LeaPS: Learning-Based Proactive Security Auditing for Clouds

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

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

Abstract. Cloud security auditing assures the transparency and accountability of a cloud provider to its tenants. However, the high operational complexity implied by the multi-tenancy and self-service nature, coupled with the sheer size of a cloud, imply that security auditing in the cloud can become quite expensive and non-scalable. Therefore, a proactive auditing approach, which starts the auditing ahead of critical events, has recently been proposed as a promising solution for delivering practical response time. However, a key limitation of such approaches is their reliance on manual efforts to extract the dependency relationships among events, which greatly restricts their practicality and adoptability. In this paper, we propose a fully automated approach, namely *LeaPS*, leveraging learning-based techniques to extract dependency models from runtime events in order to facilitate the proactive security auditing of cloud operations. We integrate LeaPS to OpenStack, a popular cloud platform, and perform extensive experiments in both simulated and real cloud environments that show a practical response time (e.g., 6ms to audit a cloud of 100,000 VMs) and a significant improvement (e.g., about 50% faster) over existing proactive approaches.

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

1 Introduction

Multi-tenancy in cloud has proved to be a double-edged sword leading to both resource optimization capability and inherent security concerns [37]. Moreover, the 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. Such complexities, coupled with the sheer size of the cloud (e.g., 1,000 tenants and 100,000 users in a decent-sized cloud [35]), can usually render security auditing in cloud expensive and non-scalable. In fact, verifying every user event at runtime can cause considerable delays even in a mid-sized cloud (e.g., over four minutes [6]). 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 a auditing system, auditing larger clouds could incur prohibitive costs.

To this end, a promising solution for reducing the response time of security auditing in clouds to a practical level is the proactive approach (e.g., [6, 25]). 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 [44, 25]. However, a key limitation of existing proactive approaches 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. The manual and static nature of existing proactive auditing approaches prevents them from realizing their full potential, which will be further illustrated through a motivating example in the following.

Motivating Example. The upper part of Figure 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 approaches to auditing such events. We discuss their limitations below to motivate our solution.

- With a traditional runtime verification approach, most of the verification effort (depicted as boxes filled with vertical patterns) 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 [33], 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.

To address those limitations, our key idea is to design a probabilistic (instead of deterministic) dependency model, and automatically extract such a model from runtime events through learning techniques. Specifically, we first design a novel probabilistic model for capturing the dependency relationships between event types while taking into consideration the inherent uncertainty in such relationships. Second, we provide detailed methodology and algorithms for our learning-based proactive security auditing system, namely, *LeaPS*. We describe our implementation of the proposed system based on OpenStack [33], and demonstrate how the system may be easily ported to other cloud platforms (e.g., Amazon EC2 [1] and Google GCP [13]). 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).

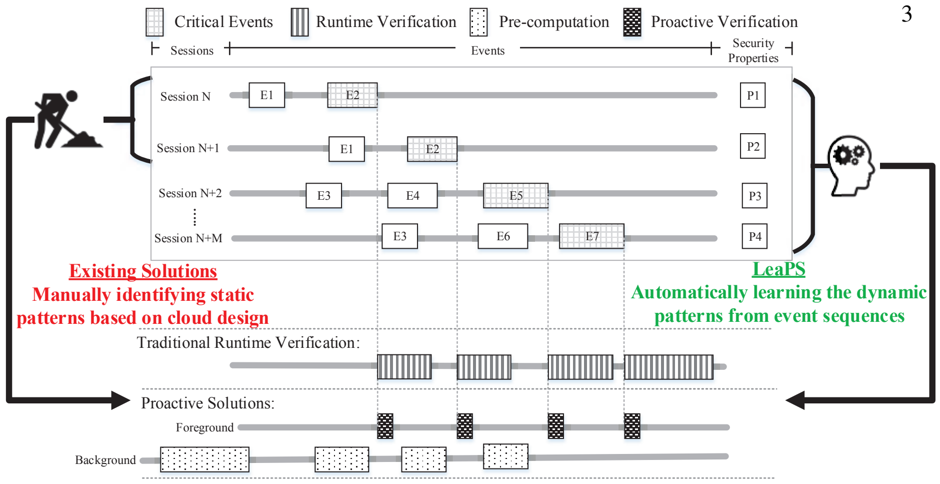


Fig. 1: Identifying the main limitations of both traditional runtime verification and proactive verification solutions and positioning our solution.

In summary, our main contributions are threefold.

