Multi-Level Proactive Security Auditing for Clouds

Suryadipta Majumdar*, Azadeh Tabiban†, Meisam Mohammady†, Alaa Oqaily†, Yosr Jarraya‡, Makan Pourzandi†, Lingyu Wang† and Mourad Debbabi†
*Information Security and Digital Forensics, University at Albany, USA
Email: smajumdar@albany.edu
†Concordia Institute for Information Systems Engineering, Concordia University, Canada
Email: {a_tabiba, m_ohamma, a_oqaily, wang, debbabi}@encs.concordia.ca
‡Ericsson Security Research, Ericsson Canada, Canada
Email: {yosr.jarraya, makan.pourzandi}@ericsson.com

***Abstract*—Runtime cloud security auditing plays a vital role in mitigating security concerns in a cloud. However, there currently does not exist a comprehensive solution that can protect a cloud tenant against the threats rendered from the multiple levels (e.g., user, virtual, and physical) of the cloud design. Furthermore, most of the existing solutions suffer from slow response time and require significant manual efforts. Therefore, a simple integration of the existing solutions for different levels is not a practical solution. In this paper, we propose a multi-level proactive security auditing system, which overcomes all the above-mentioned limitations. To this end, our main idea is to automatically build a predictive model based on the dependency relationships between cloud events, proactively verify the security policies related to different levels of a cloud by leveraging this model, and finally enforce those policies on the cloud based on the verification results. Our experiments using both synthetic and real data show the practicality and effectiveness of this solution (e.g., responding in a few milliseconds to verify each level of the cloud).**

***Index Terms*—cloud security, security auditing, proactive auditing, multi-level security**

I. Introduction

A wide-range of recent attacks (e.g., [1]–[6]) targeting different abstraction levels (e.g., user, virtual, and physical level) of a cloud infrastructure demonstrate severe security concerns in today's cloud platforms. These attacks may be launched by various actors, such as hostile tenants, careless cloud providers, and malicious insiders, and may have serious consequences such as threatening the cross-tenant isolation. For instance, stealing secrets through cross-tenant side-channels (e.g., [2], [3]), stealing computing resources [7] and bypassing security group rules [5] violate tenant isolation boundaries.

To mitigate such security threats in a cloud, one of the promising solutions is to verify the cloud against given security policies using formal verification techniques (a.k.a. *security auditing*). To this end, there exist several potential solutions (e.g., [8]–[16]), which can be divided into three major categories. First, the retroactive approach (e.g., [8]–[11]) verifies the security policies and catches violations after the fact. Second, the intercept-and-check approach (e.g., [13], [16]) intercepts each change request to the cloud and verifies the desired security policies while holding up the intercepted request. Third, the proactive approach (e.g., [12]–[15]) audits the cloud in advance before the request actually arrives.

However, none of the existing approaches protects a cloud from the threats rendering from its multiple levels (user, virtual, and physical). Furthermore, a simple integration of the existing solutions for different levels would be insufficient, as it would suffer from various practical issues. For instance, the retroactive approach cannot prevent irreversible damages (e.g., DoS and confidentiality breach). The current intercept-and-check solutions (e.g., [12], [13], [16]) cause prohibitive delay due to the sheer size of the cloud, and the existing proactive approaches (e.g., [12]–[15]) require significant manual efforts from the users. Therefore, there is a need for a practical auditing solution which can address the above-mentioned limitations and protect a cloud against the threats from its multiple levels.

In this paper, we propose an automated and efficient proactive auditing system for protecting multiple levels of a cloud. For this purpose, we first propose an automated approach to build predictive models to capture the dependency relationships among cloud events. Thus, we overcome the limitations of the existing methods that manually captures the dependency relationships. Then, we utilize this model to predict future events so that our tool can automatically verify these predicted events; which addresses the limitation of the works requiring manual input of future plans from admins. Finally, at runtime, we simply check the pre-computed verification results so that our solution can respond with a fairly small delay; that overcomes the inefficiencies of other solutions. We integrate our tool into OpenStack [17], one of the major cloud platforms, and evaluate the effectiveness and efficiency of our tool using both synthetic and real data.

The main contributions of this work are as follows.

- As per our knowledge, this is the first to propose a multi-level proactive security auditing solution, which protects a cloud against a wide-range of security threats directed at the different levels (i.e., user, virtual, and physical) of the cloud.
- We propose the first learning-based approach that can both automatically build the structure, and populate the parameters of the predictive model, with minimum need for manual inputs.

978-1-7281-2319-6/19/$31.00 ©2019 IEEE

- We integrate our solution to OpenStack, a major cloud platform, and our experimental results show negligible delay (e.g., a few milliseconds) in responding each request which demonstrates the efficiency of our solution.

The paper is organized as follows. Section II discusses preliminaries for our solution. Section III presents our proactive multi-level security auditing systems for clouds. Sections IV and V provide the implementation details and experimental results, respectively. Section VI summarizes related works. Section VII concludes the paper with future research directions.

