Proactivizer: Transforming Existing Verification Tools into Efficient Solutions for Runtime Security Enforcement

Suryadipta Majumdar[1], Azadeh Tabiban[2], Meisam Mohammady[2], Alaa Oqaily[2], Yosr Jarraya[3], Makan Pourzandi[3], Lingyu Wang[2], and Mourad Debbabi[2]

[1] Information Security and Digital Forensics, University at Albany, USA
`smajumdar@albany.edu`
[2] Concordia Institute for Information Systems Engineering, Concordia University, Canada
`{a_tabiba,m_ohamma,a_oqaily,wang,debbabi}@encs.concordia.ca`
[3] Ericsson Security Research, Ericsson Canada, Canada
`{yosr.jarraya,makan.pourzandi}@ericsson.com`

Abstract. Security verification plays a vital role in providing users the needed security assurance in many applications. However, applying existing verification tools for runtime security enforcement may suffer from a common limitation, i.e., causing significant delay to user requests. The key reason to this limitation is that these tools are not specifically designed for runtime enforcement, especially in a dynamic and large-scale environment like clouds. In this paper, we address this issue by proposing a proactive framework, namely, *Proactivizer*, to transform existing verification tools into efficient solutions for runtime security enforcement. Our main idea is to leverage existing verification tools as black boxes and to proactively trigger the verification process based on dependency relationships among the events. As a proof of concept, we apply Proactivizer to several existing verification tools and integrate it with OpenStack, a popular cloud platform. We perform extensive experiments in both simulated and real cloud environments and the results demonstrate the effectiveness of Proactivizer in reducing the response time significantly (e.g., within 9 milliseconds to verify a cloud of 100,000 VMs and up to 99.9% reduction in response time).

Keywords: Proactive framework, runtime security enforcement, security verification.

1 Introduction

Security verification has been playing an important role in protecting a wide-range of IT infrastructures mainly due to its capability of providing security guarantee in diverse environments (e.g., networking [13], cyber physical systems [42] and software programs [29]). However, there is a paradigm shift in the concept of security solutions especially after the wide-adoption of clouds [1, 33]; where it is now essential to prevent a security breach to avoid its potentially unrecoverable damages and ensure continuous security guarantee; both of which can only be achieved through runtime security enforcement.

To that end, most existing security verification tools (e.g., [6, 8, 20, 22, 25, 30, 39]) for the cloud fall short in dealing with the dynamic and large-scale nature of clouds and offering runtime security enforcement. Specifically, those tools may cause significant delay in responses at runtime. This is not surprising since those tools are not specifically designed for runtime security enforcement of a large-scale dynamic environment like clouds; which also implies that modifying those tools for this purpose can be difficult. We further illustrate this limitation through a motivating example.

Motivating Example. The upper part of Figure 1 shows the typical response time when several existing verification tools (e.g., declarative logic programming (Datalog) [30, 19] and boolean satisfiability problem (SAT) [26], graph theory [6] and access control [20]) are utilized for runtime security policy enforcement, and highlights their common limitation. The lower part of the figure illustrates our key ideas to overcome that limitation, as detailed in the following.

– Even though these tools have been successful in diverse security applications, such as verifying virtual infrastructure [6, 30] and virtual network [19, 30], and enforcing access control policies [20, 26] in the cloud, all of them may suffer from a common limitation, i.e., causing a significant delay (e.g., 15 seconds to four minutes), when applied to continuously protecting the cloud through runtime security enforcement [6, 19, 20, 26, 30].
– The complexity of those tools and the fact that they were not initially designed for runtime enforcement imply that it could require tremendous amounts of time and effort to modify those tools for efficient runtime enforcement.
– Alternatively, our key idea is to take a blackbox approach, and proactively trigger those tools based on predicted events, such that we will already have the verification results ready, when the actual events arrive (e.g., conducting verification of the `add network` event in advance as soon as the `create VM` event occurs).

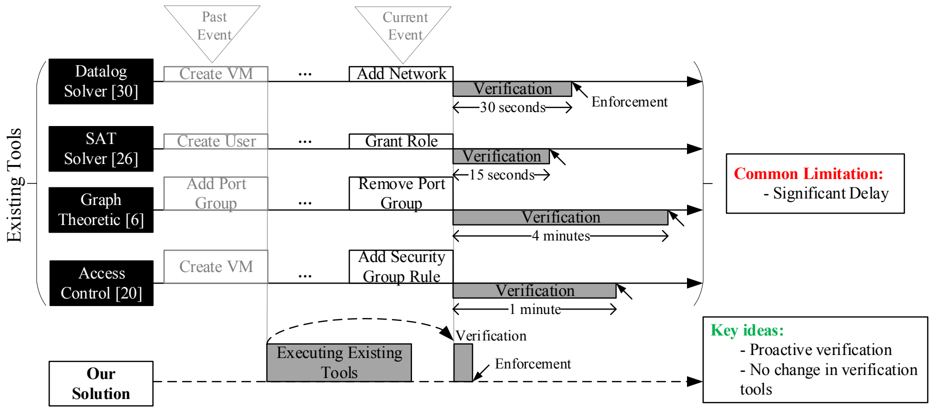


Fig. 1: Identifying the common issue in existing verification tools to offer runtime security enforcement and positioning our solution

More specifically, we propose a proactive verification framework, which considers those verification tools as blackboxes and transforms them into efficient solutions for runtime security enforcement. First, we develop a predictive model to capture various dependency relationships (e.g., probabilistic and temporal) to anticipate future events. Second, we design our proactive verification framework, namely, *Proactivizer*, with detailed methodology and algorithms. Third, as a proof of concept, we apply Proactivizer to several existing verification tools (i.e., Congress [30], Sugar [38], Weatherman [6] and Patron [20]), which adopt diverse verification methods, i.e., Datalog, SAT, graph-theoretic and access control, respectively. Fourth, we detail our implementation of the proposed framework based on OpenStack [31], and demonstrate how our system may be easily ported to other cloud platforms (e.g., Amazon EC2 [2] and Google GCP [12]). Finally, we evaluate our solution through extensive experiments with both synthetic and real data. The results confirm our framework can reduce the response time of those verification tools to a practical level (e.g., within nine milliseconds for 85.8% of the time).

In summary, our main contributions are as follows.

- As per our knowledge, we are the first to propose a generic proactive framework to transform existing verification tools into efficient solutions for runtime security enforcement. The main benefit of this framework is that it requires little modification to those tools.
- By applying the Proactivizer framework to a diverse collection of existing verification tools (e.g., Datalog, Satisfiability solver, access control and graph theoretic solution), we demonstrate its potential as a low cost solution for improving the efficiency of other existing verification tools in a wide range of applications (e.g., IoTGuard [7] and TopoGuard [15, 36]).
- As a proof of concept, we integrate our solution into OpenStack [31], a major cloud platform, and evaluate the effectiveness of our predictive model (e.g., up to 93% prediction accuracy) and the efficiency of our runtime security enforcement system (e.g., responding in maximum few milliseconds) using both synthetic and real data.

The remainder of the paper is organized as follows. Section 2 provides preliminaries. Section 3 presents our framework. Sections 5 and 6 provide the implementation details and experimental results, respectively. Section 7 discusses different aspects of this work. Section 8 summarizes related works. Section 9 concludes the paper.