- To the best of our knowledge, this is the first learning-based, probabilistic approach to proactive security auditing in clouds. First, our probabilistic dependency model allows handling the uncertainty that is inherent to runtime events. Second, our learning-based methodology eliminates the need for impractical manual efforts required by other proactive solutions.
- 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 positives rates; any inaccuracy in the learning results would only affect the efficiency, as will be demonstrated through experiments later in the paper. We believe this idea of leveraging learning for efficiency, instead of security, may be adapted to benefit other security solutions.
- As demonstrated by our implementation and experimental results, the proposed system, LeaPS, provides an automated, efficient, and scalable solution for different cloud platforms to increase their transparency and accountability to tenants.

The remainder of the paper is organized as follows. Section 2 describes the threat model and the dependency models. Section 3 details our methodology. Section 4 provides the implementation details and experimental results. Section 5 discusses several aspects of our approach. Section 6 reviews related works. Section 7 concludes the paper.

2 Models

In this section, we describe the threat model and define our dependency model.

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 may be applied to establish a chain of trust from TPM chips embedded inside the cloud hardware, e.g., [3, 21]). 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 (we provide a guideline on identifying critical events in Appendix A). 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.

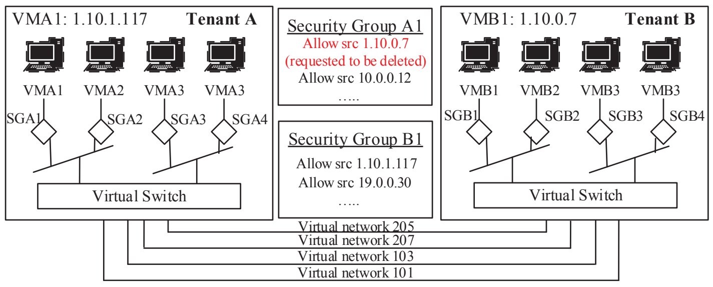


Fig. 2: An exploit of a vulnerability in OpenStack [31], leading to bypassing the security group mechanism.

Running Example. A real world vulnerability in OpenStack[1], CVE-2015-7713 [31], 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 2 shows a potential attack scenario exploiting 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 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 events (steps 1-4) happens in the mentioned order (namely, *event sequence*). In the next section, we present our dependency model and how it allows capturing rich and dynamic patterns of event sequences in the cloud.

2.2 The Dependency Model

In this section, 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 3).

[1] OpenStack [33] is a popular open-source cloud infrastructure management platform.

Example 1. Figure 3 shows an example of a dependency model, where nodes represent different event types in 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.

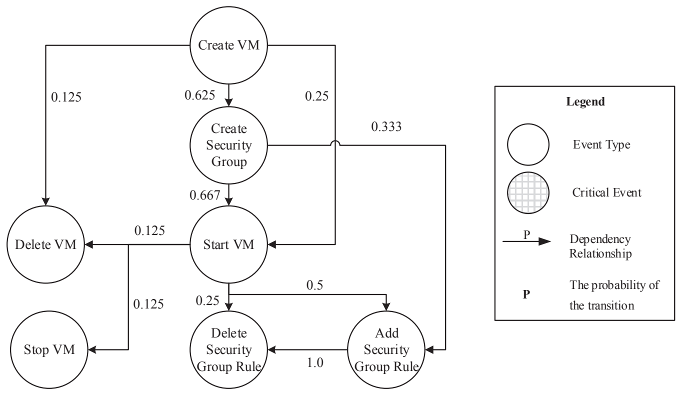


Fig. 3: An example dependency model represented as a Bayesian network.

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 2.1.

– *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 3 may be extracted from such data (note above we only show an excerpt of the data needed to construct the complete model, due to space limitations). 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 [36] 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 3, 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.

Definition 1. 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 are given as *critical events* that might breach some given *security properties*.

3 LeaPS Auditing System

This section presents our learning-based proactive security auditing system (LeaPS).

3.1 Overview

We briefly describe the auditing process of LeaPS as follows. First, it builds offline a dependency model capturing the probabilistic patterns of event type sequences in the form of a Bayesian network by learning from the runtime event instances captured in cloud event logs. Then, once the model is constructed, it is used by the online modules in order to decide, based on the current observed event instances, the most likely next critical event to occur. This would trigger the proactive modules of our auditing system to pre-compute the required conditions that should be verified, when the critical event actually occurs. We iterate on these tasks to incrementally pre-compute for other likely critical events based on the decreasing order of their conditional probabilities in the model. Once one of these critical events actually occurs, we simply verify the parameters of the events with respect to the pre-computed conditions of that event.