II. Preliminaries

This section provides a background on system and dependency models, and defines our threat model.

A. Multi-Level System Model

Figure 1 shows a multi-level (e.g., user, virtual, and physical) system model of a cloud. We elaborate on each level as follows.

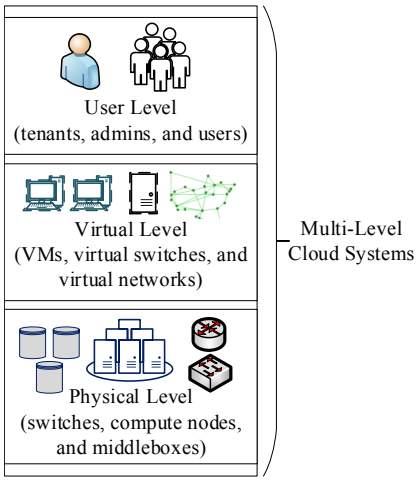

Fig. 1. A multi-level system model for clouds

User Level. The user level includes the administrative components of a cloud. More specifically, admins and users of a tenant including their interactions with the cloud platform are considered within this level. Additionally, the authentication and authorization mechanisms adopted in the cloud system are included in this level.

Virtual Level. The virtual level refers to the virtual components of a cloud. More specifically, virtual machines (VMs), virtual network components and other virtual resources are considered in this level. Additionally, the protocols involved to manage these virtual resources and their interactions with other levels are included in this level.

Physical Level. The physical level includes the physical elements of a cloud. More specifically, this level covers networking devices (e.g., switches, routers), computing nodes (e.g., servers) and middleboxes. Additionally, the protocols involved to manage these physical resources and their interactions with the virtual level are included in this level.

B. Background on Dependency Model

In this section, we discuss the existing dependency models, and highlight the drawbacks of these models. In the following, we elaborate on each of these dependencies.

Structural Dependencies. The structural dependency is the relationships among cloud events, which are imposed by the cloud management platform (e.g., OpenStack [17]). For instance, a structural dependency for the critical event (which may potentially breach a security property) *update port*, whose occurrence is possible only after the *create tenant*, *create network* and *create port* events in OpenStack. These dependencies are captured by studying the API specification of the cloud management platform, and provide an estimation about the distance (in terms of number of steps) to a critical event from any event.

Probabilistic Dependencies. The probabilistic dependency represents the behavioral pattern of cloud management operations including temporal order between event occurrences (as transitions). For instance, the probability of the critical event *add security group rule* occurrence is 0.625, given that the *create VM* event has already occurred. Such probabilistic information may provide an insight on the nature of these transitions, and help to predict the future transitions. These dependencies are learned using historical data (e.g., logs) by leveraging Bayesian network [15].

Temporal Dependencies. The temporal dependency is derived from the time intervals between occurrences of different cloud operations, and captures the patterns of this relationship. For example, the average time intervals to two different critical events (i.e., *add routing rule* and *create VM*) from the *create router* are significantly different (15s vs. 30m). Such variations in temporal distance can become critical in scheduling verification for these critical events in our approach (as will be discussed in Section III-A).

Limitations in Current Dependency Models. The current approaches (e.g., [14], [15]) to capture dependency models have the following limitations.

- The existing approaches build the structure (i.e., nodes and edges) of the model manually and automating this process may require heavy log pre-processing and mining.
- The current models do not include the temporal dependencies. Even though there exist other works (e.g., [18], [19]) capturing temporal dependencies, they are in a non-cloud environment.
- None of the existing models integrates diverse dependency relationships (e.g., structural, probabilistic and temporal) in a meaningful way. For instance, Figure 2 shows a case, where the probabilistic and temporal aspects provide the opposite directions of dependencies. The conditional probability for the occurrence of the critical event $E5$ is higher than that for the critical event $E6$ given that the event $E1$ has already happened. However, the event $E6$ requires less steps, or a smaller number of steps and less time to occur after the event $E1$.

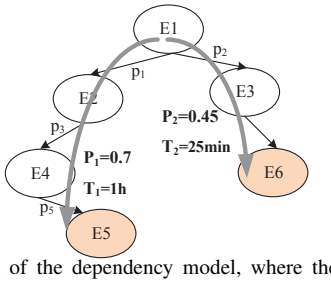

Fig. 2. An excerpt of the dependency model, where the probabilistic and temporal aspects show opposite dependencies

C. Threat Model

We assume that the cloud infrastructure management systems: a) may be trusted for the integrity of the API calls, event notifications, and database records (existing techniques on trusted computing and remote attestation may be applied to establish a chain of trust from TPM chips embedded inside the cloud hardware, e.g., [20]–[22]), and b) may have implementation flaws, misconfigurations and vulnerabilities that can be potentially exploited by malicious entities to violate security policies specified by cloud tenants. The cloud users including cloud operators and agents (on behalf of a human) may be malicious. This work focuses on attacks directed through the cloud management interfaces and more specifically, cloud management operations (e.g., create/delete/update tenant, user, VM, etc.). Any violation bypassing the cloud management interface is beyond the scope of this work.

