PerfSPEC: Performance Profiling-based Proactive Security Policy Enforcement for Containers

Hugo Kermabon-Bobinnec, Sima Bagheri,
Mahmood GholipourChoubeh, Suryadipta Majumdar, Yosr Jarraya, Lingyu Wang and Makan Pourzandi

Abstract—Container environments provide cloud native applications with scalability, flexibility, and portable support. As a popular container orchestrator, Kubernetes facilitates automatic deployment and maintenance of a large number of containerized applications. However, potential misconfigurations, vulnerabilities, or implementation flaws may empower attackers to exploit the Kubernetes cluster. Although existing solutions such as runtime security policy enforcement may prevent an attack, they can be inefficient in large scale container environments. In this paper, we propose a performance profiling-based proactive security policy enforcement solution, namely, PerfSPEC. First, we accelerate the proactivization of policies (which typically requires significant manual effort) by proposing to profile and rank existing policies according to their induced overhead. This allows us to better focus our efforts and greatly improve the overall response time (e.g., by 98% in contrast to less than 49%). Then, we address the performance limitations of existing solutions by leveraging learning-based approaches to predict future events and compute their verification results in advance. As a result, PerfSPEC achieves a viable response time (e.g., less than 10 ms in contrast to 600 ms with one of the most popular existing approaches) even for large container environments (up to 800 Pods).

Index Terms—Cloud Security, Proactive Security, Kubernetes, Policy Enforcement, Containers.

1 Introduction

Containers are becoming a standard for delivering cloud services while ensuring scalability, reliability, and observability [1]. The increased popularity of containerizing applications (e.g., in 5G mobile networks [2]) made them the main target of various security attacks exploiting misconfigurations and/or vulnerabilities. In this context, container orchestrators (e.g., Kubernetes [3]) are playing a central role in automating the deployment, scaling, and management of large numbers of containerized applications. Thus, if an adversary succeeds to compromise the orchestrator, (s)he can control the entire container platform, host command and control (C2) servers, build a botnet, or steal sensitive data.

To prevent such malicious behaviour, security policies can be enforced with different existing solutions, whose limitations are as follows. First, *retroactive* approaches (e.g., [4], [5]) detect security breaches after the fact, potentially resulting in irreversible damages (e.g., denial of service or information leakage). Second, *intercept-and-check* approaches (e.g., OPA/Gatekeeper [6], the most popular solution for runtime security policy enforcement in Kubernetes [7]) enforce security at runtime by intercepting and verifying each user request against all applicable security policies. While those runtime approaches can naturally prevent irreversible damages, they can become impractical in large and highly dynamic container environments (as explained below). Finally, *proactive* approaches (e.g., [8]) can help coping with those challenges and reducing the response time. However, proactivization (i.e., precomputing in advance) of runtime enforcement solutions may require significant manual effort. In particular, as one of the most popular solutions for Kubernetes security enforcement, OPA/Gatekeeper suffers from the following limitations.

- First, the verification of multiple policies for each request using a reactive approach like OPA/Gatekeeper induces significant delay that increases with the size of the cluster and the number of policies (e.g., OPA/Gatekeeper can cause up to 600 ms delay in a Kubernetes cluster of 800 Pods, as reported in Section 6). A significant part of this delay is due to the need to collect and process input data from the cluster at runtime to verify each policy. This may prohibit the use of such solutions in time-sensitive scenarios (e.g., microservice-based serverless computing targeting sub-millisecond latency goals [9], [10], event-driven serverless edge computing [11], [12]).
- Second, to remediate the above performance issue, OPA/Gatekeeper can optionally leverage state replication to speed up the verification instead of fetching the actual system state in real-time from the orchestrator. However, this may cause inconsistency between the replicated state and the actual one (e.g., vulnerability CVE-2021-43979 [13] which we have discovered), leading to a potential policy bypass, as shown in our motivating example below.
- Third, while proactive security policy enforcement can help reducing the response time of OPA/Gatekeeper without introducing security issues [8], the success of such a solution depends on the correct identification of which policies to proactivize first to achieve better efficiency. Such identification of policies typically relies on manual efforts in practice, as there do not exist suitable tools for this purpose.

We further depict these limitations in the following example.

Motivating Example. Fig. 1 illustrates an example of a policy bypass attack caused by the delay in state replication by OPA/Gatekeeper. The security policy in this example

- *H. Kermabon-Bobinnec, S. Bagheri, M. GholipourChoubeh, S. Majumdar and L. Wang are with the Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada.*
- *Y. Jarraya and M. Pourzandi are with Ericsson Security, Ericsson, Montreal, Canada.*

mitigates an existing vulnerability in Kubernetes (CVE-2020-8554 [5]) allowing an attacker to intercept traffic directed to a victim resource by re-using its IP address. As shown in Fig. 1, a user sends a `Create Pod` request ① via the Kubernetes Application Programming Interface (API), which is intercepted by a Kubernetes admission webhook and forwarded to the admission controller (i.e., OPA/Gatekeeper) for verification ②. The latter verifies it against the security policy and allows it ③. In ④, the Pod is created in the cluster with the IP address `192.168.1.1`. While OPA/Gatekeeper is replicating the new cluster state, a malicious user simultaneously makes a `Create Service` request with the same IP address for the *externalIP* ❺. Since this request arrives after α time which is less than the data replication delay (i.e., β), the replicated state has not been updated yet with the IP of the freshly created Pod, therefore no policy breach is detected ❻. Even though the policy is configured to be enforced, OPA/Gatekeeper allows the Service creation with an *externalIP* equal to the existing IP of the victim Pod ❼ as the replicated state is inconsistent with the actual state, leading to policy bypass. Thus, the attacker succeeds to intercept the traffic related to this IP ❽.