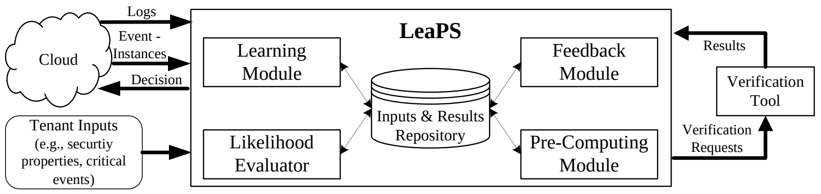


Fig. 4: An overview of LeaPS auditing approach.

Figure 4 shows an overview of LeaPS. LeaPS contains four major modules: learning, pre-computing, likelihood evaluator and feedback. The learning module is mainly responsible for conducting the learning of the probabilistic dependency model. The likelihood evaluator basically triggers the pre-computation based on the model. The pre-computing module prepares the ground for the runtime verification. At runtime, a light-weight verification tool [25], 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 to the tenant. The feedback module provides feedbacks to different modules (e.g., learning module) to improve the performance of proactive auditing. In LeaPS, the data transfer between modules is performed through a repository. In the following, we detail each module in LeaPS.

3.2 Learning Module

The major steps of this module are: processing logs and learning dependency models.

Log Processor. The event logs in the cloud are used to learn the dependencies between different event types. However, log files generated by the existing cloud platforms are not suitable to be directly fed into the learning engine, as user events are generally mixed up with many other system-initiated events. Furthermore, logs usually contain many implementation specific details (e.g., platform-specific APIs). Therefore, the log processor is responsible for eliminating such system-initiated events and to map the relevant events into implementation-independent event types (more details will be provided in Section 5). Also, the log processor groups event sequences based on their dates (i.e., each group of event sequences for each day), and generates inputs (in a specific format) to the learning engine. Table 1 shows examples of OpenStack log entries and the output format of processed logs in LeaPS.

OpenStack Log Entry	Output Format of LeaPS
`[01/Apr/2017:10:55:41 -0400] "POST /v2/servers HTTP/1.1"`	`Create VM`
`[01/Apr/2017:11:00:45 -0400] "POST /v2/os-security-groups HTTP/1.1"`	`Create security group`
`[01/Apr/2017:11:01:15 -0400] "GET /v2/os-security-groups HTTP/1.1"`	Eliminated

Table 1: Examples of OpenStack logs and output format of processed logs in LeaPS.

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 [30, 15, 36] (this choice has been justified in Section 2.2). We now first demonstrate the learning steps through an example, and then provide further details.

Example 2. Figure 5 shows the dependency model of Figure 3 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 *create VM* to *delete VM*, *start VM* and *create security group*. Then, $P(DV|CV)$, $P(CSG|CV)$, $P(SV|CV)$ and other conditional probabilities (between immediate 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 are learned from the processed logs obtained from the log processor. The third values in the labels of Figure 5 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

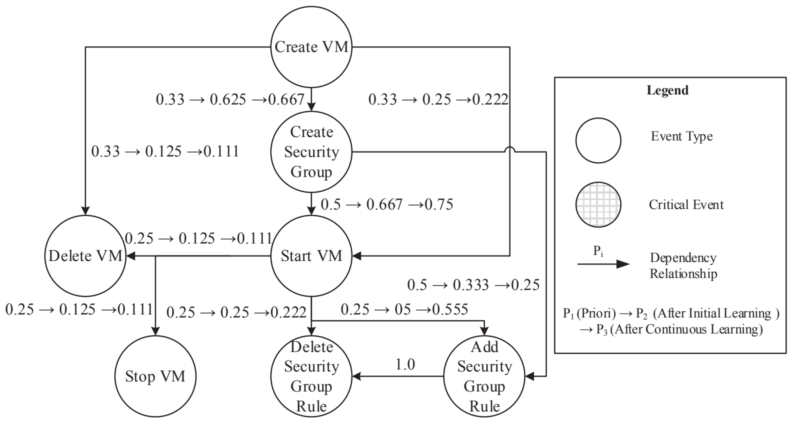


Fig. 5: The outcomes of three learning steps for the dependency model.

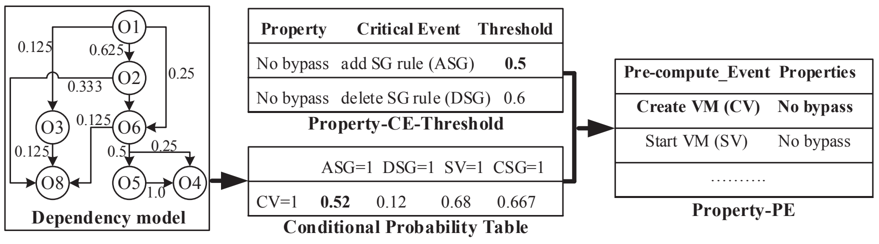
and edges in a Bayesian network, is first derived from the set of event types received as input. To this end, we provide a guideline on identifying such a set of event types in Section 5. Initially, the system considers every possible edge between nodes (and eventually delete the edges with probability 0), and conditional probabilities between immediate 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 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 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.