III. Multi-Level Proactive Auditing System

This section presents our proactive multi-level security auditing solution.

A. Overview

Figure 3 shows an overview of our approach. This approach mainly performs two major steps. The first step is to build the predictive models of cloud events. The second step is to proactively conduct security auditing for the predicted events in different levels of a cloud and apply the security compliance decision to the cloud. In the following, we elaborate on each step.

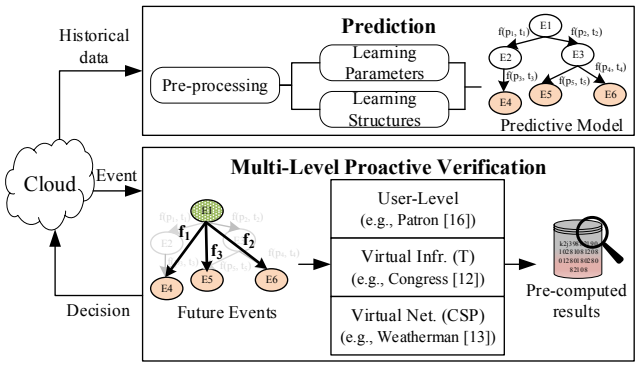

Fig. 3. A high-level overview of our approach

1) *Prediction:* To build a predictive model for cloud events, we first collect historical data (e.g., logs) from the cloud, then pre-process these logs to prepare them for the learning tools, afterwards capture the structure of the dependencies, and also measure different parameters (e.g., probabilities and time durations) from the processed cloud logs, and finally obtain the predictive model based on these learned structure and parameters.
2) *Multi-Level Proactive Verification:* To pre-compute security auditing results, we first intercept cloud events, then predict future events from the currently requested event using the obtained predictive model, afterwards proactively conduct the verification process for those predicted events by utilizing the existing verification tools for different levels, and finally apply the the verification decision (e.g., allow or deny) to the clouds.

B. Prediction

This section details our predictive model learning approach. To this end, we first discuss the steps of pre-processing of the logged data, then present our structure learning mechanism, and finally, describe how we derive the predictive model with its parameters.

Pre-Processing Bayesian Network Dataset. The pre-processing of logs is mainly to prepare the data for the learning tools. In the following, we discuss our pre-processing step. (i) *Parsing Logs:* To parse the logs, we use Logstash [23], a popular data processing tool, and feed it with sets of pre-defined rules, which, in turn, parses log entries and labels the extracted fields. (ii) *Grouping Log Entries of Each Tenant:* To provide a more accurate view of tenant-initiated activities, we group parsed log entries based on their corresponding tenant ID field. (iii) *Identifying Types of Events:* To learn the dependencies between events, we identify the event types corresponding to logged requests. (iv) *Aggregating Logs of Multiple Services:* To provide a comprehensive view of the probable temporal order between event occurrences, we aggregate the identified event types that are logged by different services and sort them based on their corresponding timestamp. (v) *Providing Input Dataset to Learning Tool:* Finally, to prepare the inputs for the learning tool (e.g., Bayesian network), we build the whole sequences of identified events, where each sequence represents dependencies between a group of events. The output of this step is forwarded to learn the structure and parameters of the model.

Learning the Model Structure. This section elaborates the main steps of structure learning. We encode the dependencies between logged events to the Bayesian network structure, where each directed edge represents the immediate consecutive occurrences of two event types corresponding to its end-vertices. For example, a directed edge between *start VM* and *stop VM* nodes (Figure 4) represents the order and their immediate appearances in the logged data.

However, considering the nature of our data, which reflects tenants' activities, this method introduces cycles to our graph. For example, it is likely that after stopping a VM, a user creates a security group, which forms a cycle in their corresponding graph (Figure 4). As a Bayesian network is an acyclic

graph by definition, we may need to remove the edges causing a cycle. However, removing edges comes with the cost of distorting probability distributions, and consequently decreasing the accuracy of our model. Likewise, ignoring the direction of cycle forming edges undermines the representativeness of the temporal order of events in our model.

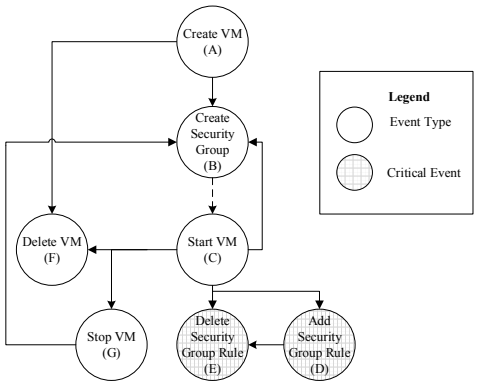

Fig. 4. An example of the model structure with cycles. Each node is an event and each edge is their transition.