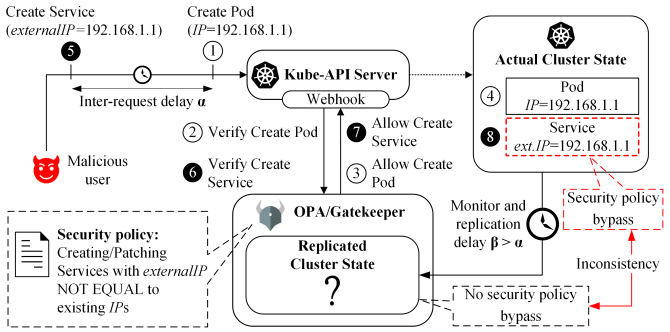


Fig. 1. Policy bypass due to data replication delay

Table 1 shows the response time (in ms) of OPA/Gatekeeper to verify a list of real-world policies [14], [15], [16] ranked (Column 1) in descending order for a cluster with 100 resources (medium sized cluster). It also shows the percentage of total response time (Column 2), the cumulative percentage of total response time to verify all prior policies (Column 3), as well as the cumulative percentage of policies verified up to this one (Column 4). Note that measurements take into account potential parallel execution of policies when possible (i.e., by relying on the OPA/Gatekeeper policy engine, with the `--max-serving-threads=4` option). As shown by the bold values, focusing proactivization efforts on only the first seven policies (ranked from #1 to #7), which represents only 22.5% of the policies to be verified, can eventually reduce around 80.1% of the total verification time. In contrast, focusing on the remaining policies (i.e., policies ranked from #8 to #31) would help reducing only 19.9% from the total verification time, while requiring more effort, as 77.5% of the policies needs to be proactivized in this case. This shows that the proactivization of the first seven policies is much more cost-effective as they require less effort and are much more expensive from the response time point of view.

The above observations demonstrate the potential for developing a performance profiling-based proactive solution to improve the overall performance of policy enforcement without introducing security issues (related to state replication). However, this comes with challenges to assess

TABLE 1
Distribution of response time for various OPA policies

Rank	Response Time (ms)	% of Total Response Time	Cumulative % of Total Response Time	Cumulative % of policies
#1	112.36	34.2	34.2	3.2
#2	26.80	8.2	42.4	6.4
#3	26.25	8.0	50.4	9.6
#4	26.00	7.9	58.3	12.9
#5	24.24	7.4	65.7	16.1
#6	23.95	7.3	73.0	19.3
#7	**23.46**	**7.1**	**80.1**	**22.5**
#8	19.54	5.9	86.0	25.7
...	...	...	...	...
#30	0.88	0.3	99.8	96.8
#31	0.71	0.2	100	100

the performance of security policies in OPA/Gatekeeper and prioritize the right ones as follows:

- Although policy performance profiling is possible in OPA (as detailed in Section 2.1), it is not accessible to Kubernetes users. To do so, one must manually perform the profiling from inside the container, or use OPA outside of Kubernetes (knowing that performance results can widely differ between inside and outside a container).
- To the best of our knowledge, no option is available to evaluate multiple policies at once: instead, policies can be profiled only one at a time.
- In addition to the performance profile, the usage frequency of each policy also needs to be taken into account. However, OPA does not expose metrics regarding single policy usage.
- Finally, there is no existing solution to automatically compare policies performance. Instead, one must manually go through multiple policy evaluation reports (facing the above challenges) and rank them to determine which ones can be improved using proactivization.

In this paper, we address those challenges with a performance profiling-based proactive approach, namely, *PerfSPEC*. The key idea behind this work is that computationally intensive verification steps can be performed in advance (i.e., before the actual events occur), leaving solely the lightweight enforcement steps at runtime and ensuring a better response time. To do so, we first evaluate the performance profile of existing security policies and rank them according to the overhead they induce. We then learn from historical data (i.e., logs of past events) and build predictive models to foretell future critical events involved in those policies incurring the highest overhead. We then apply those predictions to proactively start verifying such critical events against currently enforced security policies, starting with the ones that are highly ranked. Eventually, once the critical event occurs, PerfSPEC enforces the results of security policies based on the pre-computed verification to significantly reduce the delay incurred by policy verification in large container environments.