3.3 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.

Example 3. Figure 6 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 [8] 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 in the *Conditional-Probability* table. The conditional probability for *ASG* having *CV* (*P(ASG/CV)*) is 0.52

[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 scope of this paper.



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
……….	

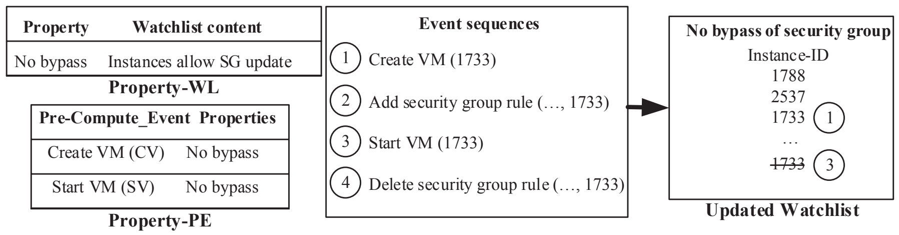
Property-PE

Fig. 6: An excerpt of the likelihood evaluator steps and their outputs.

in the *Conditional-Probability* table in Figure 6. Next, this value is compared with the thresholds of the *no bypass* property in the *Property-CE-thresholds* 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.

3.4 Pre-Computing Module

The purpose of the pre-computing module is to prepare the ground for the runtime verification. In this paper, we mainly discuss watchlist-based pre-computation [25]; 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 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 mean time. The main functionality of the pre-computing module is described using the following example.



Property	Watchlist content
No bypass	Instances allow SG update

Property-WL

Pre-Compute_Event	Properties
Create VM (CV)	No bypass
Start VM (SV)	No bypass

Property-PE

Event sequences

1 Create VM (1733)
2 Add security group rule (…, 1733)
3 Start VM (1733)
4 Delete security group rule (…, 1733)

No bypass of security group
Instance-ID
1788
2537
1733 1
…
~~1733~~ 3

Updated Watchlist

Fig. 7: Showing steps of the updating watchlist for a sample event sequences.

Example 4. Left side of Figure 7 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 *start VM 1733* operation and identify the event in the *Property-PE*. 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.

3.5 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 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 miss 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.

4 Application to OpenStack

This section describes LeaPS implementation, and presents our experimental results.

4.1 Implementation

In this section, we detail the LeaPS implementation and its integration into OpenStack along with the architecture of LeaPS (Figure 8) and a detailed algorithm (Algorithm 1).

Background. OpenStack [33] is an open-source cloud management platform in which Neutron is its network component, Nova is its compute component, and Ceilometer is its telemetry for receiving event histories from other components. In this work, we collect Ceilometer logs to later use for learning the dependencies. For learning, we leverage SMILE & GeNIe [2], which is a popular tool for modeling and learning with Bayesian network. SMILE & GeNIe uses the EM algorithm [9, 20] for parameter learning.

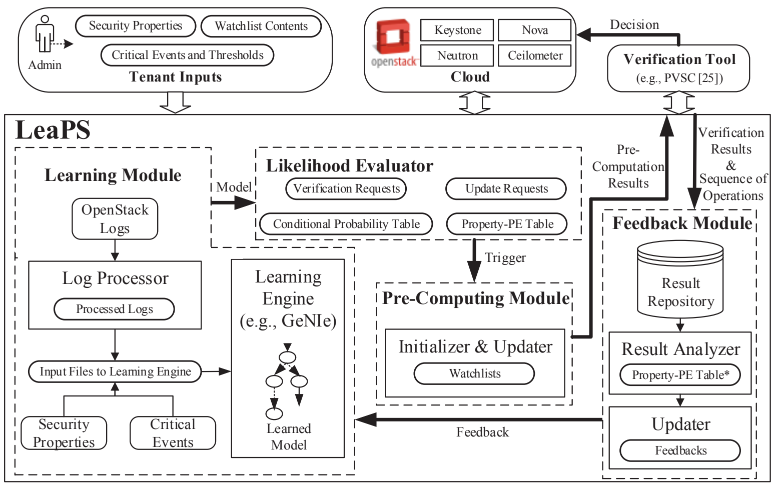


Fig. 8: A high-level architecture of LeaPS auditing system.