2 Preliminaries

This section provides a background on dependency relationships (which will later be used to build predictive models in Section 3.2) and defines our threat model.

2.1 Dependency Relationships

We mainly consider three types of dependency relationships: structural, probabilistic and temporal. In the following, we explain them by taking cloud events as examples.

Structural Dependencies. Fig. 2(a) shows an example of structural dependencies in the cloud based on [23]. The structural dependency represents the relationships among cloud events, which are imposed by the cloud management platform (e.g., OpenStack [31]), e.g., a descendent node (or event) cannot occur before any of its ancestors.

Probabilistic Dependencies. Fig. 2(b) shows an example of the probabilistic dependencies as proposed in [24]. The probabilistic dependency indicates the behavioral pattern of cloud events, e.g., the probability of occurrences of a descendent node depends on the occurrences of its ancestors.

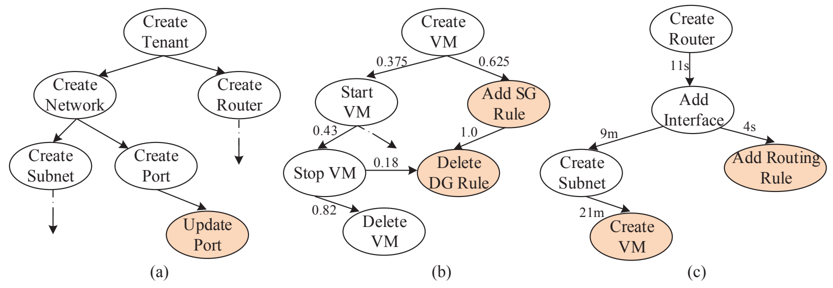


Fig. 2: Examples of (a) structural, (b) probabilistic, and (c) temporal dependency relationships among cloud events

Temporal Dependencies. Fig. 2(c) shows an example of temporal dependencies. This dependency indicates the time intervals between occurrences of different events, e.g., a descendent node occurs with an average interval from the occurrences of its ancestors.

2.2 Threat Model

In the remainder of this paper, we will focus on cloud platforms. We assume that the cloud management platforms: 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., [16, 3, 34]), and b) may have implementation flaws, misconfigurations and vulnerabilities that can be potentially exploited by malicious entities to violate security policies specified by cloud tenants. The cloud users including cloud operators and agents (on behalf of a human) may be malicious. Any threats directing from the cloud management operations is within the scope of this work. Therefore, any violation bypassing the cloud management interface is beyond the scope of this work. Also, our focus is not to detect specific attacks or intrusions, even though our framework may catch violations of specified security policies due to either misconfigurations or vulnerabilities. We assume that before our runtime approach, an initial verification is performed and potential violations are resolved. However, if our solution is added from the commencement of a cloud, obviously no prior security verification is required.

3 The Proactivizer Framework

This section presents the methodology of the Proactivizer framework.

3.1 Proactivizer Overview

Figure 3 shows an overview of our framework. There are three major steps of the Proactivizer framework: prediction, proactive verification and runtime enforcement. In Step 1 (detailed in Section 3.2), Proactivizer first extracts dependency relationships among cloud events from the historical data (e.g., logs), then builds a predictive model leveraging those dependencies and finally predicts future events utilizing the predictive model. In Step 2 (detailed in Section 3.3), to conduct proactive verification on the predicted future event, Proactivizer first prepares inputs related to that event for different verification tools, then executes those tools for verification and finally interprets the obtained verification results to prepare a watchlist (which is a list of allowed parameters for that future event). In step 3 (detailed in Section 3.4), for runtime security enforcement, Proactivizer intercepts critical events (which may cause potential violation of a security policy), then checks its parameters against the prepared watchlist and finally enforces the decision (e.g., allow or deny). In the following, we detail each step.

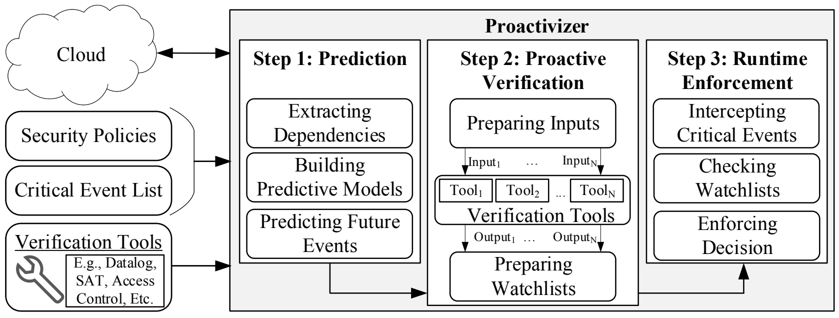
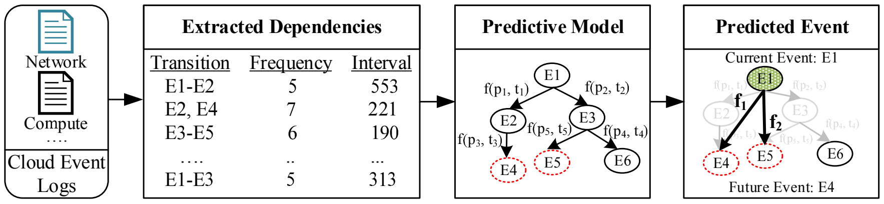


Fig. 3: A high-level overview of Proactivizer

3.2 Prediction

This section illustrates the prediction steps using an example and then elaborates them.

Example 1 Figure 4 shows an example of three major steps of building the predictive model. First, Proactivizer extracts dependencies (e.g., transitions, frequencies and intervals) from the cloud logs. The transition, `E1-E2`, indicates that event `E1` occurs before event `E2`. The corresponding frequency, `5`, means that transition `E1-E2` has appeared five times. The following interval, `553`, says that event `E2` occurs on average 553 seconds after the occurrence of event `E1`. Second, it builds the predictive model from those transitions; where the edge between events `E1` and `E2` indicates transition `E1-E2`, and the label on that edge, $f(p_1,t_1)$, is the prediction score (discussed later in this section) from the frequency (p_1) and interval (t_1). Third, it predicts critical events (`E4` or `E5`) using this model. From the current event, `E1`, it predicts event `E4` as a potential future event, because its prediction score, f_1, is greater than that of event `E5` (f_2). In *Example 2*, we show how to conduct the proactive verification of event `E4`



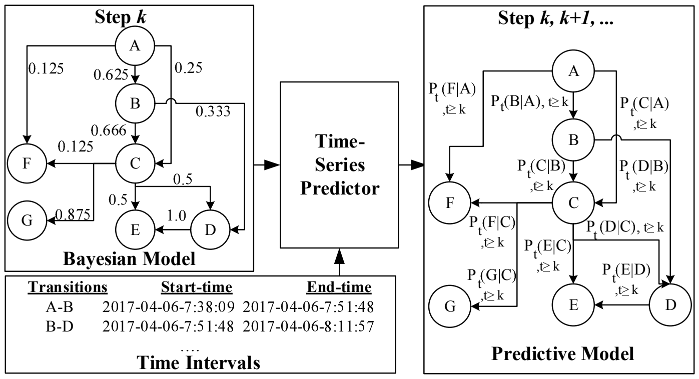
Transition	Frequency	Interval
E1-E2	5	553
E2, E4	7	221
E3-E5	6	190
....	..	...
E1-E3	5	313

Fig. 4: An example of different steps of the Proactivizer prediction

Log Processing. The processing of logs is mainly to prepare the data to build the predictive model. We describe its major steps (which are mostly inspired by LeaPS+ [27]) as follows. First, we parse raw cloud logs and label all fields of each log entry. Second, we identify the event type (i.e., generic operation name) based on the cloud-platform

API documentation. Third, we prepare the whole chain of identified events partitioned into transitions. Fourth, we obtain their frequencies and intervals. Finally, those transitions and their frequencies are utilized to obtain a probabilistic model (e.g., Bayesian network), which is then forwarded to build the predictive model. Note that Proactivizer periodically re-evaluates this Bayesian network for subsequent intervals.