To address this problem, we consider two objectives: i) minimizing the total number of removed edges, and ii) minimizing the dependency strength that is represented by a deleted edge. In the following, we elaborate on the importance of these factors to render the directed acyclic graph that corresponds to our logged identified events.

1) Total Number of Removed Edges: In most cases, our dependency graph includes multiple cycles that overlap in one or several edges. For example, in Figure 4, the edge between node B to C is involved in two cycles. By knowing the number of cycles an edge is involved in, we can grade the benefit of removing it. For instance, eliminating the edge from node B to C, removes two cycles in the above-mentioned case. However, by eliminating the edge from node G to B, we still need to eliminate another edge to remove the cycle between node B and C. Therefore, our goal is to prioritize the deletions of edges that maximize the number of removed cycles.
2) The Dependency Strength Represented by a Deleted Edge: The larger number of immediate occurrences of two identified events (e.g., the number of times an instance of the *stop VM* event type is observed immediately after an instance of the *start VM* event type in our processed logs) associates with a higher probable dependency between them. Therefore, removing edges that correspond to a larger number of occurred ordered pairs of events has a greater distortion effect on the probability distribution that is modeled by our Bayesian Network.

   Therefore, we assign a grade to all cycle forming edges based on the number of cycles they are involved in and the frequency of the immediate occurrences of their corresponding ordered pair of events. As the semantic of the two aforementioned factors implies, we consider an inverse correlation between them in our grading. Furthermore, we apply the same weights to both factors in our current grading method. We use these grades to remove cycles from Bayesian Network structure while minimizing the loss of accuracy. The algorithm of cycle removal is further discussed in Section IV. This obtained structure is then forwarded for learning the parameters and deriving the predictive model.

Building the Predictive Model. We first partition our training data into groups of events occurred during predetermined time periods. Next, for each time period, we find the conditional probability between all ordered pairs of events, called transitions, using the Bayesian Network module. Finally, all periodically learned probabilities of each transition are fed into the time-series predictor module. The time-series predictor takes inputs (Bayesian network module outputs for a certain period), and provides the predictive model of the probabilities. This model at each step is updated, and used to predict the future values of probabilities as we will require for our proactive verification in Section III-C. In this work, we use ARMAX as the predictor, which is a widely used method in prediction of stochastic processes in various fields.

C. Multi-Level Proactive Verification

This section details the multi-level proactive verification step of our solution. We elaborate on this step through two case studies, where we redesign and integrate existing verification tools to our auditing system to support proactive multi-level auditing.

User-Level Security Verification. The user-level security auditing is mainly conducted by protecting authorization and authentication mechanisms of the cloud. To this end, we leverage Patron [16], which enforces access control policies, and LeaPS [15], which enforces security properties related to authentication. In the following, we first provide a brief background on Patron, and then elaborate our adaption mechanism for it. LeaPS requires a similar adaption mechanism.

Background. Luo et al. [16] propose Patron, a runtime access control policy verification solution for clouds. To this end, they intercept each cloud management operation, fetch its related data, verify the event data against security policies, and apply the verification decision to the cloud. These steps are mainly performed by two modules: AEM and Patron. AEM is responsible for interpreting the events, filtering the events, managing the cache and interacting with the policy verifier (namely, Patron). Patron verifies each query (containing mainly intercepted event type and data) against a set of policies, and sends back the decision to AEM. For further description of Patron, the readers are referred to [16]. The main limitation of Patron is its slow response time, as it verifies the current intercepted event.

Adapting Mechanism. To address the above-mentioned limitation of Patron and to offer runtime user-level access control enforcement on the cloud, our main idea is to verify the predicted future events (instead of current event) and store these verification results in a cache. To this end, we mainly make three changes (steps 3-5 in the following algorithm) in

Patron as follows. First, we predict a list of future events by leveraging our predictive model (discussed in Section III-B). Second, instead of verifying the current intercepted event, we render Patron to verify the predicted list of events. Third, the verification results of predicted future events are stored in the cache. In the following, we describe the re-designed algorithm of Patron incorporating the aforementioned changes.

1) Each cloud management operation is intercepted at runtime.
2) The parameters of the intercepted operation are retrieved.
3) The distances to the critical events (i.e., the events that may violate a policy) are measured using the predictive model and a buffer of potential future events is initiated.
4) The buffer is continuously updated based on the current event and the predictive model.
5) The events in the buffer are sequentially verified prior to their occurrences against corresponding security policies.
6) These verification results are stored in a cache.
7) Upon the occurrence of a critical event, the decision is fetched from the cache (if possible). Otherwise, verification of the current event is conducted.
8) Based on the verification decision, the security policy is applied to the cloud.