Our main contributions are as follows:

- To the best of our knowledge, this is the first work offering a fully automated framework for proactive security policy enforcement in container environments. PerfSPEC ensures that security policies are enforced with a practical response time, even for large container environments (e.g., less than 10 ms for 800 Pods in contrast to 600 ms with OPA/Gatekeeper).
- We automate the arduous process of evaluating the overhead induced by security policies on the cluster performance. We establish a ranking to proficiently target expensive policies

with our proactive approach, and continuously update such ranking with new policies as they are created. Our ranking can greatly improve the response time of policy enforcement using the same amount of effort (e.g., improvement by 98% with ranking in large clusters versus less than 49% without).
- We build the first predictive models for container events by studying the dependencies between Kubernetes events, then comparing different learning approaches (namely, Bayesian Network, n-gram and LSTM). Our models can be used for other proactive solutions beyond policy enforcement.
- We integrate PerfSPEC with Kubernetes, the most popular container orchestrator, and discuss its adaptation to other environments (e.g., Docker Swarm [17], OpenShift [18]). Additionally, we discover and publish a vulnerability in OPA/Gatekeeper (CVE-2021-43979 [13]).

2 Background and Models

This section provides backgrounds and our threat model.

2.1 Background

Containerization. As demonstrated in [19] and shown in the left side of Fig. 2, a container is a bundle of applications and their dependencies running through operating system (OS)-level virtualization. Unlike VMs, containers do not require hardware virtualization, thus resulting in much faster deployments and less resource consumption. The role of the container orchestrator (e.g., Kubernetes [3]) is to indirectly manage containers (i.e., via a container runtime environment such as Docker [20] or Containerd [21]) through their entire life cycle (scheduling, deployment, and deletion). More detailed background on Kubernetes is in Section 5.2.

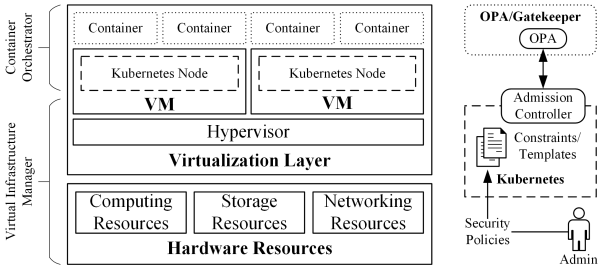


Fig. 2. Architecture of a Kubernetes environment using OPA/Gatekeeper [19]

Security Policy Compliance. As shown on the right side of Fig. 2, OPA/Gatekeeper is a policy enforcement solution for Kubernetes. At its core, OPA (Open Policy Agent) is a compliance verification engine running inside a container. In the case of Gatekeeper, OPA is deployed in a container alongside the necessary interface to make it queryable from the Kubernetes API through a webhook. Additionally, security policies can be deployed directly in the Kubernetes cluster using two Custom Resource Definitions (CRD): a `Template` and a `Constraint`. Gatekeeper periodically queries the Kubernetes cluster to discover if any new Templates and Constraints are deployed.

Security Policy Performance. OPA offers a way to evaluate and detail the performance metrics of a policy at runtime. To do so, the user can call the `opa eval` or the `opa bench` commands. `opa eval` performs profiling, i.e., it measures the time needed to load and compile the policy, and evaluate the query. `opa bench` conducts benchmarking, i.e., presenting the same metrics but also extensively running the evaluation a great number of times to provide statistical data. Although both commands generate a profile report about the time taken by a policy, these features by themselves are not enough for our purpose, as mentioned in Section 1.

2.2 Threat Model

In-scope Threats. In-scope threats include both external and insider attacks, indifferently. We assume such attacks are made possible due to implementation flaws, misconfigurations, or vulnerabilities in the container environment. We limit our scope to attacks resulting from sequences of operations made to the Kubernetes API. Like most existing works on security verification (e.g., [22], [23]), we assume that the integrity of PerfSPEC and the Kubernetes environment (with its API requests and events logs) are protected with existing trusted computing techniques such as remote attestation [24], [25].

Out-of-scope Threats. As PerfSPEC focuses on the Kubernetes policy compliance, other related issues such as container runtime security and attack detection are out of its scope. We exclude attacks that can completely bypass the Kubernetes API, and attacks that do not involve any Kubernetes API requests. Similar to most works on security verification (e.g., [22], [23]), we do not consider attackers who can tamper with Kubernetes or the PerfSPEC solution itself. Like many other verification works [26], [27], [28], PerfSPEC relies on security experts to correctly define security policies for the security of the system.

3 PerfSPEC Approach

This section presents the PerfSPEC approach.

