloud. The main objective of this part of the experiments is to evaluate the applicability of our log processor in a real cloud environment. Table 5.9 shows the summary of the results that we obtain for the real data. Due to the much larger size (e.g., 1.6 GB text-based logs) of the real-life logs, the parsing time is quite long (4 hours and 40 minutes). However, once LeaPS is active, it may potentially log intercepted events in an incremented manner to avoid the delays at the parsing step. After parsing, we eliminate the log entries related to listing resources and their details, as the corresponding events to these entries are beyond our interest. The time for the remaining steps is quite similar to what is measured for our testbed cloud logs with much smaller size of logs. Note that the grouping step is not measured for Neutron, as the tenant ID is missing in the Neutron logs collected from the real cloud (as discussed in Section 5.3.1). However, we group them arbitrarily to measure the time for next steps. From the results of the real data, our observation is that our log processor is scalable once the parsing step

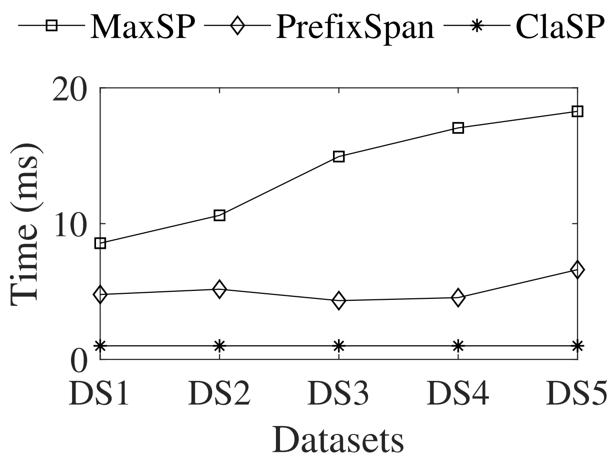


Figure 5.19: Time required (in milliseconds) for running PrefixSpan, MaxSP and ClaSP algorithms in the SPMF library.

is performed; which possibly allows our approach to process huge logs in a reasonable time.

Services	**# of Log Entries**	**Parsing**	**Grouping**	**Interpretation**	**Merging**	**Generating Sequences**
Nova	1,450,011	4h 40m	0.0777s	99.271s	0.02206s	1.4483s
Neutron	3,992,644		-	51.820s		

Table 5.9: Summary of the experimental results with real data for Nova and Neutron services. The *parsing*, *merging* and *generating sequences* steps are performed together for both services. Note that the grouping step is not measured for Neutron, as the tenant ID is missing in the Neutron logs collected from the real cloud.

5.7.3 Results on Proactive Verification System

In the following, we discuss the obtained experimental results for our proactive verification system both in testbed and real clouds.

Experiments with Testbed Cloud. The objective of the first set of experiments with our proactive verification system is to demonstrate the time efficiency. Figure 5.20 shows the time in milliseconds required by LeaPS to verify the *no bypass of security group* [18] and *no cross-tenant port* [53] properties. Our experiment shows the time for both properties remains almost the same for different datasets, because most operations during this step are database queries; SQL queries for our different datasets almost take the same time. Figure 5.21 shows the time (in seconds) required by GeNIe to learn the model while we vary the number of events from

2,000 to 10,000. In Figure 5.22, we measure the time required for different steps of the offline pre-computing for the *no bypass* property. The total time (including the time of incrementally updating WL and updating PE) required for the largest dataset is about eight seconds which justifies performing the pre-computation proactively. A one-time initialization of pre-computation is performed in 50 seconds for the largest dataset. Figure 5.23 shows the time in seconds required to update the model and to update the list of pre-compute events. In total, LeaPS requires less than 3.5 seconds for this step.

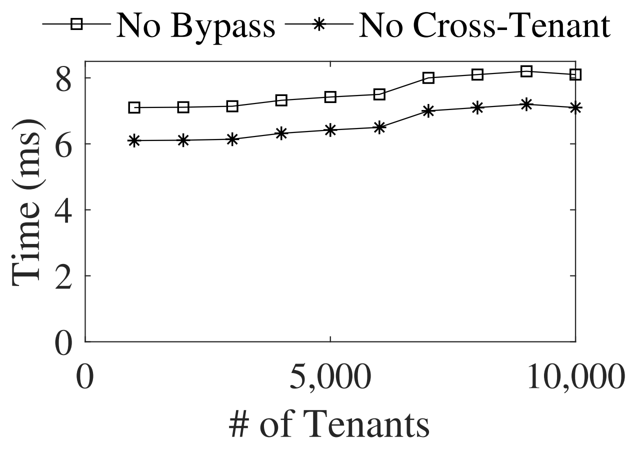


Figure 5.20: Showing time required for the online runtime verification by varying the number of VMs for the *no bypass* and *no cross-tenant* properties. The verification time includes the time to perform interception, matching of event type and checking in the watchlist.