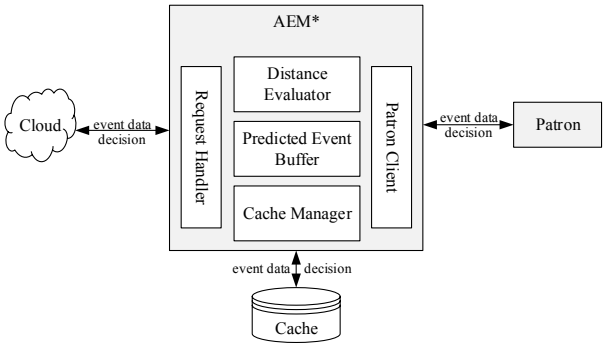

Fig. 5. The modified design of Patron to support runtime enforcement of user-level access control policies

Figure 5 depicts the modified design of Patron, which contains two major modules: AEM* and Patron. AEM* is the altered version of the original AEM module. The original Patron module is considered as a black box, and no modification is required to this module. AEM* further contains five different sub-modules. First, the request handler is to intercept each event, fetch the event parameters and apply the verification decision to the cloud. Second, the distance evaluator measures the distances from the intercepted event to different critical events. Third, the predicted event buffer maintains the queue of potential future events. Note that this list gets up-to-date after almost each operation. Fourth, the Patron client fetches the top of the event queue, and sends the query for that event to the Patron. Fifth, the caching manager stores and fetches verification decisions.

Virtual-Infrastructure-Level Security Verification. The virtual infrastructure layer security enforcement is mainly conducted by verifying a list of security properties (e.g., cross-tenant isolation, virtual network isolation, etc.) covering both cloud tenant and service provider levels. To this end, our candidate verifiers are Congress [12] and Weatherman [13], which verify a wide range of security properties at these levels. In the following, we detail the adaption mechanism of Congress starting with its background. Our adaption mechanism for Weatherman is the same, because of their similarity in the verification mechanism.

Background. Congress [12] is an OpenStack project to verify security policies to ensure governance and compliance for virtual infrastructures. Congress mainly operates in three modes. Among them, the proactive mode is to prevent any breach before they occur. This mode requires admins provide their future change plan in advance so that Congress can verify the legitimacy of the changes on its simulated environment. This requirement becomes impractical specially for a dynamic environment like cloud.

Adapting Mechanism. To overcome the aforementioned issue and to offer runtime security policy enforcement on the virtual infrastructure level of the cloud, our key idea is to verify a list of predicted future events instead of future change plan provided by the admins, and to store the verification results so that at runtime simply by checking these results, enforcement of policies can be performed. To this end, we mainly make three changes to the Congress verification process. First, based on the intercepted event, we predict future events using the predictive model. Second, Congress is invoked to verify the predicted future events. Third, we store these pre-computed verification results, based on which runtime security enforcement is conducted at the occurrence of any critical event.

In the following, we describe the steps of re-designed algorithms of Congress verification for runtime enforcement.

1) An initial list of critical events is provided as input.
2) Each cloud management operation is intercepted at runtime.
3) The parameters of the intercepted operation are retrieved.
4) The distances of the initial list of critical events from the intercepted event utilizing the predictive models is measured, and critical events are sorted based on this distance in the ascending order.
5) An allowed parameter list is prepared for each critical event based on the corresponding security policy.
6) To confirm the allowed parameters, Congress is invoked in its proactive mode to verify the sorted list of events with each of the allowed parameters.
7) If Congress finds no violation, then the parameter is stored.
8) At the interception of any of the critical events, the stored results are consulted for the decision (e.g., allow or deny).
9) The decision is applied to the cloud to preserve the policy.

The following example further illustrates the aforementioned steps.

Figure 6 shows the modified design of Congress. To this end, the distance evaluator prepares a sorted list of critical events based on the distances in the predictive model (similarly as in the user level). The context identifier retrieves the context of the intercepted event. The Congress (or Weatherman) verifier is considered as a black box, and no changes has been made. We maintain a buffer to store these pre-computed results for runtime enforcement.

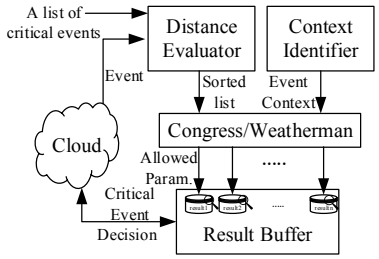

Fig. 6. The re-design of Congress (and Weatherman) to support runtime enforcement on the virtual infrastructure

IV. Implementation

This section presents the implementation details.

A. Architecture

There are five major components in our solution (Figure 7). The data collector continuously collects logs and configurations from the cloud platform. The predictive model builder pre-processes the raw logs, derives the probabilistic dependencies (using Bayesian network), and builds the predictive model using time-series. The interceptor intercepts cloud management operations, and applies the decision (e.g., allow/deny) from the cache to the cloud. The policy enforcement module pre-computes the heavy part of the verification in advance consulting the predictive model, and stores the verification results in the cache. Several security verifiers which are plugged to our system verify various levels of the cloud.