Building the Predictive Model. Figure 5 shows the inputs and output of our time-series predictor [14] . The inputs are mainly the Bayesian networks obtained from the previous steps for different time periods, and the time intervals between event transitions. Then, we feed these intervals and the corresponding Bayesian network for a certain period to the time-series predictor for training. After the $(k-1)$th (where k is an integer number) step of training, we predict the conditional probability, $P_t(B|A)$, between events B and A at a given time t in the future. Thus, the predictor also measures the conditional probability for non-immediate transitions (e.g., $P_t(D|A)$). Our predictor follows a continuous training, part of which may update the value of $P_t(D|A)$ at the step k+1, and progressively updates the model. The effectiveness of our predictive model is evaluated in Section 6. we utilize this model to conduct the proactive verification as follows.



Transitions	Start-time	End-time
A-B	2017-04-06-7:38:09	2017-04-06-7:51:48
B-D	2017-04-06-7:51:48	2017-04-06-8:11:57
....		

Fig. 5: An excerpt input/output of the time-series predictor

3.3 Proactive Verification

This section illustrates the proactive steps using an example and then elaborates them.

Example 2 Figure 6 shows different steps of our proactive verification for the predicted event (`E4`) in *Example 1*. First, Proactivizer prepares the inputs for different verification tools to verify the predicted event, `E4`, with the current event parameter, `S1`. To that end, it identifies policy `P2` as one of the affected policies by critical event (CE) `E4`. Then, the input for tool $Tool_1$ is prepared to verify event `E4` with parameter `S1` against policy `P2`. Similarly, policies, $P3$ and $P27$, are prepared for tools, $Tool_2$ and $Tool_N$. Second, it executes verification tools, $Tool_1$, $Tool_2$ and $Tool_N$, to verify those policies (note that, while Proactivizer can support different tools at the same time, integrating many tools may inadvertently increase the system's complexity, which is why this feature is optional). Third, after the verification, Proactivizer interprets their outputs, and conclude that none of these policies will be breached, if the `E4` event with the `S1` parameter really occurs. Therefore, we add the parameters, `S1`, to the watchlist of the event `E4`. Similarly, we can show that for another parameter set, S3, the event E4 violates the policy P3 and hence, S3 is not added to the watchlist; which is further illustrated in *Example 3*.

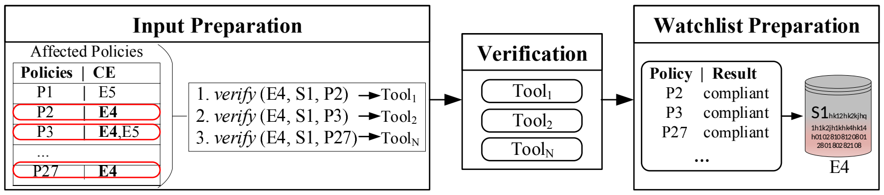


Fig. 6: An example of different steps of our proactive verification (where the predicted critical event (CE) is E4 and current event parameter is S1)

Figure 7 shows the major steps of our proactive verification. The figure also indicates the common steps for all verification tools that are integrated with the Proactivizer framework, and tool-specific unique steps.

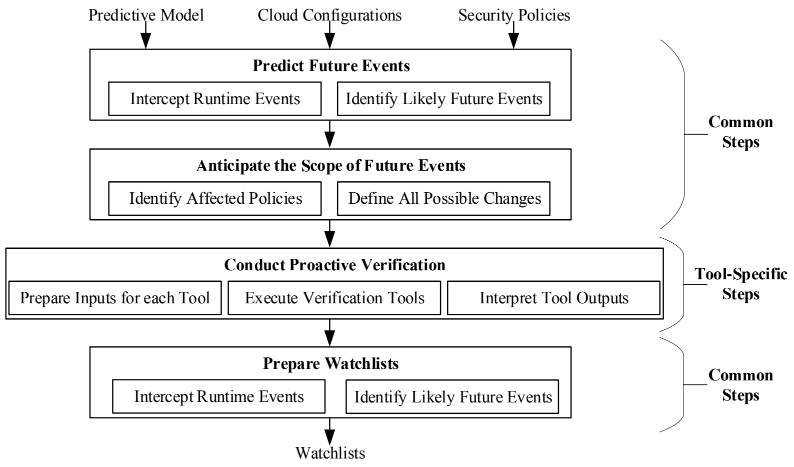

Predictive Model
Cloud Configurations
Security Policies
Predict Future Events
Intercept Runtime Events
Identify Likely Future Events
Anticipate the Scope of Future Events
Identify Affected Policies
Define All Possible Changes
Conduct Proactive Verification
Prepare Inputs for each Tool
Execute Verification Tools
Interpret Tool Outputs
Prepare Watchlists
Intercept Runtime Events
Identify Likely Future Events
Watchlists
Common Steps
Tool-Specific Steps
Common Steps

Fig. 7: The major steps of the proactive verification by the Proactivizer framework

Common Verification Steps. We elaborate on three major common steps as follows.

– *Predicting Future Events.* This step is to predict the future events from the current event using the predictive model (obtained in Section 2). To this end, Proactivizer first intercepts each event request sent to the cloud platform and obtains the detailed information (e.g., event type and its parameters). Second, it obtains the prediction scores (as discussed in Section 3.2) for each critical event from the intercepted event type. Third, it shortlists the predicted events which have greater prediction scores than the threshold (which is set by the users of Proactivizer).
– *Anticipating the Scope of Future Events.* This step is to anticipate the possible changes related to the predicted future event, as the event specifics (such as exact parameter values) are unknown at this point of time. To this end, Proactivizer first identifies the affected policies by that event from the list of security policies and corresponding critical events (as shown in Figure 6). Second, it anticipates the possible parameters for the future events by considering all available values of those parameters from the

current cloud configurations. These anticipated information will be later used by the tool-specific steps in Section 4.

– *Preparing Watchlists.* After the tool-specific steps (in Section 4), Proactivizer prepares the watchlist(s) from the verification results. Specifically, Proactivizer identifies the set of parameters for which a security violation is reported by any of the tools, and, includes the remaining anticipated parameters in the watchlist of the predicted event.

The tool-specific verification steps will be discussed in Section 4.

3.4 Runtime Security Enforcement

This step enforces security policies using the watchlists (obtained from the previous step) at runtime as follows. a) It holds the event execution whenever a critical event has occurred. b) It checks the parameters of the current event against its watchlists. c) Proactivizer only allows (or recommends to pass) the execution of the current event, if the parameters are in the watchlists. Otherwise, it denies the request.