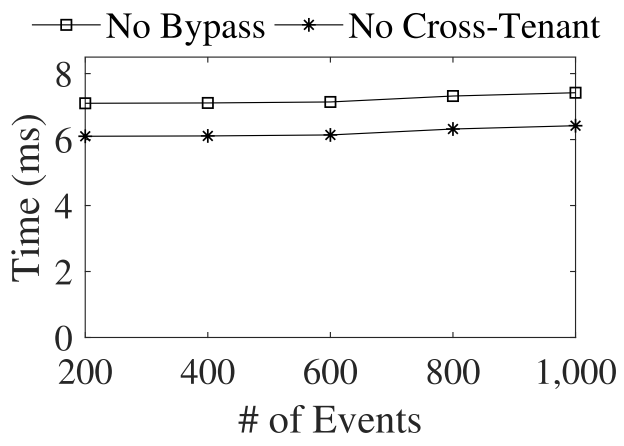


Figure 5.21: Showing time required for the offline learning process by varying the number of event instances in the logs for the *no bypass* and *no cross-tenant* properties.

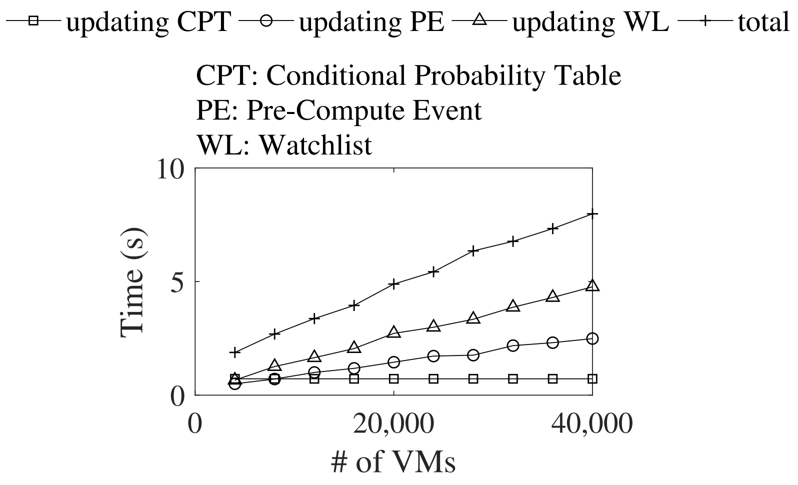


Figure 5.22: Showing time required in seconds for the pre-computation considering the *no bypass* property by varying the number of instances.

In the second set of experiments, we demonstrate how much LeaPS may be affected by a wrong prediction resulted from inaccurate learning. For this experiment, we simulate different prediction error rates (PER) of a learning engine ranging from 0 to 0.4 on the likelihood evaluator procedure in Algorithm 7. Figure 5.24 shows in seconds the additional delay in the pre-computation caused by the different PER of a learning engine for three different number of VMs. Note that, the pre-computation in LeaPS is an offline step. The delay caused by 40% PER for up to 100k VMs remains under two seconds, which is still acceptable for most applications.

In the final set of experiments, we compare LeaPS with a baseline approach (similar to [71]), where all possible paths are considered with equal weight, and the number of steps in the model is the deciding factor for the pre-computation. Figure 5.25 shows the pre-computation time for both approaches in the average case, and LeaPS performs about 50% faster than the baseline approach (the main reason is that, in contrast to the baseline, LeaPS avoids the pre-computation for half of the critical events on average by leveraging the probabilistic dependency model). For this experiment, we choose the threshold, *N-th* (an input to the baseline), as two, and the number of security properties as four. Increasing both the value of *N-th* and the number of properties increase the pre-computation overhead for the baseline. Note that a longer pre-computation time eventually affects the response time of a proactive auditing.

Experiments with Real Cloud. Table 5.10 summarizes the obtained results. We first measure

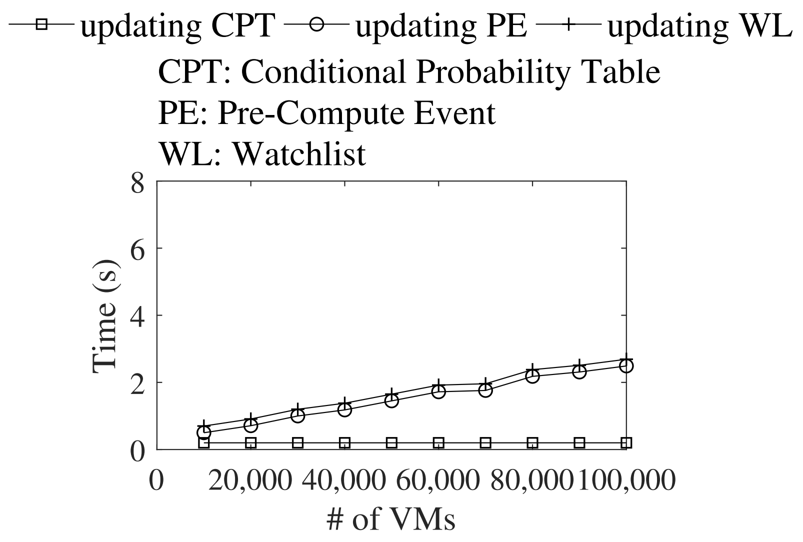


Figure 5.23: Showing time required in seconds for the feedback modules considering the *no bypass* property by varying the number of instances.

the time efficiency of LeaPS. Note that the results obtained are shorter due to the smaller size of the community cloud compared to our much larger simulated environment. Furthermore, we measure the prediction error rate (PER) of the learning tool using another dataset (for 5 days) of this cloud. For the 3.4% of PER, LeaPS affects maximum 9.62 milliseconds additional delay in its pre-computation for the measured properties.