Integration to OpenStack. Figure 8 shows a high-level architecture of LeaPS. The learning module is responsible for processing OpenStack logs, preparing inputs for GeNIe, and conducting the learning process using GeNIe. LeaPS first automatically collects logs from different OpenStack components, e.g., Nova, Neutron, Ceilometer, etc. Then, these logs are converted to the input format (in .dat) of GeNIe. Additionally, the structure of the network and its parameters are provided to GeNIe. We intercept events based on the audit middleware [34] in Keystone, which was previously supported by PyCADF [7], to intercept Neutron and Nova events by enabling the audit middleware and configuring filters. The pre-computing module stores its results into a MySQL database, and the feedback module is implemented in Python. Those modules work together to support the methodology described in Section 3, as detailed in Algorithm 1.

Algorithm 1: LeaPS Auditing ($CloudOS$, $Properties$, $critical\text{-}events$, WL)

```
1:  procedure LEARN($CloudOS$, $Proeprty\text{-}CE\text{-}Threshold$)
2:      for each component $c_i \in CloudOS$ do
3:          $processedLogs$ = processLog($c_i.logs$)
4:      for each property $p_i \in Properties$ do
5:          $structure$ = buildStructure($p_i$, $critical\text{-}events$)
6:          $learnedModels$ = learnModel($structure$, $processedLogs$)

7:  procedure EVALUATE-LIKELIHOOD($CloudOS$, $WL$, $Property\text{-}PE$, $Event\text{-}Operation$)
8:      for each event type $e_i \in CloudOS.event$ do
9:          $Conditional\text{-}Probability\text{-}Table$ = inferLikelihood($e_i$, $learnedModels$)
10:         if checkThreshold($Conditional\text{-}Probability\text{-}Table$,$Property\text{-}CE\text{-}Threshold$) then
11:             insertProperty-PE($e_i$,$Property\text{-}CE\text{-}Threshold.property$)
12:     $interceptedEvent$ = intercept-and-match($CloudOS$, $Event\text{-}Operation$)
13:     if $interceptedEvent \in critical - events$ then
14:         $decision$ = verifyWL($WL$, $interceptedEvent.params$)
15:         return $decision$
16:     else if $interceptedEvent \in Property\text{-}PE$ then
```

```
17:         Pre-compute-update($WL$, $property$, $interceptedEvent.params$)
18: procedure PRE-COMPUTE-INITIALIZE($CloudOS$, $Property\text{-}WL$)
19:     for each property $p_i \in Properties$ do
20:         $WL_i$= initializeWatchlist($p_i$, $Property\text{-}WL$, $CloudOS$)
21: procedure PRE-COMPUTE-UPDATE($WL$, $property$, $parameters$)
22:     updateWatchlist($WL$, $property$, $parameters$)
23: procedure FEEDBACK($Result$, $learnedModels$)
24:     storeResults($Result$, $learnedModels$)
25:     if analyzeSequence($Result.seq$) = “new-trend” then
26:         updateModel($Result.seq$,‘new’)
27:     else
28:         updateModel($Result.seq$,‘old’)
29:     for each property $p_i \in Properties$ do
30:         $change\text{-}in\text{-}threshold[i]$ = analyzeDecision($Result.decision$, $p_i$)
```

LeaPS interacts with three external entities (i.e., tenant, cloud platform and the verification tool). Cloud tenants provide security properties, and their thresholds, specification of watchlist contents and critical events to LeaPS. Then, OpenStack is responsible for providing the logs from its different components. We also leverage a verification tool [25], which verifies parameters of an intercepted critical event with the watchlists.

4.2 Experimental Results

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

Experimental Settings. Our test 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 [35] on OpenStack, we simulated 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 learning, we use GeNIe academic version 2.1. We repeat each experiment 100 times.

Results. The objective of the first set of experiments is to demonstrate the time efficiency of LeaPS. Figure 9(a) shows the time in milliseconds required by LeaPS to verify the *no bypass of security group* [8] and *no cross-tenant port* [18] 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 9(b) 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 10(a), 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

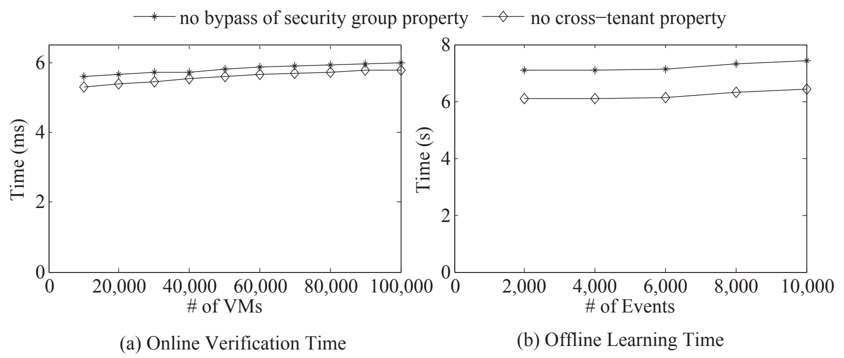


Fig. 9: Showing time required for the (a) online runtime verification by varying the number of VMs and (b) offline learning process by varying the number of event instances in the logs for the *no bypass* and *no cross-tenant* properties. The verification time includes the time to perform interception, matching of event type and checking in the watchlist.

initialization of pre-computation is performed in 50 seconds for the largest dataset. Figure 10(b) 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.