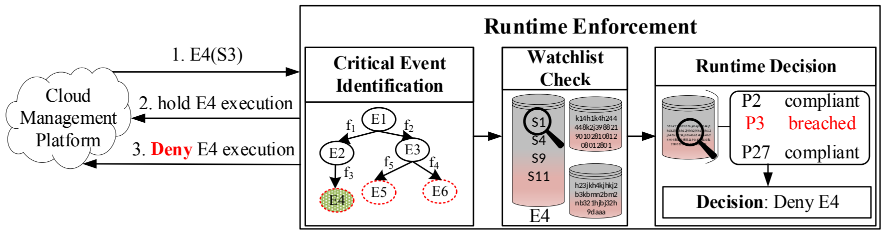


Fig. 8: An example of different steps of the Proactivizer runtime security enforcement

Example 3 Figure 8 shows an example of this step; where Proactivizer intercepts the critical event `E4` with the parameter set, `S3`. First, Proactivizer identifies that `E4` is a critical event, and hence, requests the cloud platform to hold the execution of the `E4` event request. Second, it searches `S3` in the watchlist of `E4`. The parameter set, `S3`, is not found in the watchlist, specifically, because `S3` breached policy `P3` in the previous step in *Example 2*. Finally, Proactivizer denies the current event request `E4`.

4 Proactivizer Applications

This section details the Proactivizer integration steps for three candidate applications.

4.1 Datalog-Based Security Verification

This section first provides the background on a Datalog-based security verification tool, namely, Congress [30], and then describes how we integrate Congress with Proactivizer.

Background. Congress [30] is an OpenStack [31] project to verify security policies for cloud infrastructures. Congress leverages declarative logical programming language (a.k.a. Datalog). We discuss its integration with Proactivizer as follows.

The details of Congress Integration. The major steps of this integration are to prepare Congress inputs and interpret its verification results. To prepare its inputs, our one-time efforts are to express policies in the Datalog format and identify required data and their sources for each security policy. Then, the runtime efforts are to populate the policy with the current event parameters and execute Congress with the prepared inputs. After Congress finishes the verification, Proactivizer analyzes Congress outputs to identify the parameter values for which a policy will be breached before preparing the watchlist (as in Section 3.3). We further show the Congress integration using an example as follows.

Example 4 In this example, we consider a security policy (provided by Congress [30]), which states that *"every network connected to a VM must be either public or private and owned by someone in the same group as the VM owner"*. For this policy, the `add network` event is one of the critical events, and we store the allowed network ID for each VM on the watchlist. We first express this policy as in Congress's format:

$$\begin{aligned}\texttt{error(vm)}:-\texttt{nova}:\texttt{instance(vm)},\texttt{nova}:\texttt{network(vm,network)},\\ \texttt{notneutron}:\texttt{public(network)},\texttt{nova}:\texttt{owner(vm,vmowner)},\\ \texttt{neutron}:\texttt{owner(network,netowner)},\texttt{notsame\_group(vmowner,netowner)}\\ \texttt{same\_group(x,y)}:-\texttt{group(x,g)},\texttt{group(y,g)}\\ \texttt{group(x,g)}:-\texttt{keystone}:\texttt{group(x,g)}\\ \texttt{group(x,g)}:-\texttt{ad}:\texttt{group(x,g)}\end{aligned} \tag{1}$$

For the explanation of this expression, we refer the readers to [30]. The data sources are the configurations from different services (e.g., Nova and Neutron) of OpenStack.

At runtime, the common steps in Section 3.3 provides the predicted event `add network`, the parameter of current event, `VM1`, and all the network IDs, `N1`, `N2` and `N3`, for that tenant. As a part of the Congress-specific effort, Proactivizer updates the nova:network table on the simulated environment of Congress as if these three networks are added to the `VM1` VM, and executes Congress to verify the above-mentioned policy. The result of Congress is formatted as `error(VM1,N3)`, which indicates that adding network `N3` to VM `VM1` will violate the policy. Using this interpretation, the final common step (in Section 3.3) prepares the watchlist.

4.2 SAT-Based Security Verification

This section first provides the background on a satisfiability (SAT) solver, namely, Sugar [38], and then describes how we integrate Sugar with Proactivizer.

Background. Sugar [38] is a SAT solver, which expresses policies as a constraint satisfaction problem (CSP). If all constraints are satisfied, then Sugar returns SAT (which indicates a policy violation). We discuss its integration with Proactivizer as follows.

The details of Sugar Integration. To prepare Sugar inputs, our one-time efforts are to express policies in the CSP format and identify the required data and their sources. Then, our runtime efforts are to populate that CSP policy with both current and predicted event parameters and execute Sugar to verify that policy. After the verification, Proactivizer interprets Sugar's outputs to prepare the watchlist to avoid any policy violation. We further show the Congress integration using an example as follows.

Example 5 For this example, we consider a security policy (provided by [26]), which states that *"a user must not hold any role from another domain"* (where domain is a collection of tenants). Here, the `grant role` event is a critical event, and the allowed roles are in the watchlist for each domain. Proactivizer first expresses the policy as:

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

Here, *u* is the user, *d* is the domain, *t* is the tenant and *r* is the role. For the explanation of each constraint, we refer the readers to [26]. For this policy, we collect user, role, tenant and domain, from the identity management service (Keystone) of OpenStack.

At runtime, similarly as in *Example 4*, the predicted event is `grant role`, the current event parameter is `u1` user ID and possible roles are `r1`, `r2` and `r3`. Then, Proactivizer instantiates the policy in *Equation (2)* with each user-role pair, e.g., (`u1`, `r1`), (`u1`, `r2`) and (`u1`, `r3`), and executes Sugar to verify the policy. The result of Sugar provides the pair, (`u1, r3`), for which it gets a SAT result; which means that granting role `r3` to user `u1` will violate the policy, and hence, role `r3` will not be added to the watchlist.

4.3 Access Control Policy Verification

This section first provides the background on an access control tool, namely, Patron [20], and then describes how we integrate Patron with Proactivizer.

Background. Patron [20] is an access control policy verification solution for clouds. To that end, Patron verifies each runtime event request against a list of access control policies defined by tenants. We discuss its integration with Proactivizer as follows.

The details of Patron Integration. At runtime, Proactivizer expresses the predicted event in Patron's format, which includes event type, caller of an event and requested resources, and executes Patron to verify that event. After Patron's verification, Proactivizer checks Patron's decision to prepare the watchlist. We further explain this step using an example as follows.

Example 6 In this example, we consider a security policy stating that *"a tenant admin only can add security group rules to the VMs of the same tenant"*. At runtime, the predicted event is `add security group rule` and the caller of the current event is `tenant1-admin`. Proactivizer prepares the Patron inputs as: {input 1: {`add security group rule`, `user: tenant1-admin`, `VM1`} and input 2: {`add security group rule`, `user: tenant1-admin`, `VM2`}}. The results of Patron are {input 1: `allow`} and {input 2: `deny`}, respectively. Therefore, it only adds the VM1 to the watchlist.

5 Implementation

This section presents the high-level architecture of Proactivizer, and then details its integration into OpenStack [31], a popular cloud platform.