Properties	**Learning**	**Pre-Compute**	**Feedback**	**Verification**	**PER**	**Delay***
No bypass	7.2s	424ms	327ms	5.2ms	0.034	9.62ms
No cross-tenant	5.97s	419ms	315ms	5ms	0.034	9.513ms

Table 5.10: Summary of the experimental results with real data. The reported delay is in the pre-computation of LeaPS due to the prediction error (PER) of the learning engine.

5.8 Discussions

Adapting to Other Cloud Platforms. LeaPS is designed to work with most popular cloud platforms (e.g., OpenStack [89], Amazon EC2 [5], Google GCP [38] and Microsoft Azure [77]) with a one-time effort for implementing a platform-specific interface. More specifically, LeaPS interacts with the cloud platform (e.g., while collecting logs and intercepting runtime events)

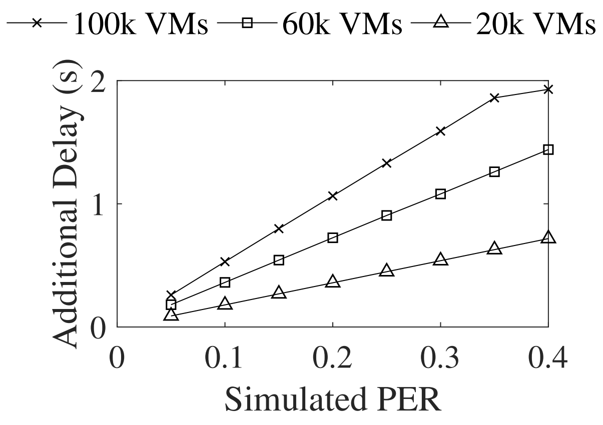


Figure 5.24: The additional delay (in seconds) in LeaPS pre-computation time caused by different simulated prediction error rates (PER) of a learning tool.

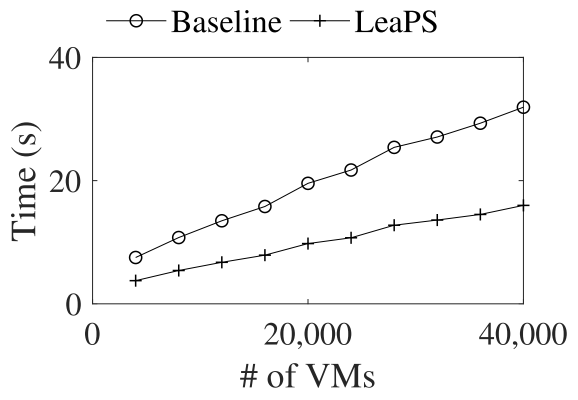


Figure 5.25: the comparison (in seconds) between LeaPS and a baseline approach.

through two modules: log processor and interceptor. These two modules require to interpret implementation specific event instances and intercept runtime events. First, to interpret platform-specific event instances to generic event types, we currently maintain a mapping of the APIs from different platforms. Table 5.11 enlists some examples of such mappings. Second, the interception mechanism may require to be implemented for each cloud platform. In OpenStack, we leverage WSGI middleware to intercept and enforce the proactive auditing results so that compliance can be preserved. Through our preliminary study, we identify 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 LeaPS 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 event managing service from Azure to monitor and control event routing which is quite similar as our interception mechanism. Therefore, we believe that LeaPS can be an extension of the Azure Event Grid to proactively audit cloud events. Table 5.12 summarizes the interception support in these cloud platforms. The rest modules of LeaPS deal with the platform-independent data, and hence, the next steps in LeaPS are platform-agnostic.

LeaPS Event Type	**OpenStack [89]**	**Amazon EC2-VPC [5]**	**Google GCP [38]**	**Microsoft Azure [77]**
create VM	`POST /servers`	`aws opsworks -region create-instance`	`gcloud compute instances create`	`az vm create l`
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 {name}`	N/A	`az network nsg delete`

Table 5.11: Mapping event APIs from different cloud platforms to LeaPS event types.

Cloud Platform	**Interception Support**
OpenStack	*WSGI Middleware* [113]
Amazon EC2-VPC	*AWS Lambda Function* [5]
Google GCP	*GCP Metrics* [38]
Microsoft Azure	*Azure Event Grid* [77]

Table 5.12: Interception supports in major cloud platforms

Effects of False Positives in the Learning Technique. LeaPS leverages learning techniques in a different manner so that the false positive/negative rates cannot affect the security of our system directly, and rather affects the performance of our system. In LeaPS, learning parameters of Bayesian network is utilized to learn the probabilistic dependencies. Any error in learning results a dependency model with incorrect probabilities. Later, while consulting this dependency model by the pre-computation module (as described in Section 5.5.2), LeaPS may choose wrong highly-likely events and perform unnecessary pre-computation for them. At the same time, LeaPS may delay the pre-computation for actual highly-likely events. The final result of such mistakes in LeaPS due to the false positives/negatives in the learning tool is the increase in the response time (as reported in Figure 5.24).

Possibility of a DoS Attack against LeaPS. To exploit the fact that a wrong prediction may result in a delay in the LeaPS pre-computation, an attacker may conduct a DoS attack to bias the learning model step by generating fake events and hence, to exhaust LeaPS pre-computation. However, Figure 5.24 shows that an attacker requires to inject a significantly large amount (e.g., 40% error rate) of biased event instances to cause a delay of two seconds. Moreover, biasing the model is non-trivial unless the attacker has prior knowledge of patterns of legitimate event sequences. Our future work will further investigate this possibility and its countermeasures.