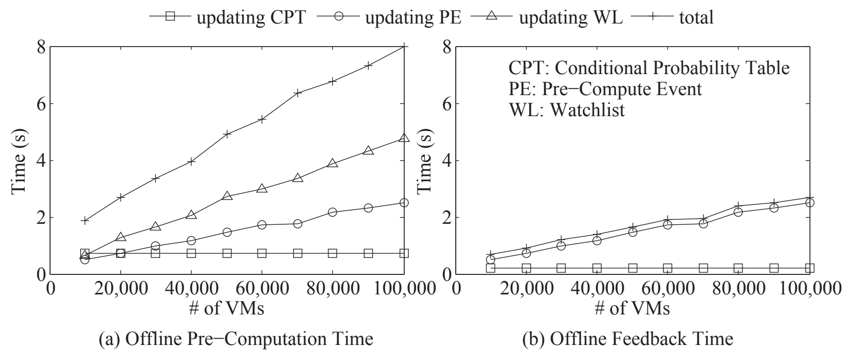


Fig. 10: Showing time required in seconds for the (a) pre-computation and (b) feedback modules considering the *no bypass* property by varying the number of instances.

In the second set of experiments, we demonstrate that how much LeaPS may be affected from 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 1. Figure 11(a) 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 [25]), where all possible paths are considered with equal weight, and number of steps in the model is the deciding factor for the pre-computation. Figure 11(b) 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.

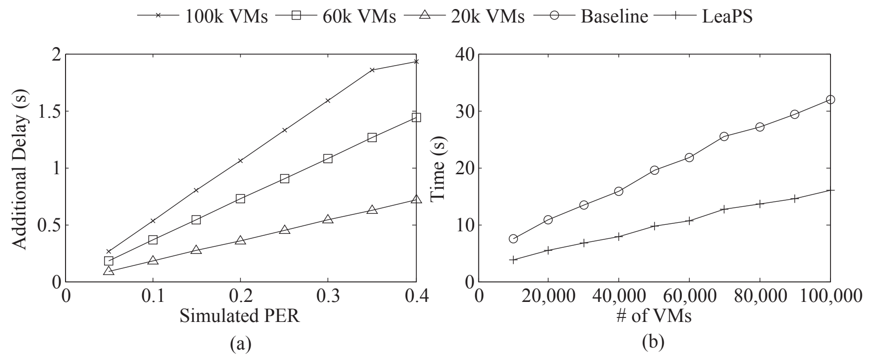


Fig. 11: (a) The additional delay (in seconds) in LeaPS pre-computation time caused by different simulated prediction error rates (PER) of a learning tool, (b) the comparison (in seconds) between LeaPS and a baseline approach.

Experiment with Real Cloud. We further test LeaPS using data collected from a real community cloud hosted at one of the largest telecommunications vendors. The main objective is to evaluate the applicability of our approach in a real cloud environment. 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 was not configured in this cloud, we utilize Nova and Neutron logs which increases the log processing efforts significantly. Table 2 summarizes the obtained results. We first measure 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 9.62ms 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 2: 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 Discussions

Adapting to Other Cloud Platforms. LeaPS is designed to work with most popular cloud platforms (e.g., OpenStack [33], Amazon EC2 [1], Google GCP [13], Microsoft Azure [28]) with a minimal one-time effort. Once a mapping of the APIs from these platforms to the LeaPS event types are provided, rest of the steps in LeaPS are platform-agnostic. Table 3 enlists some examples of such mappings.

LeaPS Event Type	**OpenStack [33]**	**Amazon EC2-VPC [1]**	**Google GCP [13]**	**Microsoft Azure [28]**
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 3: Mapping event APIs of different cloud platforms to LeaPS event types.

Possibility of a DoS Attack against LeaPS. To exploit the fact that a wrong prediction may result 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 11(a) shows that an attacker requires to inject significantly large amount (e.g., 40% error rate) of biased event instances to even cause a delay of two seconds. Moreover, biasing the model is non-trivial unless the attacker has prior knowledge on 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 for 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 better profile of each business or tenant. This level of learning is mainly suitable for companies following process management strictly 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 less users (e.g., admins) who perform cloud events. Conversely, if a company follows process management, user level learning will be an overkill, as different users would exhibit very similar patterns.