Architecture. There are five major components in our solution (Figure 9). i) The data collector collects logs and configurations from the cloud platform. ii) The predictive model builder is mainly to build the predictive model using Bayesian network and time-series. iii) The interceptor interacts with the cloud platform at runtime. iv) The proactive verifier mainly provides the interface to plug various verification tools and

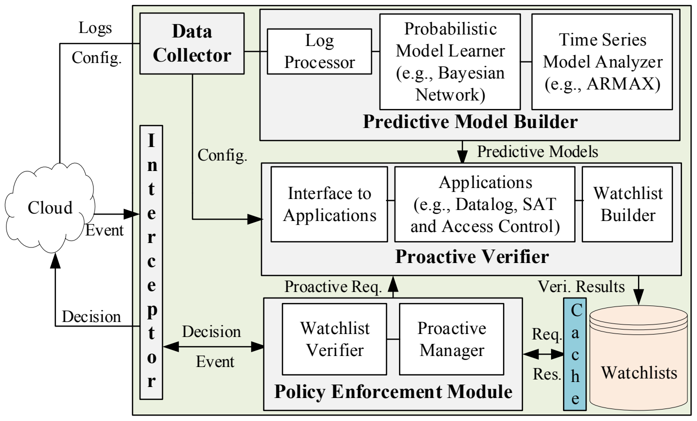


Fig. 9: A high-level architecture of our solution

builds watchlists. v) The policy enforcement module enforces the runtime decision on the cloud platform. Algorithm 1 shows the functionalities of these modules.

Implementation Details. In the predictive model builder, to process raw logs, we first use Logstash [10], a data processing engine, and then utilize our own scripts to further obtain event sequences and their corresponding frequency and intervals. The resulted set of sequences is the input dataset to a Python Bayesian Network toolbox[1]. Afterwards, the obtained Bayesian networks are provided to a time-series predictor, ARMAX [14], which is a widely used method in prediction of stochastic processes in various fields.

Algorithm 1: Proactivize (*CloudOS*, *Policies*, *Tools*)

```
1: procedure BUILDPREDICTIVEMODEL(Logs)
2:     for each timePeriod $t_i \in T$ do
3:         Sequence[] = identifySequence($Logs_{t_i}$)
4:         bayesianNetwork = prepareBN(Sequence[])
5:         predictiveModel = buildTimeSeriesModel(bayesianNetwork, Sequence[].interval)

6: procedure INTERCEPTEVENT(interceptedEvent , Policies)
7:     for each policy $p_i \in$ Policies do
8:         if interceptedEvent $\in$ Policy.critical-events then
9:             decision = verifyWL(interceptedEvent)
10:            return decision
11:        else
12:            manageProactive(interceptedEvent)

13: procedure VERIFYWL(critical-event)
14:     if critical-event.params $\in$ critical-events.WL then
15:         decision= "allow"
16:     else
17:         decision= "deny"
18: procedure MANAGEPROACTIVE(interceptedEvent)
19:     for each critical-event $c_i \in$ Policies.critical-events do
20:         distance = measureDistance(predictiveModel, $c_i$, interceptedEvent)
21:         if distance = $c_i$.threshold then
22:             policy[]=affectPolicy($c_i$)
23:             verifyProactive($c_i$, interceptedEvent.params, policy[])

24: procedure VERIFYPROACTIVE(critical-event, params, Policies)
25:     for each policy $p_i \in$ Policies do
26:         input = prepareInput (Tools, $p_i$, critical-event, params)
27:         result = interpretResult(verify (Tools, input))
```

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

The interceptor is implemented as a middleware so that it intercepts each request to the OpenStack services (similarly as in [37, 20]). The proactive verifier currently integrates three candidate applications (Congress [30], Patron [20] and Sugar [38]) with Proactivizer. The watchlists are mainly stored in a MySQL database. In addition, we implement a cache as memory-mapped file system (mmap) in Python (similarly as in [20]); this cache stores recent watchlist queries to accelerate the decision mechanisms.

6 Experiments

This section first explains our experimental settings, and then presents the results using both synthetic and real data.

6.1 Experimental Settings

Our testbed cloud is based on OpenStack version Mitaka. There are one controller node and up to 80 compute nodes, each has a dual-core CPU and 2GB memory with the Ubuntu 16.04 server. Based on a recent survey [32] on OpenStack, we simulate an environment with maximum 1,000 tenants and 100,000 VMs. There are four synthetic datasets, DS1-DS4, where we vary the number of VMs from 10,000 to 100,000 and number of tenants from 1,000 to 10,000, simultaneously. 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 telecommunication vendors; which contains 1.6 GB text-based logs with 128,264 relevant entries (and 400 uniques records after processing) for the period of 500 days. We repeat each experiment at least 100 times.

6.2 Experimental Results

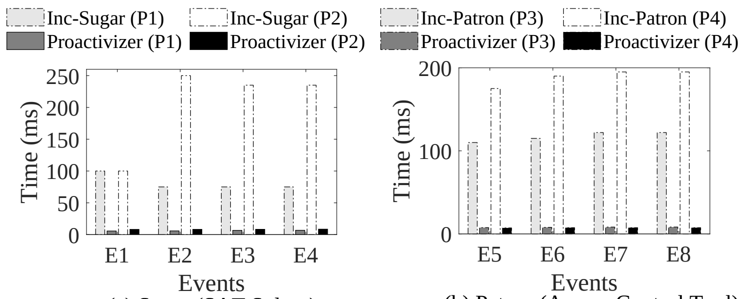
In the following, we present our experimental results.

Efficiency Improvement in Proactivizer Applications. The objective of the first set of experiments is to demonstrate the efficiency improvement resulted from the Proactivizer integration with different applications. Table 1 summarizes the response time of three candidate applications (i.e., Congress [30] (a Datalog solution), Patron [20] (an access control tool) and Sugar [38] (SAT solver)) before and after the integration with Proactivizer for four different datasets (DS1-DS4) and four different policies (P1-P4). The response time of those applications without the Proactivizer integrations results from five seconds to 103 seconds. On the other hand, after the Proactivizer framework integration, the response time of these tools remains within nine milliseconds. In summary, our framework significantly improves the response time of these tools (e.g., around 99.9% reduction on average). Furthermore, in Figure 10, we compare the response time of incremental implementations of both Sugar [26] (Inc-Sugar) and Patron [20] (Inc-Patron) for different events with and without the integration of Proactivizer. Specifically, Figure 10(a) shows that the response time of Sugar has been reduced to around 8 ms from 200 ms as an effect of Proactivizer. Figure 10(b) shows the similar nature of response time improvement in Patron. On average, Proactivizer reduces the response time of Inc-Sugar and Inc-Patron by 93.74% and 94.64%, respectively. We further check the effect

		Sugar [38]			Patron [20]		Congress [30]		
Dataset	**Proactivizer**	P1	P2	P3	P1	P2	P2	P3	P4
DS1	**without (in s)**	5.3	6.6	96.5	12.3	60.1	20.1	27.2	30
	with (in ms)	5.6	8.1	7.5	5.6	8.1	8.1	7.5	7.1
DS2	**without (in s)**	6.5	7.2	102.3	15.9	67.1	21.1	29.2	35
	with (in ms)	5.8	8.2	7.8	5.8	8.2	8.2	7.8	7.4
DS3	**without (in s)**	9.4	10.5	109.5	21.9	75.3	25.4	31.9	35.7
	with (in ms)	6.6	8.3	8.1	6.6	8.3	8.3	8.1	7.4
DS4	**without (in s)**	15.3	16.4	118.7	29.5	87.9	30	34.2	39.1
	with (in ms)	6.8	8.3	8.2	6.8	8.3	8.3	8.2	7.4
Average Improvement (%)		**99.93**			**99.97**		**99.96**		

Table 1: The response time before (in seconds) and after (in milliseconds) the integration of Proactivizer to Sugar (a SAT solver), Congress (a Datalog-based tool), and Patron (an access control tool) for the runtime security enforcement of different policies (P1, P2, P3 and P4) for different datasets, DS1-DS4. Here, P1: Common Ownership, P2: Minimum Exposure, P3: No Cross-Tenant Port and P4: No Bypass policies

of our cache implementation, and observe that the response time can be reduced to even less than one millisecond (which is shown in Appendix B for the space constraint).