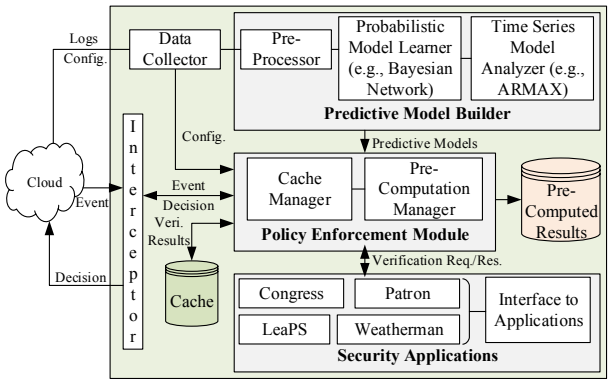

Fig. 7. A high-level architecture of our solution

B. Implementation Details

The implementation details of each module is discussed as follows. To parse unstructured logs, we first use Logstash [23], a data processing engine. In the next step, we investigate the requested event types corresponding to each log entry by providing mapping rules (i.e., the conjunction of METHOD and PATH_INFO for each REST API of OpenStack services). For the event types that cannot be uniquely identified through the conjunction of aforementioned attributes, we rely on the request content that is logged by PERMON [24], which is a pluggable interface in OpenStack compute and networking services. Next, we aggregate the processed logs of these services, and sort them based on their corresponding timestamps.

The resulted set of sequences is the input dataset to a Python Bayesian Network toolbox[1]. Then, we partition the whole chain of events using a customized algorithm (see Section III-B), through which we preserve all pairs of events. We build a primary structure events as nodes and all their logged immediate occurrences as edges. Next, we detect all exiting cycles using Python implementation of Johnson's algorithm [2]. We grade all the edges that are involved in a cycle, and their frequencies in our log data. To remove cycles from our initial structure, we use an iterative algorithm, where at each iteration, we delete the edge with the highest value of the aforementioned grade. After each iteration, we recalculate the grade of the edges belonging to the remaining cycles. The algorithm stops when there is no cycle left in the graph. The resulted Bayesian network is then provided to the ARMAX function of MATLAB and Statistics Toolbox Release 2017a to obtain the prediction model.

The interceptor is implemented as a middleware so that it intercepts each request that are made to Nova service (similarly as in [16], [24]). The body of requests, contained in the *wsgi.input* attribute of the intercepted requests, is scrutinized to identify the event type. We implement a cache (to accelerate the decision mechanisms) as memory-mapped file system (mmap) in Python (similarly as in [16]). Our cache structure is as follows: `(event type, caller_tenant_id::caller_user_id, target_tenant_id::target_id) -> decision`. We also maintain a MySQL database to store the intermediary pre-computed verification results. The interface to different security applications is implemented to customize part of their design to interact with the policy enforcement module.

V. Experiments

This section present our experimental results.

A. Experimental 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 with the Ubuntu 16.04 server. Based on a recent survey [25] on OpenStack, we simulate an environment with maximum 1,000 tenants and 100,000 VMs. The synthetic dataset includes over 4.5 millions records. We further utilize data collected from a real community cloud hosted at one of the largest telecommunications 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. We repeat each experiment at least 100 times.

B. Experimental Results

We present our obtained experimental results as follows.

Efficiency of Multi-Level Security Auditing. The objective of our first set of experiments is to demonstrate the efficiency

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

[2]https://github.com/qpwo/python-simple-cycles

of our proposed multi-level proactive security auditing solution. Figure 8 illustrates the response time (in milliseconds) for both user and virtual levels using the modified Patron and Congress (or Weatherman), respectively. In Figure 8(a), for different sizes of cache, we observe a quasi constant response time (i.e., less than one millisecond) through cache. This figure also shows that the pre-computation effort is around four milliseconds. Figure 8(b) shows the results of a similar experiment on virtual infrastructure verification (e.g., modified Congress). The response time remains within 6 milliseconds for 85.5% of of the time on average, and the prediction error may cost the pre-computation effort of up to 137 milliseconds (and the verification time of Congress in the proactive mode). Overall the results show the response time in several milliseconds in best cases and several hundred of milliseconds in worst cases.