6 Related Work

Table 4 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 objectives (auditing or anomaly detection). The next columns compare these works according to different features, i.e., proactiveness, automated and dynamic dependency capturing, cloud-platform-agnostic and probabilistic dependency.

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 with new trends. Finally, the LeaPS methodology is cloud-platform agnostic.

Proposals	Methods	Coverage		Features				
		Environment	Objective	Proactive	Automatic	Dynamic	Agnostic	Probabilistic
Doelitzscher et al. [10]	Custom Algorithm	Cloud	Auditing	-	N/A	●	●	N/A
Ullah et al. [40]	Custom Algorithm	Cloud	Auditing	-	N/A	-	-	N/A
Majumdar et al. [26]	CSP Solver	Cloud	Auditing	-	N/A	-	-	N/A
Madi et al. [24]	CSP Solver	Cloud	Auditing	-	N/A	-	-	N/A
Jiang et al. [19]	Regression Technique	Non-cloud	Anomaly Det.	●	●	-	N/A	●
Solanas et al. [39]	Classifiers	Cloud	Anomaly Det.	-	●	-	-	●
Ligatti et al. [23]	Model Checking	Non-Cloud	Auditing	●	N/A	●	N/A	-
PVSC [25]	Custom Algorithm	Cloud	Auditing	●	-	-	-	-
Weatherman [6]	Graph-theoretic	Cloud	Auditing	●	-	●	-	-
Congress [32]	Datalog	Cloud	Auditing	●	-	-	-	-
LeaPS	Custom + Bayesian	Cloud	Auditing	●	●	●	●	●

Table 4: Comparing existing solutions with LeaPS. The symbols (●), (-) and N/A mean supported, not supported and not applicable respectively.

Retroactive and Intercept-and-Check Auditing. Retroactive auditing approach (e.g., [24, 26, 41, 42, 40, 10] 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., [6, 32]) 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., [6, 25, 43]). Weatherman [6] verifies security policies on a future change plan in a virtualized infrastructure using the graph-based model proposed in [5, 4]. PVSC [25] 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 [32] is an OpenStack project offering similar features as Weatherman. Foley et al. [12] proposes an algebra for anomaly-free firewall policies for OpenStack. Many state-based formal models (e.g., [38, 22, 23, 11] are proposed for program monitoring. Our work further expands the proactive monitoring approach into cloud differing in scope and methodology.

Learning-based Detections. There are many learning-based security solutions (e.g., [39, 14, 16, 19, 29, 17, 27]), which offer anomaly detection. Unlike above-mentioned works, this paper proposes a totally different learning-based techniques to facilitate the proactive auditing.

7 Conclusion

In this paper, we proposed LeaPS, a fully automated system leveraging the learning-based techniques to accelerate the performance of a proactive auditing approach. We integrated LeaPS to OpenStack, and evaluated the performance of LeaPS extensively (using both synthetic and real data) which show LeaPS keeps the response time to a practical level (e.g., about 6ms to verify 100,000 VMs), and improves the speed up significantly (e.g., about 50%) over existing proactive approaches. As future work, we will investigate the possibility of applying other learning techniques to further improve the efficiency; we will also apply supervised learning to automate the process of identifying critical events and security properties from logs.