Granularity of Learning. The above-mentioned learning can be performed at different levels (e.g., cloud level, tenant level and user level). The cloud level learning captures business nature only for companies using a private cloud. The tenant level learning depicts a better profile of each business or tenant. This level of learning is mainly suitable for companies strictly following process management, where users mainly follow the steps of processes. In contrast, the user level learning is suitable for smaller organizations (where no process management is followed) with fewer users (e.g., admins) who perform cloud events. Conversely, if a company follows process management, user level learning will be less useful, as different users would exhibit very similar patterns.

Dependency on Critical Event Lists. The list of critical events provided by LeaPS users for each security property is very critical for LeaPS to accurately prevent any security violation. Any incompleteness in this list may result in violations undetected by LeaPS. Therefore, we provide a guideline on identifying different LeaPS inputs including lists of critical events. The steps to identify sets of event types as the inputs to the learning engine are as follows: i) from the property definition, we identify involved cloud components; ii) we enlist all event types in a cloud platform involving those components; and iii) we identify the critical events (which is already provided by the tenant) from the list, and further shortlist the event types based on the attack scenario. The specification of watchlist is a LeaPS input from the tenant. The specification of watchlist can be decided as follows: i) from the property definition, the asset to keep safe is identified; ii) the objectives of a security property are to be highlighted; and iii) from the attack scenario, the parameters for the watchlist for each critical event is finalized.

Tackling Single-Step Violation. 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 properties 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 5.20, and may be detected by performing all steps at a single point in several seconds (e.g., around six seconds for a decent-sized cloud with 60,000 VMs as shown in Figure 5.22); which is still faster than any other existing works (which respond 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 LeaPS even for a single-step violation. Additionally, During our case studies, we observed that OpenStack performs several internal tasks to complete a user request. We may leverage this sequence of system events corresponding to a single user request to proactively perform the LeaPS steps. We elaborate those two ways of tackling single-step violations in our future work.

The Concept of Proactive Security Auditing for Clouds. The concept of proactive security auditing for clouds is different than the traditional security auditing concept. Apart from ours, proactive security auditing for clouds is also proposed in [16]. Additionally, the Cloud Security Alliance (CSA) recommends continuous auditing as the highest level of auditing [19], from which our work is inspired. The current proactive and runtime auditing mechanisms are more of a combination of traditional auditing and incident management. For example, in LeaPS, we learn from incidents and intercepted events to process or detect in a similar manner as a traditional incident management system. At the same time, LeaPS verifies and enforces compliance against different security properties, which are mostly taken from different security standards, and provide detailed evidence for any violation through the LeaPS dashboard. Therefore, the concept of proactive security auditing is a combination of incident management and security auditing.

5.9 Related Work

Table 5.13 summarizes the comparison between existing works and LeaPS. 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 main objective (auditing or anomaly detection). 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 offers an automated dependency learning, we check the automatic feature. We mark this feature as 'N/A' for the works, which do not involve any dependencies. The dynamic feature refers to the dynamic and runtime pattern capturing. The probabilistic feature is marked when a work involves non-deterministic or probabilistic dependencies. For non-dependency-model-based works, we put 'N/A'. The expressive feature is checked for the works, which utilize well-known expressive policy languages (e.g., first order logic) to express security properties. By the self-reliant feature, we mean the works, which only depend on the user-provided security properties for the accuracy of the verification. In the last four columns of the table, we compare the works based on their supporting cloud platforms. The adaptable field is checked for those works, which support multiple cloud platforms or describe how their works can be ported to other platforms.

In summary, LeaPS mainly differs from the state-of-the-art works as follows. Firstly, LeaPS is the first proactive auditing approach, which captures the dependency automatically from the patterns of event sequences. Secondly, LeaPS is the only learning-based work, which aims at improving proactive auditing and not (directly) at anomaly detection. Thirdly, the dynamic dependency model allows LeaPS to evolve over time to adapt to new trends. Finally, the LeaPS methodology is cloud-platform agnostic. However, there are still few limitations in LeaPS. LeaPS is less expressive than other general purpose formal verification approaches. LeaPS relies on a complete list of critical events provided by tenant admins or security experts to provide 100% accuracy.

Retroactive and Intercept-and-Check Auditing. Retroactive auditing approach (e.g., [70, 73, 110, 112, 108, 25] in the cloud is a traditional way to verify the compliance of different components of a cloud. Unlike our proposal, those approaches can detect violations only after they occur, which may expose the system to high risks. Existing intercept-and-check approaches (e.g., [16, 88]) perform major verification tasks while holding the event instances blocked, and usually cause significant delay to a user request. Unlike those works, LeaPS provides a proactive auditing approach.

Proactive Auditing. Proactive security analysis in the cloud is comparatively a new domain with fewer works (e.g., [16, 71, 114]). Weatherman [16] verifies security policies on a future change plan in a virtualized infrastructure using the graph-based model proposed in [14, 13]. PVSC [71] proactively verifies security compliance by utilizing the static patterns in dependency models. Both in Weatherman and PVSC, models are captured manually by expert knowledge. In contrast, this work adopts a learning-based approach to automatically derive the dependency model. Congress [88] is an OpenStack project offering similar features as Weatherman. Foley et al. [31] propose an algebra for anomaly-free firewall policies for OpenStack. Many state-based formal models (e.g., [103, 65, 66, 26]) are proposed for program monitoring. Our work further expands the proactive monitoring approach to cloud differing in scope and methodology.