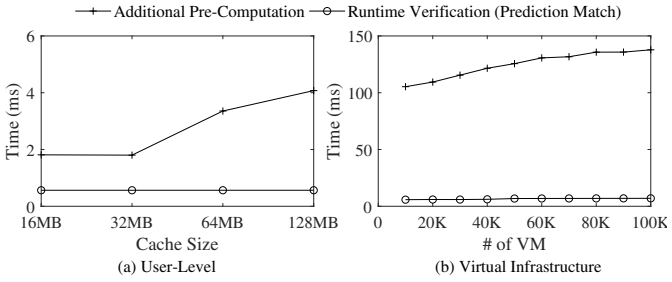

Fig. 8. Time required for both runtime verification and additional pre-computation for (a) user-level (using re-designed Patron) and (b) virtual infrastructure (using re-designed Congress) while varying the size of the cache and number of VMs, respectively

Accuracy of our Predictive Model. The second set of experiments is to show the accuracy improvements by our proposed predictive model. The prediction match rate refers to the percentage of time our prediction is correct. Therefore, higher prediction match rate ensures better response time. On the other hand, the prediction error rate refers to the situation where our prediction causes an inaccurate pre-computation. Therefore, the lower error rate ensures minimal wastage of computations. Figure 9 shows a quantitative comparison (in terms of prediction match and error) between our predictive model and the dependency model proposed in LeaPS [15]. More specifically, Figures 9 (a) and (b) show the percentages of prediction match and prediction error for three distant pairs of events in Figure 4, respectively. Figure 9(c) depicts the overall percentages of prediction match and prediction error. In all cases, our predictive model shows significant improvements. Specifically, the prediction match rate is increased up to two times and the prediction error rate is decreased up to 12 times. Figure 9(c) reports that our prediction results on average 10.1% false pre-computation. However, our system ensures the best response time (when results are in the cache) on average 85.5% of the time. With a selective choice of the threshold may provide up to 93% of prediction match. The price of a false prediction is measured in our previous work, Proactivizer [26].

In addition, we conduct similar experiments on real data. Despite the fact that the number of observations is relatively small (only around 400 records), the ARMAX model still depicts its superiority over the LeaPS model (e.g., up to 65% improvement in prediction match rate as shown in Table I).

Probability Threshold	0.4	0.45	0.5	0.55	0.6	0.65	0.7	0.75	0.8
ARMAX Prediction Match (%)	97.36	96.5	96.5	88.6	82.45	82.4	73.7	68.4	66.6
LeaPS Prediction Match (%)	65	65.8	65.8	65.8	65.78	65.8	65.8	65.8	65.8
Improvement Ratio (%)	48	46.6	46.6	65.8	34.6	25.3	12	4	1.3
ARMAX Prediction Error (%)	80.6	77.41	64.5	54.8	42	42	32.2	25.8	25.8
LeaPS Prediction Error (%)	74.2	74.2	74.2	74.2	74.2	74.2	74.2	74.2	74.2
Improvement Ratio (%)	-8.7	-4.3	13	26	43.5	56.5	65.2	65.2	78.2

TABLE I
EFFECTIVENESS OF ARMAX VS. LEAPS [15] FOR REAL DATA WITH THE SIZE OF 400 RECORDS

VI. RELATED WORK

In this section, we discuss different categories of related works. Retroactive auditing approach (e.g., [10], [11], [27], [28]) in the cloud is a traditional way of conducting auditing. 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., [12], [13]) perform major verification tasks while holding the event instances blocked. This approach usually causes significant delay (e.g., four minutes to verify mid-sized cloud [13]) to a user request. In contrast, our solution applies a proactive approach to overcome this limitation.

Proactive auditing approaches perform a part of the verification in advance. To this end, there exist several works (e.g., [12]–[15], [29]). Weatherman [13] and verify security policies on a future change plan in a virtualized infrastructure. PVSC [14] proactively verifies security compliance by utilizing the static patterns in dependency models. Both in Weatherman and PVSC, models are captured manually by expert knowledge. LeaPS [15] partially addresses this limitation by automating the parameter learning process of the model. However, LeaPS still relies on manual identification of the structure of the model and does not include the temporal dependencies in the model. More importantly, unlike our work, none of those works provides a comprehensive solution for multiple levels of clouds.

Additionally, many studies have been focusing on mining dependency relations from the ordering of log messages. Various definitions of temporal dependency have been proposed, such as, forwarding conditional probabilities [18], transition invariants (e.g., A always follow B) [30], and transition significance [31]. These studies mainly focus only on mining reliable pattern relations from the data and do not consider the overall quality of the structural events. Unlike ours, none of these works consider the dependencies among cloud events and time intervals between their transitions.