3.1 Overview

Fig. 3 depicts an overview of our approach and its three major phases: *ranking*, *learning* and *runtime*. During the *ranking* phase, PerfSPEC focuses on evaluating the performance of policies and ranking them so that proactivization efforts can be properly prioritized. To do so, it gathers knowledge about the policies present in the environment, their usage, and their performance profiles. Eventually, PerfSPEC ranks the policies according to different metrics and chooses the top group to proceed in the second phase, considered as priority. During the *learning* phase, PerfSPEC captures the dependencies between events to predict future events. To that end, PerfSPEC processes historical data and builds a predictive model that captures the probabilistic dependency relationships among events in the container environment. Such predictive model is leveraged in the third phase to foretell events. During the *runtime* phase, PerfSPEC enforces security policy in a proactive manner. Specifically, it first conducts proactive verification against selected security policies (provided by the *ranking* phase) for predicted future events by utilizing the predictive models, and then enforces those proactive verification results when actual events occur. In the following, we elaborate on each phase.

3.2 The Ranking Phase

This phase (top part of Fig. 3) is to evaluate the policies and rank them based on performance metrics as follows.

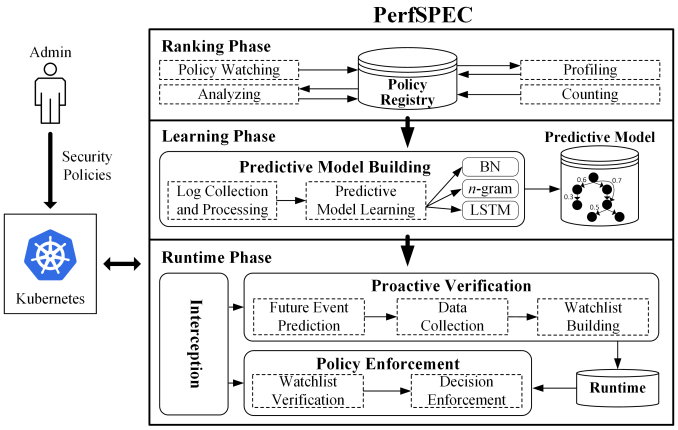


Fig. 3. Overview of the PerfSPEC approach

Overview. The *ranking* phase consists of four steps. It takes different parameters as input and returns a ranking of security policies based on their evaluations using performance metrics (as explained next). First, it continuously performs *policy watching*, i.e., PerfSPEC monitors the presence of new policies (or update of existing ones) in the Kubernetes cluster. It stores information about the policies in the `policy registry`. The *profiling* step performs profiling of resource consumption and response time of those policies. To do so, it starts by taking as input a query from historical data (e.g., the first query ran against a particular policy). It then leverages OPA for profiling and stores the performance results in the `policy registry`. Meanwhile, our solution continuously performs *counting* to track the usage frequency of each policy and estimate their global time and computation consumption (done during *analyzing*). These numbers are also continuously updated in the `policy registry`. Finally, during the *analyzing* step, PerfSPEC queries the `policy registry` and ranks the policies by performance (from worst to best) according to a parameter specified by the admin. It also provides hints about what specific line of code can be improved (e.g., by using proactive computation [8]) and what are the critical events involved in each policy. In the following, we detail each component of our solution.

Policy Registry. The `policy registry` is a database storing policies, the count of their usage, and their performance profiles. Its single `Policy` table contains seven different columns:

- *Policy_name* is a key value identifying the name of the policy. Even though Rego files can contain multiple policies per file, we consider each policy independently. In the context of Kubernetes and OPA/Gatekeeper, we consider that one Constraint represents one independent policy (i.e., multiple Constraints associated with the same Template will be stored as multiple policies).
- *Count* tracks the number of times a *Policy* was used.
- *First_observed* contains the timestamp at which the policy was added to the database. This value is necessary in order to compute the usage frequency of policies. The age of the policy is computed while *analyzing* the policies to rank them.
- *Response_time* is a value reported by the *profiling* step.
- *Computation_resource* is a value storing the average percentage of resources (CPU and memory) used over the total amount available, reported by the *profiling* step. For instance, if a policy benchmark reports a CPU consumption of 14% and a memory consumption of 26%, the average value of 20% is considered.
- *Profile_report* contains a complete profiling evaluation report as provided by the *profiling* step. This report is to be handed to the end-user alongside the list of policies to help understand the result.
- *Sample_data* contains a sample of Kubernetes request used to verify against a policy. This usually corresponds to the Kubernetes request used in the first query observed for a policy.

Policy Watching. During this step, PerfSPEC periodically (e.g., every few seconds) queries Kubernetes to keep track of the deployed policies and update the `policy registry` accordingly. If a new policy is found, a new entry is added to the registry (by filling the *policy_name* and *first_observed* fields, setting the *count* to 0, and leaving other fields empty). If an existing policy was updated, its *count* is reset to 0; its *first_observed* timestamp is updated and its existing *(response_time, computation_resource, profile_report, sample_data)* fields are deleted. This is because updated policies can have very different performance profiles and use different input data, thus they should be re-evaluated during the next *profiling* step.

Profiling. During *profiling*, PerfSPEC measures performance profiles for each policy. First, it periodically looks for policies in the `policy registry` that have an empty *profile_report* field. Then, if *sample_data* is available but no profile yet exists (e.g., a policy has been recently added/updated), it establishes a policy profile using the `opa eval` command (as explained in Section 2.1). The profiling process is described in Algorithm 1. The query's *response_time*, the *computation_resource* as well as the *profile_report* are averaged over n runs (e.g., $n = 100$) and saved in the `policy registry`.