Learning-Based Detections. There are many learning-based security solutions (e.g., [104, 43, 45, 55, 78, 46, 75]), which offer anomaly detection. Unlike above-mentioned solutions, this work proposes a totally different learning-based technique to facilitate the proactive auditing.

Proposals	Methods	Coverage		Features						Supporting Platforms			
		Environment	Objective	Proactive	Automatic	Dynamic	Probabilistic	Expressive	Self-Reliant	OpenStack	Azure	VMware	Adaptable
Doelitzscher et al. [25]	Custom Algorithm	Cloud	Auditing	-	N/A	●	N/A	-	●	●	-	-	●
Ullah et al. [108]	Custom Algorithm	Cloud	Auditing	-	N/A	-	N/A	-	●	●	-	-	-
Majumdar et al. [73]	CSP Solver	Cloud	Auditing	-	N/A	-	N/A	●	●	●	-	-	-
Madi et al. [70]	CSP Solver	Cloud	Auditing	-	N/A	-	N/A	●	●	●	-	-	-
Jiang et al. [55]	Regression Technique	Non-cloud	Anomaly Det.	●	●	-	●	-	●	N/A	N/A	N/A	N/A
Solanas et al. [104]	Classifiers	Cloud	Anomaly Det.	-	●	-	●	-	●	●	-	-	-
Ligatti et al. [66]	Model Checking	Non-Cloud	Auditing	●	N/A	●	-	●	●	N/A	N/A	N/A	N/A
PVSC [71]	Custom Algorithm	Cloud	Auditing	●	-	-	-	-	-	●	-	-	-
Weatherman [16]	Graph-theoretic	Cloud	Auditing	●	-	●	-	-	-	-	-	●	-
Congress [88]	Datalog	Cloud	Auditing	●	-	-	-	●	-	●	-	-	-
ProSAS	Custom + Bayesian	Cloud	Auditing	●	●	●	●	-	-	●	-	-	●

Table 5.13: Comparing existing solutions with LeaPS. The symbols (●), (-) and N/A mean supported, not supported and not applicable, respectively.

Log Processing in Clouds. There exist several works (e.g., [67, 115, 76, 4, 39]) on log processing in clouds. Lin et al. [67] leverage the big data analytics, Hadoop, and in-memory computing capacity of Spark, to propose a cloud platform which can efficiently process and analyze logs in batches. Similarly, Yu et al. [115] leverage cloud computing and distributed big data analytics to process large amounts of logs. Unlike those works, we focus more on processing logs retrieved from OpenStack clouds. However, leveraging such big data analytics and memory-efficient methods may enhance the performance of our log processing. Sahara [76] offers a real-time log analysis using Spark. Right after each log entry is created, it is fed into Sahara, parsed into separate fields, stored and visualized at an HTTP endpoint. Although Sahara's methodology on real-time analysis can inspire our future attempt on this matter, Sahara currently does not offer next stages (e.g., identifying event types and their sequences) of our log processing. Additionally, Amazon CloudWatch [4] and Google Cloud Dataflow [39] perform advanced real-time analysis provided for troubleshooting of systems by other existing technologies. Unlike those works, we focus more on identifying event types and their sequences in logs to facilitate various learning techniques for analysis.

5.10 Conclusion

In this work, we proposed an automatic learning-based proactive security auditing system, *LeaPS*, which completes a mandatory pre-requisite step (e.g., log processing) and addresses the limitations of existing proactive solutions (by automating the dependency learning). To this end, we first conducted a case study on real-world cloud logs and highlighted the challenges in processing such logs. Then, we designed and implemented a log processing approach for OpenStack to feed its outputs to the learning tools to capture dependencies. Afterwards, we leveraged learning techniques (e.g., Bayesian network) to learn probabilistic dependencies for the dependency model. Finally, using such dependency models, we perform proactive security auditing. Our proposed solution is integrated to OpenStack, a popular cloud management platform. The results of our extensive experiments with both real and synthetic data show that LeaPS can be very scalable for a decent-size cloud (e.g., 6 milliseconds to audit a cloud of

100,000 VMs) and a significant improvement (e.g., about 50% faster) over existing proactive approaches. In addition, we demonstrated that other learning techniques such as sequence pattern mining algorithms can be executed on the outputs of our log processor efficiently (e.g., 18 milliseconds to find frequencies of all possible patterns using PrefixSpan). As future work, we will investigate the feasibility of applying runtime data streaming to process logs incrementally and in a more scalable manner. We also intend to conduct case studies on logging of other cloud platforms to offer a platform-agnostic log processing solution, which might be very useful for LeaPS-like security solutions.

Chapter 6

Conclusion

The ever-changing and self-service nature of clouds brings the necessity to audit the cloud at runtime to ensure continuous security compliance, which is essential for cloud provider's accountability and transparency towards their tenants. To this end, there exist two types of cloud-specific security auditing approaches: intercept-and-check and proactive. However, existing works under these approaches either fail to provide a practical response time due to the sheer scale of the cloud, or require a future change plan in advance which may be impractical in most cloud environments. In this thesis, we addressed these major limitations of the existing runtime auditing solutions and proposed a proactive security auditing system for clouds. To this end, we first proposed a runtime security auditing system for the user level of the cloud; which verifies different authentication and authorization mechanisms (e.g., RBAC, ABAC and SSO) incrementally. Second, to reduce the response time of runtime auditing, this thesis delivered a proactive security auditing system; which offers a novel proactive auditing approach, where we leverage the dependency relationships among cloud events to incrementally pre-compute the major auditing efforts. As a result, the runtime effort in our proactive approach remains lightweight and results in a practical response time (e.g., 8.5 milliseconds for 100,000 virtual ports). Third, to extend the coverage of our auditing system and improve its practicality, we leveraged learning techniques to design and implement an automated approach to process raw cloud logs and utilize them to capture various dependency relationships.