(a) Sugar (SAT Solver) (b) Patron (Access Control Tool)

Fig. 10: The response time (in milliseconds) before and after the integration of Proactivizer for verifying (a) the Common Ownership (P1) and Minimum Exposure (P2) policies by Inc-Sugar (using SAT) and (b) the No Cross-Tenant Port (P3) and No Bypass (P4) policies by Inc-Patron (using access control) for different events and our largest dataset. Here, E1: grant role, E2: delete role, E3: delete user, E4: delete tenant, E5: create VM, E6: start VM, E7: add security group rule and E8: delete security group rule

Effectiveness of our Predictive Model. The second set of experiments is to show the effectiveness of our predictive model in terms of prediction match/error and fitting to the real observation. Figure 11 shows a comparison between our predictive model (based on the ARMAX function) and the state-of-art dependency model (based on Bayesian network (BN)) [24] for different threshold values. Here, the prediction match rate refers to the percentage of time proactive verification results are useful, and the prediction error rate is its inverse. Specifically, Figure 11(a) shows that our model ensures the best response time on average 85.8% of the time. In the best case, it can reach up to 93% of prediction match with selective threshold values. Figure 11(b) shows the superior fitting capability of ARMAX over BN; where we train the ARMAX model for 24 hours and the resulted model is used in prediction for the next 24 hours. As illustrated in the magnified window, both measurements in BN (dashed yellow lines) are lagging behind the real dataset (in blue). On the other hand, our trained ARMAX model (in red) can predict the time series more accurately (85% fit). These results strongly support the effectiveness of our ARMAX model.

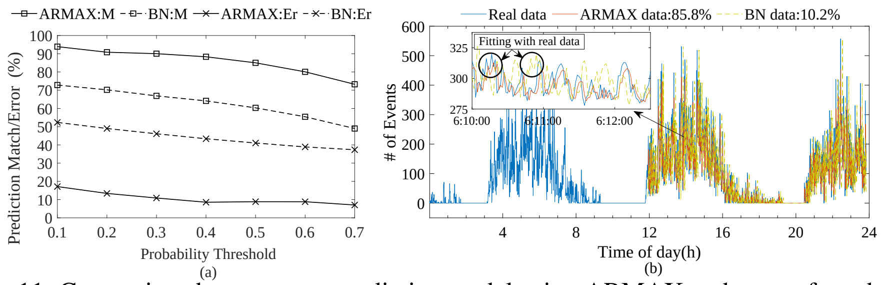


Fig. 11: Comparison between our predictive model using ARMAX and state-of-art dependency model (based on Bayesian network (BN)) [24] in terms of (a) the percentage of prediction match(M)/error(Er) (b) the percentage of fitting with the real data

Experiments with Real Cloud. We conduct similar experiments on the real data. Due to the significantly smaller number of observations (i.e., 400 unique records), the ARMAX model shows less superiority (up to 65%) over Bayesian network and in few cases is inferior. The real effectiveness of our prediction model is shown through the relatively larger datasets (in Figure 11).

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
Bayesian Network 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
Bayesian Network 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 2: Effectiveness of ARMAX vs. Bayesian network for real data (with 400 records)

Overall these results show that the response time with our framework can be less than one millisecond in the best case (with cache), and in the worst case (for an incorrect prediction), Proactivizer will have no effect on those applications. However, for most cases (around 85.8% time), Proactivizer can keep the response time of these applications within nine milliseconds.

7 Discussion

Additional Efforts to Add a New Verification Tool. Proactivizer is a framework to plug different verification tools. Therefore, we design the Proactivizer framework in a manner that most steps remain tool-agnostic (as shown in Section 3.3). As a result, to add a new tool with Proactivizer, the main efforts are to prepare the inputs specific to that tool and interpret its results (as shown in Section 4).

Choosing the Value of the Threshold. As described in Section 3 and evaluated in Section 6, our solution schedules the computation based on a threshold probability. As shown in Figure 11, lower values of threshold result in better prediction match. However, the prediction error also increases in such cases. Therefore, an optimal threshold value has to be chosen based on the tenant's need and experiences.

Reliance on the List of Critical Events. Like other existing solutions (e.g., Congress [30] and Weatherman [6]), our solution currently relies on manual identification of a list of critical events as inputs. However, our preliminary study shows that this identification process can be at least semi-automated by adopting a feedback module, which progressively can update and complete this list leveraging retroactive auditing tools (e.g., [22, 25]). We report the detailed results of the study in our future work.

Choice of Prediction Function. Proactivizer leverages the predictive model to proactively trigger those verification tools. Therefore, the accuracy of our prediction function might be critical in achieving the best performance of Proactivizer. In this paper, we explore Bayesian network and ARMAX time series function and show the superiority of ARMAX function for our purpose (through experimental results in Section 6).

Supported Security Policies. Proactivizer is a general framework in which various verification tools can be plugged to verify a wide range of security policies. Therefore, potentially Proactivizer could support a wide range of security policies. To demonstrate this generality, we have so far integrated three verification tools with three totally different policy languages and specifications. In the near future, we intend to extend applying Proactivizer beyond the cloud environment, such as, in SDN and IoT.

Adapting to Other Cloud Platforms. Even though our current implementation is for OpenStack, the Proactivizer design is platform-agnostic. Therefore, Proactivizer can be adapted to other cloud platforms (e.g., Amazon EC2 [2] and Google GCP [12]) with a one-time effort for implementing a platform-specific interface. To this end, we provide a concrete guideline to adapt Proactivizer for other cloud platforms in Appendix A.

8 Related Work

Table 3 summarizes the comparison between existing works and Proactivizer. The first and second columns list existing works and their verification methods. The next three columns indicate different cloud layers, such as user-level, virtual infrastructure at the tenant level (T) and virtual network at the cloud service provider (CSP) level. The next three columns compare these works based on the adopted approaches. The next columns compare them according to different features, i.e., runtime enforcement capability, considering verification tools as blackboxes, serving as a general-purpose solution, supporting expressive policy languages and offering automated inputs. Note that the (○) symbol is for the Run. Enforcement column indicates that the corresponding work offers runtime enforcement with significant delay, and an (N/A) in the Blackbox column means that the corresponding solution is not utilizing any so-called verification tool.

In summary, Proactivizer differs from the existing works as follows. Firstly, Proactivizer is the first proactive framework, which leverages existing tools as a blackbox and transforms them into efficient solutions for runtime security enforcement. Secondly, Proactivizer can potentially support a wide range of security policies due to its inherited expressiveness from the integrated tools, and serve as a general-purpose framework.