Algorithm 1 Profiling process of PerfSPEC *ranking* phase

```
1: Input: PolicyName, SampleInputData
2: Output: ResponseTime, ComputationResource, ProfileReport
3: procedure PROFILE(PolicyName, SampleInputData)
4:     for Policy in PolicyRegistry do
5:         if SampleData is not empty and ProfileReport is empty then
6:             Get the corresponding policy_file;
7:             for i from i=0 to i=n do
8:                 Run profiling;
9:                 Save profiling result;
10:            Average profiling results;
11:            Save ResponseTime, ComputationResource, ProfileReport
   to PolicyRegistry;
```

Counting. While *counting*, PerfSPEC keeps track of the usage of existing policies. As soon as a policy is verified in OPA/Gatekeeper, it increments the `count` field of that policy by one. Additionally, it adds the Kubernetes request received from the query to the *sample_data* field when the policy is queried for the first time (i.e., the `count` value was equal to 0).

Analyzing. During the *analyzing* step, PerfSPEC computes a ranking of the policies according to their performance. To do so, it assigns a score to each policy. First, it takes as input a configuration from the end-user that specifies whether response time or computational resources (i.e., CPU/memory) should be taken as the main criteria for ranking the policies. This configuration consists of a trade-off coefficient $\alpha \in [0,1]$ expressing the amount of response time to be weighed into the ranking, while its complement $(1-\alpha)$ represents the amount

of computational resource to be weighed into the ranking. To compute the score of a policy, we use a formula expressing the product of its performance and its usage frequency: $Score = (\alpha * RT + (1-\alpha) * CR) \frac{Count}{(CurrentTime - FirstObserved)}$ where RT is the response time and CR is the computational resources consumed. Note that we do not simply consider the number of times a policy is used (i.e., its $Count$), but instead the frequency at which policies are used, i.e., the count of their usage over the amount of time they have been enforced. PerfSPEC ranks existing policies according to that score.

Moreover, PerfSPEC enriches the ranking by identifying critical events involved in some policies and providing hints about ways to make a policy proactive. Specifically, our solution parses each policy file and extracts the method and type of resources triggering it (as specified in the `match` field of the Constraint CRD [29]) and uses it as the critical event for the policy for the next steps. Also, it indicates to the user which part of the policy can be potentially made proactively (e.g., HTTP calls made to the Kubernetes API, in the form `GET https://kubernetes.default.svc:6443/api/v1/<resource_type>` can often be made in advance and their results stored in a watchlist, as detailed in Section 3.4). Finally, PerfSPEC returns to the end-user a ranking of policies for proactivization (in the next phases).

Example 1. Fig. 4 details an example of the *ranking* phase using our approach. In Step ❶, the admin creates a new policy named *Label Existence* using Kubernetes CRD Template and Constraint. During *policy watching*, PerfSPEC detects this new policy in Step ❷ and adds it to the `policy registry` in Step ❸ (*timestamp* 1669329373). In Step ❹, the new policy entry is initialized with a *count* of 0. Once the new policy has been used once, *profiling* is done using the `opa eval` command to measure a performance profile in Step ❺. The results are stored in the `policy registry` in Step ❻.

After a while, in Step ❼ (*timestamp* 1669334850), suppose the admin decides to optimize its security policies performance and privileges response time over resource consumption by specifying the trade-off coefficient $\alpha = 0.7$ and calling the *analyzing* step. Our solution then calculates the score of each policy in Step ❽. For instance, the policy named *Ingress Conflicts* was used 12 times since its deployment at *timestamp* 1627266506. Its average response time is 62.7 ms and its average resource consumption is 26%. PerfSPEC calculates a score of 0.147 (scaled by a factor of 10,000 for readability). It does the same for the policy *Block CVE-2020-8554* and calculates a score of 38.718. In Step ❾, the admin concludes that the policy *Block CVE-2020-8554* should preferably be optimized first, as it has a higher score than *Ingress conflicts*. Meanwhile, the *Label Existence* policy continues to be profiled for future ranking operations.

3.3 The Learning Phase

This phase (middle part of Fig. 3) is to learn a predictive model that captures the probabilistic dependencies among management events in the container environment as follows.

Overview. The *learning* phase consists of two steps, namely, *log collection and processing* and *predictive model learning*, which will be detailed later. A predictive model can be represented as a directed graph where nodes indicate Kubernetes events, directed edges indicate one event happening (directly or indirectly) after another event, labelled with a probability. Two types of dependencies between events are represented in our model: (i) *inter-resource dependency* among different resources (e.g., a Pod resource cannot be created in a Namespace unless that Namespace resource first exists), and (ii) *intra-resource dependency* within one resource (e.g., a `delete` event on a Pod resource can only be performed after creation of the Pod). Note that similar management events and their dependencies exist for other container orchestrators, as discussed in Section 5.2. We give details about the *learning* phase in the following.