However, our work still has a few limitations, which can be addressed in future works. First, our auditing approach is signature-based and hence, cannot detect any zero-day security violations. Potentially, an anomaly-based or hybrid (i.e., combining signature and anomaly based) auditing approach may improve the coverage of our solution; which we consider as a future work. Second, the efficiency of the current approach may be affected when multiple user requests appear simultaneously. A parallel or distributed approach might reduce the effect of this situation; which we consider as a potential future work. Third, our current proactive solution cannot efficiently handle single-step violations. An efficient runtime approach might help to address this concern. Fourth, our watchlist content identification process is currently static and manual; which may affect the accuracy of our solution. Adapting similar approach as our feedback module in Section 4.3.5 may progressively and semi-automatically update the content of watchlists; which will be covered in the upcoming work. Fifth, this work cannot prevent potential privacy concerns from the audit results. We target this problem in our future work. As a future work, we will also investigate the feasibility of applying runtime data streaming to process logs incrementally and in a more scalable manner. We also intend to conduct case studies on logging of other cloud platforms to offer a platform-agnostic log processing solution, which might be useful for security solutions, which are involved with cloud logs.

In summary, this thesis work significantly contributes towards improving security, efficiency and automation of cloud security auditing. Our hope is that this work can show a path to future research as follows. Our proposed proactive approach may be adopted to other security solutions (especially those which involve high computational overhead) to enable incremental pre-computation and reduce the response time. Also, our idea of leveraging learning techniques for efficiency, instead of security, may be useful to benefit other security solutions.

Bibliography

[1] E. Aguiar, Y. Zhang, and M. Blanton. An overview of issues and recent developments in cloud computing and storage security. In *High Performance Cloud Auditing and Applications*. Springer, 2014.

[2] G.-J. Ahn, H. Hu, J. Lee, and Y. Meng. Representing and reasoning about web access control policies. In *COMPSAC*, 2010.

[3] Q. Alam, S. U. Malik, A. Akhunzada, K.-K. R. Choo, S. Tabbasum, and M. Alam. A cross tenant access control (CTAC) model for cloud computing: Formal specification and verification. *IEEE TIFS*, 2016.

[4] Amazon. Amazon CloudWatch. Available at: `https://aws.amazon.com/cloudwatch/`, last accessed on: February 15, 2018.

[5] Amazon. Amazon virtual private cloud. Available at: `https://aws.amazon.com/vpc`.

[6] Amazon Web Services. Security at scale: Logging in AWS. Technical report, Amazon, 2013.

[7] K. Arkoudas, R. Chadha, and J. Chiang. Sophisticated access control via SMT and logical frameworks. *ACM TISSEC*, 2014.

[8] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google apps. In *ACM FMSE*, 2008.

[9] BayesFusion. GeNIe and SMILE. Available at: `https://www.bayesfusion.com`, last accessed on: February 14, 2018.

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

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

[12] N. Bjørner and K. Jayaraman. Checking cloud contracts in Microsoft Azure. In *Distributed Computing and Internet Technology*. Springer, 2015.

[13] S. Bleikertz, T. Groß, M. Schunter, and K. Eriksson. Automated information flow analysis of virtualized infrastructures. In *European Symposium on Research in Computer Security (ESORICS)*, pages 392–415. Springer, 2011.

[14] S. Bleikertz, C. Vogel, and T. Groß. Cloud Radar: near real-time detection of security failures in dynamic virtualized infrastructures. In *Proceedings of the 30th annual computer security applications conference (ACSAC)*, pages 26–35. ACM, 2014.

[15] S. Bleikertz, C. Vogel, and T. Groß. Cloud radar: near real-time detection of security failures in dynamic virtualized infrastructures. In *ACSAC*, 2014.

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

[17] Cloud Security Alliance. Security guidance for critical areas of focus in cloud computing v 3.0, 2011.

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

[19] Cloud Security Alliance. CSA STAR program and open certification framework in 2016 and beyond, 2016. Available at: `https://cloudsecurityalliance.org`.

[20] CUMULUS. Certification infrastructure for multi-layer cloud services project (cumulus). *EU project*, 2012.

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

[22] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977.

[23] Distributed Management Task Force, INC. Cloud auditing data federation, 2016. `https://www.dmtf.org/standards/cadf`.

[24] F. Doelitzscher. *Security Audit Compliance for Cloud Computing*. PhD thesis, Plymouth University, 2014.

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

[26] E. Dolzhenko, J. Ligatti, and S. Reddy. Modeling runtime enforcement with mandatory results automata. *International Journal of Information Security*, 14(1):47–60, 2015.

[27] Elasticsearch. Logstash. Available at: `https://www.elastic.co/products/logstash`, last accessed on: February 14, 2018.

[28] ENISA. European union agency for network and information security, 2016. `https://www.enisa.europa.eu`.