References

1. Amazon. Amazon virtual private cloud. Available at: `https://aws.amazon.com/vpc`.
2. BayesFusion. GeNIe and SMILE. Available at: `https://www.bayesfusion.com`.
3. M. Bellare and B. Yee. Forward integrity for secure audit logs. Technical report, Citeseer, 1997.
4. S. Bleikertz, T. Groß, M. Schunter, and K. Eriksson. Automated information flow analysis of virtualized infrastructures. In *ESORICS*, 2011.
5. S. Bleikertz, C. Vogel, and T. Groß. Cloud Radar: Near real-time detection of security failures in dynamic virtualized infrastructures. In *ACSAC*, 2014.
6. S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim. Proactive security analysis of changes in virtualized infrastructure. In *ACSAC*, 2015.
7. Cloud auditing data federation. PyCADF: A Python-based CADF library, 2015. Available at: `https://pypi.python.org/pypi/pycadf`.
8. Cloud Security Alliance. Cloud control matrix CCM v3.0.1, 2014. Available at: `https://cloudsecurityalliance.org/research/ccm/`.
9. 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*, 1977.
10. F. Doelitzscher, C. Fischer, D. Moskal, C. Reich, M. Knahl, and N. Clarke. Validating cloud infrastructure changes by cloud audits. In *IEEE SERVICES*, 2012.
11. E. Dolzhenko, J. Ligatti, and S. Reddy. Modeling runtime enforcement with mandatory results automata. *International Journal of Information Security*, 2014.
12. S. N. Foley and U. Neville. A firewall algebra for OpenStack. In *IEEE CNS*, 2015.
13. Google. Google cloud platform. Available at: `https://cloud.google.com`.
14. S. Guha. *Attack Detection for Cyber Systems and Probabilistic State Estimation in Partially Observable Cyber Environments*. PhD thesis, Arizona State University, 2016.
15. D. Heckerman. A tutorial on learning with Bayesian networks. In *Learning in graphical models*. 1998.
16. R. A. Hemmat and A. Hafid. SLA violation prediction in cloud computing: A machine learning perspective. Technical report, 2016.
17. H. Holm, K. Shahzad, M. Buschle, and M. Ekstedt. P^2 CySeMoL: Predictive, probabilistic cyber security modeling language. *IEEE TDSC*, 2015.
18. ISO Std IEC. ISO 27017. *Information technology- Security techniques- Code of practice for information security controls based on ISO/IEC 27002 for cloud services (DRAFT), `http://www.iso27001security.com/html/27017.html`*, 2012.
19. 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*. ACM, 2010.
20. S. L. Lauritzen. The EM algorithm for graphical association models with missing data. *Computational Statistics & Data Analysis*, 19(2):191–201, 1995.
21. M. Li, W. Zang, K. Bai, M. Yu, and P. Liu. MyCloud: supporting user-configured privacy protection in cloud computing. In *ACSAC*, 2013.
22. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM TISSEC*, 2009.
23. J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *ESORICS*, 2010.
24. T. Madi, S. Majumdar, Y. Wang, Y. Jarraya, M. Pourzandi, and L. Wang. Auditing security compliance of the virtualized infrastructure in the cloud: Application to OpenStack. In *ACM CODASPY*, 2016.
25. 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.

26. S. Majumdar, T. Madi, Y. Wang, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi. Security compliance auditing of identity and access management in the cloud: Application to OpenStack. In *IEEE CloudCom*, 2015.
27. S. Mehnaz and E. Bertino. Ghostbuster: A fine-grained approach for anomaly detection in file system accesses. In *ACM CODASPY*, 2017.
28. Microsoft. Microsoft Azure virtual network. Available at: `https://azure.microsoft.com`.
29. R. Mitchell and R. Chen. Behavior rule specification-based intrusion detection for safety critical medical cyber physical systems. *IEEE TDSC*, 2015.
30. K. Murphy. A brief introduction to graphical models and bayesian networks. 1998.
31. OpenStack. Nova network security group changes are not applied to running instances, 2015. Available at: `https://security.openstack.org/ossa/OSSA-2015-021.html`.
32. OpenStack. OpenStack Congress, 2015. Available at: `https://wiki.openstack.org/wiki/Congress`.
33. OpenStack. OpenStack open source cloud computing software, 2015. Available at: `http://www.openstack.org`.
34. OpenStack. OpenStack audit middleware, 2016. Available at: `http://docs.openstack.org/developer/keystonemiddleware/audit.html`.
35. OpenStack. OpenStack user survey, 2016. Available at: `https://www.openstack.org/assets/survey/October2016SurveyReport.pdf`.
36. J. Pearl. Causality: Models, reasoning and inference, 2000.
37. K. Ren, C. Wang, and Q. Wang. Security challenges for the public cloud. *IEEE Internet Computing*, (1):69–73, 2012.
38. F. B. Schneider. Enforceable security policies. *ACM TISSEC*, 2000.
39. M. Solanas, J. Hernandez-Castro, and D. Dutta. Detecting fraudulent activity in a cloud using privacy-friendly data aggregates. Technical report, arXiv preprint, 2014.
40. K. Ullah, A. Ahmed, and J. Ylitalo. Towards building an automated security compliance tool for the cloud. In *IEEE TrustCom'13*.
41. C. Wang, S. S. Chow, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE TC*, 2013.
42. 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.
43. S. S. Yau, A. B. Buduru, and V. Nagaraja. Protecting critical cloud infrastructures with predictive capability. In *IEEE CLOUD*, 2015.
44. X. Zhu, S. Song, J. Wang, S. Y. Philip, and J. Sun. Matching heterogeneous events with patterns. In *IEEE ICDE*, 2014.

A A Guideline to Choose LeaPS Inputs

We provide a guideline on identifying different inputs of LeaPS. Identifying sets of event types as the input to the learning engine are described 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 is to be highlighted; and iii) from the attack scenario, the parameters for the watchlist for each critical event is finalized.