Log Collection and Processing. In order to build predictive models, PerfSPEC first collects event logs from the container environment (e.g., from Kubernetes), then prepares the log entries, identifies the event types and extracts sequences of events. Doing so requires to filter out repeated events and arrange the events in sequences suitable for learning.

Example 2. Fig. 5 shows an example of *log collection and processing*. In (1) and (2), PerfSPEC collects and extracts the events "`Create Pod`, `Delete Pod`, `Create Service`, `Create Pod`, `Create Pod`, `Create Service`, `Patch Service`, `Create Pod`, `Patch Service`". Then, in (3), it removes a repetition of `Create Pod`. Next, sequences are built in (4).

Predictive Model Learning. PerfSPEC learns predictive models (including their nodes, edges, and labels of edges) from sequences obtained in the previous step. Currently, PerfSPEC offers three different ways of learning predictive models, namely, Bayesian learning [30], n-gram [31] and Long Short-Term Memory (LSTM) network [32]. The choice of predictive model learning approach to apply in practice is left to the end-user. In the following, we detail the model learning step using each of these approaches and we evaluate their performance in Section 6.2.

3.3.1 Bayesian Learning

A Bayesian network [30] is a directed acyclic graph (DAG) representing random variables (nodes) and their conditional dependencies. Bayesian learning operates on a set of data and is typically composed of two steps: structure learning and parameter learning. The former aims at identifying the nodes of the model and their relationships while the latter computes their conditional dependency probabilities. The result is a Bayesian network in which each node corresponds to an event and a directed edge from one event A to another event B represents the probability of observing event B following event A. We can use such model to determine the probability of one event to happen after another and thus predict events.

Since the structure of the model is easily determined from the logs, PerfSPEC builds the structure of the Bayesian network based on the sequences of event, then proceeds to learn parameters.

- First, PerfSPEC identifies all unique event types from the sequences as the nodes of the model. Then, it considers immediate transitions between event-pairs as edges between those event nodes. For instance, Fig. 6a shows an excerpt of the output with such nodes and edges for Kubernetes. The resulting model is a directed graph.

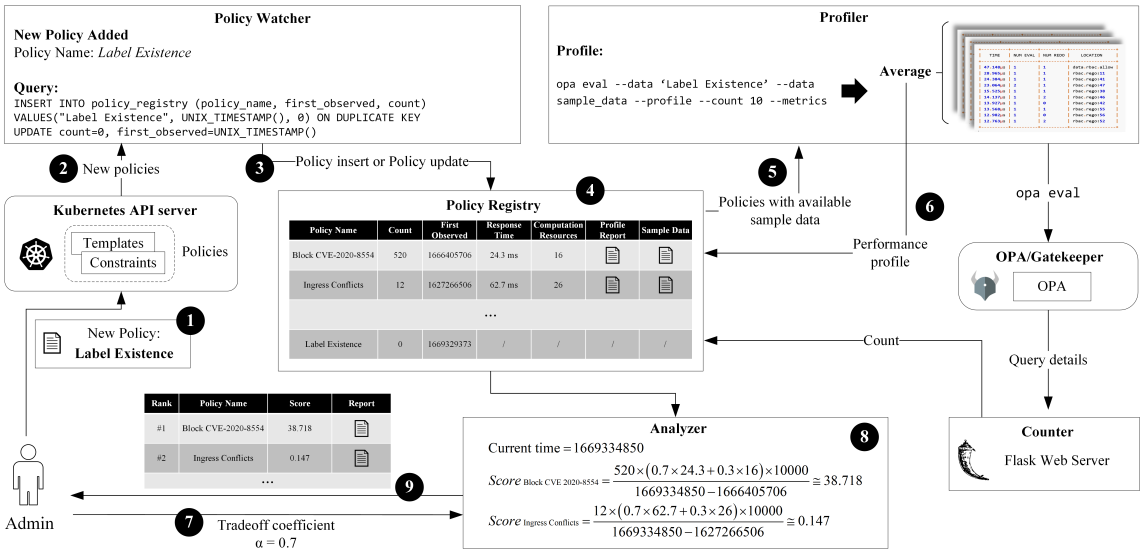


Fig. 4. An example of *ranking* phase using PerfSPEC

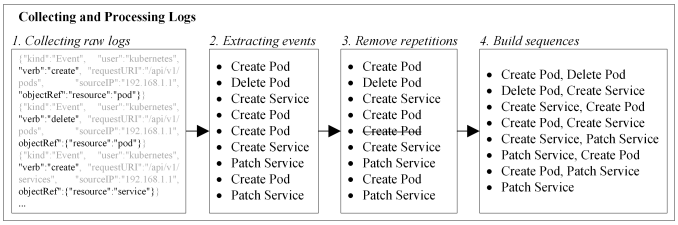
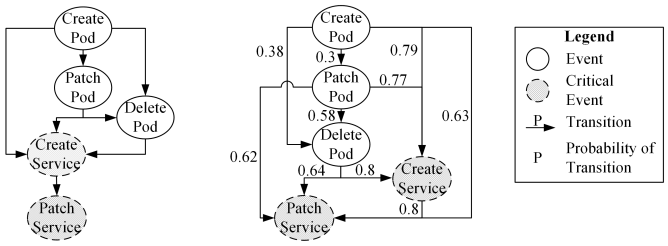