[29] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM TISSEC*, 4(3), 2001.

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

[31] S. N. Foley and U. Neville. A firewall algebra for OpenStack. In *Conference on Communications and Network Security (CNS)*, pages 541–549. IEEE, 2015.

[32] L. Foundation. Open vSwitch, 2018. Available at: `https://www.openvswitch.org`.

[33] P. Fournier-Viger. SPMF, an open-source data mining library. Available at: `http://www.philippe-fournier-viger.com/spmf/index.php`, last accessed on: February 14, 2018.

[34] P. Fournier-Viger, C.-W. Wu, and V. S. Tseng. Mining maximal sequential patterns without candidate maintenance. In *International Conference on Advanced Data Mining and Applications*, pages 169–180. Springer, 2013.

[35] getcloudify..org. OpenStack in numbers - the real stats, 2014. Available at: `http://getcloudify.org`.

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

[37] A. Gomariz, M. Campos, R. Marin, and B. Goethals. Clasp: an efficient algorithm for mining frequent closed sequences. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 50–61. Springer, 2013.

[38] Google. Google cloud platform. Available at: `https://cloud.google.com`.

[39] Google. Processing Logs at Scale Using Cloud Dataflow. Available at: `https://cloud.google.com/solutions/processing-logs-at-scale-using-dataflow`, last accessed on: February 15, 2018.

[40] A. Gouglidis and I. Mavridis. domRBAC: An access control model for modern collaborative systems. *Computers & Security*, 2012.

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

[42] T. Groß. Security analysis of the SAML single sign-on browser/artifact profile. In *AC-SAC*, 2003.

[43] S. Guha. *Attack Detection for Cyber Systems and Probabilistic State Estimation in Partially Observable Cyber Environments*. PhD thesis, Arizona State University, 2016.

[44] D. Heckerman. A tutorial on learning with bayesian networks. In *Learning in graphical models*, pages 301–354. Springer, 1998.

[45] R. A. Hemmat and A. Hafid. SLA violation prediction in cloud computing: A machine learning perspective. Technical report, Université de Montréal, 2016.

[46] H. Holm, K. Shahzad, M. Buschle, and M. Ekstedt. P^2 CySeMoL: Predictive, probabilistic cyber security modeling language. *IEEE Transactions on Dependable and Secure Computing*, 12(6):626–639, 2015.

[47] S.-J. Horng, S.-F. Tzeng, Y. Pan, P. Fan, X. Wang, T. Li, and M. K. Khan. b-SPECS+: Batch verification for secure pseudonymous authentication in VANET. *IEEE TIFS*, 2013.

[48] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to attribute based access control (ABAC) definition and considerations. *NIST SP*, 800, 2014.

[49] IBM. Safeguarding the cloud with IBM security solutions. Technical report, IBM Corporation, 2013.

[50] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy. CloudSec: A security monitoring appliance for virtual machines in the IaaS cloud model. In *5th International Conference on Network and System Security (NSS)*, pages 113–120. IEEE, 2011.

[51] Z. Ismail, C. Kiennert, J. Leneutre, and L. Chen. Auditing a cloud provider's compliance with data backup requirements: A game theoretical analysis. *IEEE TIFS*, 2016.

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

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

[54] W. Jansen. Inheritance properties of role hierarchies. In *NISSC*, 1998.

[55] Y. Jiang, E. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao. Exploiting statistical correlations for proactive prediction of program behaviors. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 248–256. ACM, 2010.

[56] X. Jin. Attribute Based Access Control Model. Available at: `https://blueprints.launchpad.net/keystone/%2Bspec/attribute-based-access-control`.

[57] X. Jin. *Attribute Based Access Control and Implementation in Infrastructure as a Service Cloud*. PhD thesis, The University of Texas at San Antonio, 2014.

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

[59] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

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

[61] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *roceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

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

[63] S. L. Lauritzen. The EM algorithm for graphical association models with missing data. *Computational Statistics & Data Analysis*, 19(2):191–201, 1995.

[64] M. Li, W. Zang, K. Bai, M. Yu, and P. Liu. Mycloud: supporting user-configured privacy protection in cloud computing. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, pages 59–68. ACM, 2013.

[65] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):19, 2009.

[66] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *European Symposium on Research in Computer Security (ESORICS*, pages 87–100. Springer, 2010.

[67] X. Lin, P. Wang, and B. Wu. Log analysis in cloud computing environment with hadoop and spark. In *5th IEEE International Conference on Broadband Network & Multimedia Technology (IC-BNMT)*, pages 273–276. IEEE, 2013.

[68] Z. Lu, Z. Wen, Z. Tang, and R. Li. Resolution for conflicts of inter-operation in multi-domain environment. *Wuhan University Journal of Natural Sciences*, 12(5), 2007.

[69] 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 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 2016.

[70] T. Madi, S. Majumdar, Y. Wang, Y. Jarraya, M. Pourzandi, and L. Wang. Auditing security compliance of the virtualized infrastructure in the cloud: Application to OpenStack. In *CODASPY*, 2016.

[71] 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.

[72] S. Majumdar, Y. Jarraya, M. Oqaily, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi. Leaps: Learning-based proactive security auditing for clouds. In *European Symposium on Research in Computer Security (ESORICS)*, pages 265–285. Springer, 2017.

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

[74] 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*, 13(5):1185–1199, 2018.