Retroactive and Intercept-and-Check Approach. Unlike our work, retroactive verification approach (e.g., [19, 22, 25, 40, 41, 39, 8]) can detect violations only after they occur, which may expose the system to high risks. Existing intercept-and-check approaches (e.g., [6, 30, 20, 26]) perform major verification tasks while holding the event instances blocked. As a result, these works tend to cause significant delay to the user requests; e.g., Weatherman [6] reports a four-minute delay to verify a mid-sized cloud. In contrast, our framework transforms these intercept-and-check approaches into an

Proposals	Methods	Layers			Approaches			Features				
		User-level	Virtual Infr.(T)	Virtual Net. (CSP)	Retroactive	Intercept-and-Check	Proactive	Run. Enforcement	Blackbox	General Purpose	Expressive	Automated
Patron [20]	Access Control	●	-	-	-	●	-	○	N/A	-	-	●
Majumdar et al. [26]	SAT Solver	●	-	-	-	●	-	-	●	-	●	●
Madi et al. [21]	SAT Solver	-	●	●	●	-	-	-	●	-	●	●
Weatherman (V1) [6]	Graph-theoretic	-	●	-	-	●	-	○	-	-	●	●
Weatherman (V2) [6]	Graph-theoretic	-	●	-	-	-	●	●	-	-	●	-
Congress (V1) [30]	Datalog	●	●	●	-	●	-	○	-	-	●	●
Congress (V2) [30]	Datalog	●	●	●	-	-	●	●	-	-	●	-
NoD [19]	Datalog	-	-	●	●	-	-	-	●	-	●	●
LeaPS [24]	Custom + Bayesian	●	●	●	-	-	●	●	N/A	-	-	●
Proactivizer	-	●	●	●	-	-	●	●	●	●	●	●

Table 3: Comparing existing works with Proactivizer. The symbols (●), (-) and N/A mean supported, not supported, and not applicable, respectively. Note that, symbol ○ is used for the solutions which support runtime enforcement with significant delay.

efficient solution for runtime security enforcement (as reported in Section 6). There exist other intercept-and-check solutions (e.g., TopoGuard [15], TopoGuard+ [36] and IoTGuard [7]) for SDN and IoT environments. These works can potentially be the applications of the Proactivizer framework to further improve their response time.

Proactive Approach. There exist few proactive works (e.g., [6, 30, 24, 23, 44]) for clouds. Weatherman [6] and Congress [30] verify security policies on a future change plan using the graph-based and Datalog-based model proposed in [5, 4], respectively. Unlike our automated predictive model, those works rely on manual inputs of future plan. PVSC [23] proactively verifies security compliance by utilizing the static patterns in dependency models. PVSC [23] and LeaPS [24] are both customized for specific security policies and environment. Whereas, Proactivizer is designed to support a wide-range of security policies in diverse environments. In addition, Foley et al. [11] propose an algebra for anomaly-free firewall policies for OpenStack. Many state-based formal models (e.g., [35, 17, 18, 9]) are proposed for program monitoring. Our work differs from them as we target in providing a generic proactive framework for plugging various verification tools, and these works potentially can be the applications of Proactivizer.

9 Conclusion

In this paper, we proposed Proactivizer, a generic proactive framework to transform existing verification tools into efficient solutions for runtime security enforcement. To this end, we leveraged the existing tools as blackboxes and proactively triggered the verification process based on the dependency relationships among the events. As a proof of concept, we applied Proactivizer to several existing verification tools (e.g., SAT solver, Datalog-based tool and access control tool) and integrated it with OpenStack, a widely used cloud platform. Through our extensive experiments in both simulated and real cloud environments, we demonstrated the effectiveness of our framework in reducing response time significantly (e.g., within nine milliseconds for 85.8% of the time). As future work, we intend to conduct a cost analysis of our proactive verification for different threshold values to help users in choosing more appropriate threshold value. Also, we plan to explore other time series functions to identify the best option.

References

1. J. Aikat, A. Akella, J. S. Chase, A. Juels, M. Reiter, T. Ristenpart, V. Sekar, and M. Swift. Rethinking security in the era of cloud computing. *IEEE Security & Privacy*, 15(3), 2017.
2. Amazon. Amazon virtual private cloud. Available at: `https://aws.amazon.com/vpc`, last accessed on: February 14, 2018.
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 *European Symposium on Research in Computer Security (ESORICS)*, pages 392–415. Springer, 2011.
5. S. Bleikertz, C. Vogel, and T. Groß. Cloud Radar: near real-time detection of security failures in dynamic virtualized infrastructures. In *Proceedings of the 30th annual computer security applications conference (ACSAC)*, pages 26–35. ACM, 2014.
6. S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim. Proactive security analysis of changes in virtualized infrastructures. In *Proceedings of the 31st annual computer security applications conference (ACSAC)*, pages 51–60. ACM, 2015.
7. Z. B. Celik, G. Tan, and P. McDaniel. IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT. In *Proceedings of 2019 Annual Network and Distributed System Security Symposium (NDSS'19)*, February 2019.
8. F. Doelitzscher, C. Fischer, D. Moskal, C. Reich, M. Knahl, and N. Clarke. Validating cloud infrastructure changes by cloud audits. In *Eighth World Congress on Services (SERVICES)*, pages 377–384. IEEE, 2012.
9. E. Dolzhenko, J. Ligatti, and S. Reddy. Modeling runtime enforcement with mandatory results automata. *International Journal of Information Security*, 14(1):47–60, 2015.
10. Elasticsearch. Logstash. Available at: `https://www.elastic.co/products/logstash`, last accessed on: February 14, 2018.
11. S. N. Foley and U. Neville. A firewall algebra for OpenStack. In *Conference on Communications and Network Security (CNS)*, pages 541–549. IEEE, 2015.
12. Google. Google cloud platform. Available at: `https://cloud.google.com`, last accessed on: February 14, 2018.
13. H. Hamed, E. Al-Shaer, and W. Marrero. Modeling and verification of ipsec and vpn security policies. In *13th IEEE International Conference on Network Protocols (ICNP'05)*, pages 10–pp. IEEE, 2005.
14. J. D. Hamilton. Time series analysis. *Economic Theory. II, Princeton University Press, USA*, pages 625–630, 1995.
15. S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *Proceedings of 2015 Annual Network and Distributed System Security Symposium (NDSS'15)*, February 2015.
16. M. Li, W. Zang, K. Bai, M. Yu, and P. Liu. Mycloud: supporting user-configured privacy protection in cloud computing. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, pages 59–68. ACM, 2013.
17. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):19, 2009.
18. J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *European Symposium on Research in Computer Security (ESORICS)*, pages 87–100. Springer, 2010.
19. N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, pages 499–512, 2015.