Fig. 5. An example of log collection and processing

- Then, to make Bayesian learning possible, the directed graph is transformed into a DAG by breaking cycles. To do so, our solution employs three steps: (i) First, it removes any incoherent edges, e.g., it ensures that for any given resource, an event involving the `create` and `delete` methods does not appear directly after or before another event involving any other methods, respectively. Loops (i.e., edges from a node to itself) are also removed. (ii) Then, it focuses on breaking bi-directional edges to remove cycles. Specifically, to choose which direction of such edges to delete, we consider that, e.g., regardless of the resource, `create` events should happen before any other event, and `delete` event should happen after any other events. (iii) Lastly, if the remaining graph is still not acyclic, we employ a Depth-First Search (DFS) algorithm [33] with backtracking to visit the graph and remove edges involved in cycles.
- Subsequently, to add the non-immediate transitions (i.e., transitions from one event to another through one or more intermediate transitions), PerfSPEC utilizes a Breadth-First Search (BFS) algorithm [33] to determine each node's ability to reach other non-adjacent nodes and includes these transitions as additional edges in the model. Non-immediate edges are added to the graph only if they do not introduce cycles. The resulting model remains a DAG. Fig. 6b shows an example of such model with immediate and non-immediate transitions.
- Finally, PerfSPEC learns the probability of transition between events by leveraging existing Bayesian parameter learning techniques [30] where the conditional probabilities indicate the likelihood for (immediate or non-immediate) transitions to happen. Those probabilities are labels on the corresponding edges in the predictive model. The built predictive model (as shown in Fig. 6b) will be used during the *runtime* phase.



(a) Structure based on immediate transitions (b) Final predictive model based on both immediate and non-immediate transitions

Fig. 6. Excerpt of PerfSPEC predictive models

Example 3. Fig. 7 shows an example of building a predictive model using Bayesian learning. In (1), the four unique nodes of the model are created: `Create Pod`, `Delete Pod`, `Create Service`, and `Patch Service`. In (2), eight edges are identified from the immediate transitions. In (3a), one edge from `Delete Pod` to `Create Pod` is considered incoherent and deleted because a Pod deletion cannot happen before its creation. In (3b), we break remaining cycles by removing bi-directional edges, only keeping edges going to the critical events `Create Service` and `Patch Service`. In (4) using the BFS algorithm, a non-immediate transition is obtained: `Delete Pod` to `Patch Service` (through `Create Service`). Finally, in (5), conditional probabilities are learned using Bayes probabilities.

3.3.2 N-gram

The n-gram [31] is a Natural Language Processing (NLP) technique to predict the occurrence of a word based on the previous $n-1$ words. That is, n-gram models are used to compute the likelihood of a word x_i to immediately follow a sequence of $n-1$ words $x_{i-1}, x_{i-2}, \ldots, x_{i-(n-1)}$. In spite of its typical usage in predicting the next word in a sentence, n-gram models can also be used to predict the next event in a sequence. Therefore, we explore n-gram as a potential predictive model learner.

We choose to implement both a 2-gram and a 3-gram, i.e.

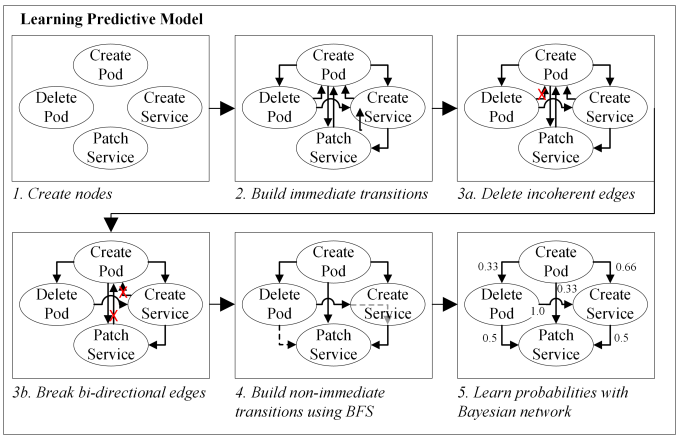


Fig. 7. An example of predictive model building using Bayesian learning

the former will use the very latest historical event to predict the next one, while the latter will use the two latest events to predict the next one. First, the training dataset is splitted into chunks of sub-sequences of 1 or 2 events (for 2-gram and 3-gram, respectively) followed by the next event (the ground truth, or label). The n-gram models then keep the absolute count of how many times they encountered each event after different inputs, then compute their relative probability of occurrence.