[75] S. Mehnaz and E. Bertino. Ghostbuster: a fine-grained approach for anomaly detection in file system accesses. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY)*, pages 3–14. ACM, 2017.

[76] M. Michael, R. Chad, M. Pete, and K. Nikita. This is Sparkhara: OpenStack Log Processing in Real-time Using Spark on Sahara. Available at: `https://www.openstack.org/videos/tokyo-2015/this-is-sparkhara-openstack-log-processing-in-real-time-using-spark-on-sahara`, last accessed on: February 15, 2018.

[77] Microsoft. Microsoft Azure virtual network. Available at: `https://azure.microsoft.com`.

[78] R. Mitchell and R. Chen. Behavior rule specification-based intrusion detection for safety critical medical cyber physical systems. *IEEE Transactions on Dependable and Secure Computing*, 12(1):16–30, 2015.

[79] K. Murphy. *A brief introduction to graphical models and Bayesian networks*. Springer, 1998.

[80] S. Narain. Network configuration management via model finding. In *Proceedings of the 19th Conference on Large Installation System Administration Conference (LISA)*, pages 15–15, 2005.

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

[82] OASIS. Security assertion markup language (SAML), 2016. Available at: `http://www.oasis-open.org/committees/security`.

[83] H.-K. Oh and S.-H. Jin. The security limitations of SSO in OpenID. In *10th International Conference on Advanced Communication Technology*, 2008.

[84] Open Data Center Alliance. Open data center alliance usage: Cloud based identity governance and auditing rev. 1.0. Technical report, 2012.

[85] OpenID Foundation. OpenID: the internet identity layer, 2016. Available at: `http://openid.net`.

[86] OpenStack. Neutron firewall rules bypass through port update, 2015. Available at: `https://security.openstack.org/ossa/OSSA-2015-018.html`.

[87] 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.

[88] OpenStack. OpenStack Congress, 2015. Available at: `https://wiki.openstack.org/wiki/Congress`.

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

[90] OpenStack. OpenStack audit middleware, 2016. Available at: `http://docs.openstack.org/developer/keystonemiddleware`.

[91] OpenStack. OpenStack command list, 2016. Available at: `http://docs.openstack.org/developer/python-openstackclient/command-list.html`, last accessed on: February 14, 2018.

[92] OpenStack. OpenStack user survey, 2016. Available at: `https://www.openstack.org`.

[93] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy (SP)*, pages 233–247. IEEE, 2008.

[94] J. Pearl. Causality: Models, reasoning and inference, 2000.

[95] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on knowledge and data engineering*, 16(11):1424–1440, 2004.

[96] D. Petcu and C. Craciun. Towards a security SLA-based cloud monitoring service. In *CLOSER*, 2014.

[97] N. Pustchi and R. Sandhu. MT-ABAC: A multi-tenant attribute-based access control model with tenant trust. In *NSS*, 2015.

[98] K. Ren, C. Wang, and Q. Wang. Security challenges for the public cloud. *IEEE Internet Computing*, 16(1):69–73, 2012.

[99] R. Sandhu. The authorization leap from rights to attributes: maturation or chaos? In *SACMAT*, 2012.

[100] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 1996.

[101] N. Schear, P. T. Cable II, T. M. Moyer, B. Richard, and R. Rudd. Bootstrapping and maintaining trust in the cloud. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 65–77. ACM, 2016.

[102] J. Schiffman, Y. Sun, H. Vijayakumar, and T. Jaeger. Cloud verifier: Verifiable auditing service for iaas clouds. In *Services (SERVICES), 2013 IEEE Ninth World Congress on*, pages 239–246. IEEE, 2013.

[103] F. B. Schneider. Enforceable security policies. *Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[104] M. Solanas, J. Hernandez-Castro, and D. Dutta. Detecting fraudulent activity in a cloud using privacy-friendly data aggregates. Technical report, arXiv preprint, 2014.

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

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

[107] B. Tang and R. Sandhu. Extending OpenStack access control with domain trust. In *Network and System Security*, pages 54–69. Springer, 2014.

[108] K. Ullah, A. Ahmed, and J. Ylitalo. Towards building an automated security compliance tool for the cloud. In *TrustCom*, 2013.

[109] VMware. VMware vCloud Director. Available at: `https://www.vmware.com`.

[110] C. Wang, S. S. Chow, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE TC*, 2013.

[111] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE S&P*, 2012.

[112] Y. Wang, Q. Wu, B. Qin, W. Shi, R. H. Deng, and J. Hu. Identity-based data outsourcing with comprehensive auditing in clouds. *IEEE TIFS*, 2017.

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

[114] S. S. Yau, A. B. Buduru, and V. Nagaraja. Protecting critical cloud infrastructures with predictive capability. In *8th International Conference on Cloud Computing (CLOUD)*, pages 1119–1124. IEEE, 2015.

[115] H. Yu and D. Wang. Mass log data processing and mining based on hadoop and cloud computing. In *7th International Conference on Computer Science & Education (ICCSE)*, pages 197–202. IEEE, 2012.

[116] T. Zhang and R. B. Lee. CloudMonatt: An architecture for security health monitoring and attestation of virtual machines in cloud computing. In *42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 362–374. IEEE, 2015.

[117] X. Zhu, S. Song, J. Wang, S. Y. Philip, and J. Sun. Matching heterogeneous events with patterns. In *30th International Conference on Data Engineering (ICDE)*, pages 376–387. IEEE, 2014.