VII. CONCLUSIONS

In this paper, we proposed a multi-level proactive security auditing solution for clouds. More specifically, first, we proposed an automated approach to learn the structure of the dependencies in the cloud. Second, we derived a predictive model, which utilizes structural, probabilistic and temporal dependencies to predict the future events. Third, we re-designed and integrated four security solutions (e.g., Congress [12],

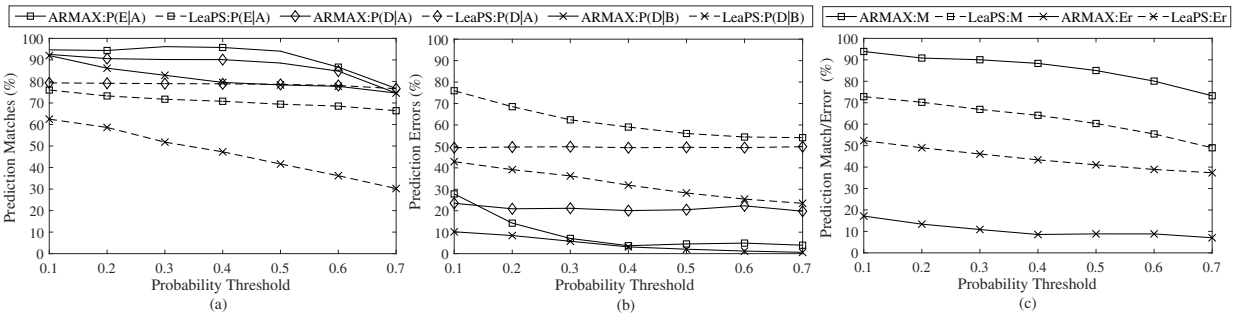

Fig. 9. Comparison between our predictive model using ARMAX and dependency model proposed in LeaPS [15] in terms of (a) percentage of prediction match for each pair of event (b) percentage of prediction error for each pair of event and (c) overall percentage of prediction match/error

Weatherman [13] Patron [16] and LeaPS [15]) to our system to offer multi-level proactive security aduiting. Finally, using both synthetic and real data, we conducted experiments to show the efficiency (e.g., responding in a few milliseconds) of our proposed solution. However, our solution comprises the following limitations, which we identify as potential future works. First, our system currently does not integrate any solution for virtual network layer 2 or SDN. Second, we currently rely on a specific time series model for prediction.

Acknowledgement. The authors thank the anonymous reviewers for their valuable comments. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under CRD Grant N01823 and by PROMPT Quebec.

REFERENCES

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

[2] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *CCS*. ACM, 2012.

[3] ——, "Cross-tenant side-channel attacks in paas clouds," in *CCS*. ACM, 2014.

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

[5] 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.

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

[7] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: improve your cloud performance (at your neighbor's expense)," in *CCS*. ACM, 2012.

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

[9] K. W. Ullah, A. S. Ahmed, and J. Ylitalo, "Towards building an automated security compliance tool for the cloud," in *TrustCom*. IEEE, 2013, pp. 1587–1593.

[10] T. Madi, S. Majumdar, Y. Wang, Y. Jarraya, M. Pourzandi, and L. Wang, "Auditing security compliance of the virtualized infrastructure in the cloud: Application to openstack," in *CODASPY*. ACM, 2016, pp. 195–206.

[11] S. Majumdar, T. Madi, Y. Wang, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi, "Security compliance auditing of identity and access management in the cloud: application to OpenStack," in *CloudCom*. IEEE, 2015, pp. 58–65.

[12] OpenStack, "OpenStack Congress," 2015, available at: https://wiki.openstack.org/wiki/Congress, last accessed on: February 14, 2018.

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

[14] 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.

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

[16] 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 *CLOUD*, 2016.

[17] OpenStack, "OpenStack open source cloud computing software," 2015, available at: http://www.openstack.org, last accessed on: February 14, 2018.

[18] W. Peng, C. Perng, T. Li, and H. Wang, "Event summarization for system management," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2007.

[19] W. Hämäläinen and M. Nykänen, "Efficient discovery of statistically significant association rules," in *ICDM*. IEEE, 2008, pp. 203–212.

[20] M. Li, W. Zang, K. Bai, M. Yu, and P. Liu, "Mycloud: supporting user-configured privacy protection in cloud computing," in *ACSAC*. ACM, 2013, pp. 59–68.

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

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

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

[24] A. Tabiban, S. Majumdar, L. Wang, and M. Debbabi, "Permon: An openstack middleware for runtime security policy enforcement in clouds," in *SPC*, June 2018.

[25] OpenStack, "OpenStack user survey," 2016, available at: https://www.openstack.org/assets/survey/October2016SurveyReport.pdf, last accessed on: Feb 14, 2018.

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

[27] C. Wang, S. S. Chow, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for secure cloud storage," *IEEE transactions on computers*, vol. 62, no. 2, pp. 362–375, 2013.

[28] Y. Wang, Q. Wu, B. Qin, W. Shi, R. H. Deng, and J. Hu, "Identity-based data outsourcing with comprehensive auditing in clouds," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 4, pp. 940–952, 2017.

[29] S. S. Yau, A. B. Buduru, and V. Nagaraja, "Protecting critical cloud infrastructures with predictive capability," in *CLOUD*. IEEE, 2015, pp. 1119–1124.

[30] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011.

[31] J. Kwon and K. M. Lee, "A unified framework for event summarization and rare event detection from multiple views," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 9, pp. 1737–1750, 2015.