20. Y. Luo, W. Luo, T. Puyang, Q. Shen, A. Ruan, and Z. Wu. OpenStack security modules: A least-invasive access control framework for the cloud. In *IEEE 9th International Conference on Cloud Computing (CLOUD)*, 2016.
21. T. Madi, Y. Jarraya, A. Alimohammadifar, S. Majumdar, Y. Wang, M. Pourzandi, L. Wang, and M. Debbabi. ISOTOP: Auditing virtual networks isolation across cloud layers in OpenStack. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):1, 2018.
22. T. Madi, S. Majumdar, Y. Wang, Y. Jarraya, M. Pourzandi, and L. Wang. Auditing security compliance of the virtualized infrastructure in the cloud: Application to openstack. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 195–206. ACM, 2016.
23. 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 *European Symposium on Research in Computer Security (ESORICS)*, pages 47–66. Springer, 2016.
24. S. Majumdar, Y. Jarraya, M. Oqaily, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi. Leaps: Learning-based proactive security auditing for clouds. In *European Symposium on Research in Computer Security (ESORICS)*, pages 265–285. Springer, 2017.
25. 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 *7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 58–65. IEEE, 2015.
26. S. Majumdar, T. Madi, Y. Wang, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi. User-level runtime security auditing for the cloud. *IEEE Transactions on Information Forensics and Security*, 13(5):1185–1199, 2018.
27. S. Majumdar, A. Tabiban, Y. Jarraya, M. Oqaily, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi. Learning probabilistic dependencies among events for proactive security auditing in clouds. *Journal of Computer Security*, 27(2):165–202, 2019.
28. Microsoft. Microsoft Azure virtual network. Available at: `https://azure.microsoft.com`, last accessed on: February 14, 2018.
29. N. Nitta, Y. Takata, and H. Seki. An efficient security verification method for programs with stack inspection. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 68–77. ACM, 2001.
30. OpenStack. OpenStack Congress, 2015. Available at: `https://wiki.openstack.org/wiki/Congress`, last accessed on: February 14, 2018.
31. OpenStack. OpenStack open source cloud computing software, 2015. Available at: `http://www.openstack.org`, last accessed on: February 14, 2018.
32. OpenStack. OpenStack user survey, 2018. Available at: `https://www.openstack.org/user-survey/2018-user-survey-report/`, last accessed on: Apr 24, 2019.
33. K. Ren, C. Wang, and Q. Wang. Security challenges for the public cloud. *IEEE Internet Computing*, 16(1):69–73, 2012.
34. N. Schear, P. T. Cable II, T. M. Moyer, B. Richard, and R. Rudd. Bootstrapping and maintaining trust in the cloud. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016.
35. F. B. Schneider. Enforceable security policies. *Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.
36. R. Skowyra, L. Xu, G. Gu, T. Hobson, V. Dedhia, J. Landry, and H. Okhravi. Effective topology tampering attacks and defenses in software-defined networks. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*, June 2018.

37. A. Tabiban, S. Majumdar, L. Wang, and M. Debbabi. Permon: An openstack middleware for runtime security policy enforcement in clouds. In *Proceedings of the 4th IEEE Workshop on Security and Privacy in the Cloud (SPC 2018)*, June 2018.
38. N. Tamura and M. Banbara. Sugar: A CSP to SAT translator based on order encoding. In *Proceedings of the Second International CSP Solver Competition*, pages 65–69, 2008.
39. K. W. Ullah, A. S. Ahmed, and J. Ylitalo. Towards building an automated security compliance tool for the cloud. In *12th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1587–1593. IEEE, 2013.
40. C. Wang, S. S. Chow, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE transactions on computers*, 62(2):362–375, 2013.
41. 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*, 12(4):940–952, 2017.
42. D. C. Wardell, R. F. Mills, G. L. Peterson, and M. E. Oxley. A method for revealing and addressing security vulnerabilities in cyber-physical systems by modeling malicious agent interactions with formal verification. *Procedia computer science*, 95:24–31, 2016.
43. WSGI. Middleware and libraries for WSGI, 2016. Available at: `http://wsgi.readthedocs.io/en/latest/libraries.html`, last accessed on: February 15, 2018.
44. S. S. Yau, A. B. Buduru, and V. Nagaraja. Protecting critical cloud infrastructures with predictive capability. In *8th International Conference on Cloud Computing (CLOUD)*, pages 1119–1124. IEEE, 2015.

A Guideline to Adapt to Other Cloud Platforms

Our solution interacts with the cloud platform (e.g., while collecting logs and intercepting runtime events) through two modules: pre-processor and interceptor. These two modules require to interpret implementation- specific event instances, and intercept runtime events. First, to interpret platform-specific event instances to generic event types, we currently maintain a mapping of the APIs from different platforms. Table 4 enlists some examples of such mappings. Second, the interception mechanism may require to be implemented for each cloud platform. In OpenStack, we leverage WSGI middleware to intercept and enforce the proactive auditing results so that compliance can be preserved. Through our preliminary study, we identified that almost all major platforms provide an option to intercept cloud events. In Amazon using AWS Lambda functions, developers can write their own code to intercept and monitor events. Google GCP introduces GCP Metrics to configure charting or alerting different critical situations. Our understanding is that our solution can be integrated to GCP as one of the metrics similarly as the *dos_intercept_count* metric, which intends to prevent DoS attacks. The Azure Event Grid is an event managing service from Azure to monitor and control event routing which is quite similar as our interception mechanism. Therefore, we believe that our solution can be an extension of the Azure Event Grid to proactively audit cloud events. Tables 4 and 5 represent the necessary mapping to be used for extending our approach from OpenStack to other cloud platforms. The rest modules of our solution deal with the platform-independent data, and hence, the next steps in our solution are platform-agnostic.

Generic Event Type	**OpenStack [31]**	**Amazon EC2-VPC [2]**	**Google GCP [12]**	**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`	N/A	`az network nsg delete`

Table 4: Mapping event APIs from different cloud platforms to generic event types

Cloud Platform	**Interception Support**
OpenStack	*WSGI Middleware* [43]
Amazon EC2-VPC	*AWS Lambda Function* [2]
Google GCP	*GCP Metrics* [12]
Microsoft Azure	*Azure Event Grid* [28]

Table 5: Interception supports to adopt our solution in major cloud platforms

B Performance of the Cache Implementation

Figure 12 illustrates the response time in case there is a cache hit (when runtime parameters is found in the implemented cache memory) and the additional delay for a cache miss (when requested parameters is not in the cache memory) for Patron and Congress, respectively. In Figure 12(a), for different sizes of cache, we observe a quasi constant response time (which is less than one millisecond) for Patron with our framework, and an additional delay for a cache miss of up to four milliseconds. Figure 12(b) shows the results of similar experiment for Congress with our framework; where a cache hit causes further improvement on the response time, but a cache miss may cause up to 137 milliseconds of delay. Overall the results show the response time can be even less than one millisecond at the best case, and at the worst case (when the prediction is incorrect), Proactivizer will have no effect on those applications. However, for most cases (around 85.5% time), Proactivizer can keep their response time within ten milliseconds .

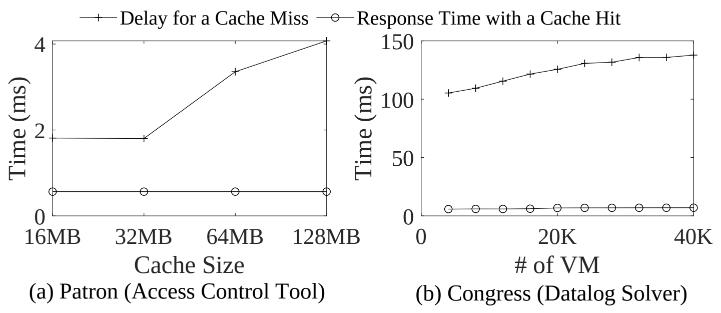


Fig. 12: The average response time for a cache hit and delay for a cache miss for (a) Patron (access control tool) and (b) Congress (Datalog solver), while varying the size of the cache and number of VMs, respectively