Example 4. A 2-gram's count table is represented in Table 2, and the relative probabilities as calculated after the end of training are also given in the right part of the table. For this example, we consider the training sequence "`Create Pod`, `Delete Pod`, `Create Pod`, `Delete Pod`, `Create Service`", from which we first extract four input samples: "`Create Pod`, `Delete Pod`", "`Delete Pod`, `Create Pod`", "`Create Pod`, `Delete Pod`", and "`Delete Pod`,`Create Service`". As a result, the 2-gram relative probabilities are calculated, i.e., after `Create Pod`, only the event `Delete Pod` was observed (two times). The probability of transition for the edge (`Create Pod`, `Delete Pod`) is then 1.0. Similarly, after `Delete Pod`, the events `Create Pod` and `Create Service` were observed one time each, thus their probability of transition would be 0.5 each.

TABLE 2
An example of the count and prediction tables of a 2-gram after training

Input	Count			Probability of prediction		
	`Create Pod`	`Delete Pod`	`Create Service`	`Create Pod`	`Delete Pod`	`Create Service`
`Create Pod`	0	2	0	0	1.0	0
`Delete Pod`	1	0	1	0.5	0	0.5
`Create Service`	0	0	0	0	0	0

3.3.3 LSTM

Long Short Term Memory (LSTM) [32] is a deep learning-based technique that is widely used in various applications (including security solutions). In the same way as n-gram, LSTM can take as input sub-sequences of different sizes. We choose to experiment with two different window sizes of one and two events. The data is prepared in the same way as that for n-gram, i.e., we extract sub-sequences of one (or two) events, while the next event is being used as ground truth to train the model.

3.4 The Runtime Phase

This phase (bottom part of Fig. 3) is to intercept events, conduct proactive verification, and enforce security policies at runtime.

Overview. The *runtime* phase of PerfSPEC takes as input a configuration from the user (prediction threshold value and critical events) and data from the previous phases (security policies and predictive models). It is composed of four steps: first, during *interception*, our solution catches an event at runtime and triggers another step depending on the nature of that event. If the event is not considered critical, it calls the *proactive verification* step to build a watchlist, otherwise it begins *policy enforcement* with the content of previously computed watchlists. Once finished, our solution returns to the *interception* step and wait for another event to happen. PerfSPEC keeps track of its watchlists and configuration in a database. Algorithm 2 summarizes the *runtime* phase of PerfSPEC.

Algorithm 2 PerfSPEC *runtime* phase

```
1: Input: Intercepted request
2: procedure RUNTIME(Request)
3:     Parse the request
4:     Extract the relevant fields and type the event accordingly
5:     if event is critical then
6:         Verify the watchlist and return a decision
7:     else
8:         Get the probability of critical event from the model
9:         if probability > policy prediction threshold then
10:            Start pre-computation and build watchlist
```

Runtime Database. The *runtime* phase maintains a `Runtime` database composed of four tables, `PolicySettings`, `PolicyThreshold`, `PolicyWatchlist`, and `Model`.

- The `PolicySettings` table stores the configuration of each policy and contains a policy description attribute, the corresponding action attribute (e.g., deny, warn, and allow), as well as a boolean attribute for enabling or disabling the proactive feature for that policy.
- The `PolicyThreshold` table stores the critical events and their prediction threshold values defined for each policy, and contains a policy foreign key referring to the policy primary key of the `PolicySettings` table, a critical event attribute containing an event considered critical for that policy, and a threshold attribute containing the prediction threshold value for that critical event.
- The `PolicyWatchlist` table stores the actual watchlists content pre-computed by PerfSPEC for each policy, and contains a policy foreign key referring to the policy primary key of the `PolicySettings` table.
- Finally, the `Model` table stores the excerpt of predictive model for each policy (built during the *learning* phase), and contains a policy foreign key referring to the policy primary key of the `PolicySettings` table, pairs `(current event, future event)` representing a possible transition, and the probability of that transition.

Interception. PerfSPEC intercepts requests made by users to the container orchestrator (e.g., Kubernetes), and provides the details of events to the following runtime steps. The current request is initially blocked to determine if it is critical (i.e., could potentially breach a security policy). In case the event is critical, PerfSPEC maintains the blocking until it completes the *policy enforcement* step. Otherwise, if the event is not critical,

then PerfSPEC releases the block to allow Kubernetes to execute the event. In parallel, PerfSPEC predicts the potential next event and performs *proactive verification*.

Proactive Verification. Based on the current event and using the predictive model, PerfSPEC predicts future events and performs proactive verification. Precisely, it first queries the predictive model to identify the highly probable future critical events (i.e., with a prediction probability higher than the chosen prediction threshold). Second, for such predicted events, PerfSPEC collects the existing resource data related to each security policy from the container environment. Finally, it prepares the watchlist(s) (e.g., a blacklist of parameters that may lead to a policy breach) by verifying the collected data against each policy. As PerfSPEC blocks critical events until proactive verification is done, it prevents the kind of inconsistencies demonstrated in Section 1. Our approach causes significantly less delay to users than an *intercept-and-check* solution, as our experiments show in Section 6.

Policy Enforcement. At runtime, PerfSPEC enforces security policies based on the content of the watchlists built during proactive verification. In case the intercepted event is considered critical w.r.t. a security policy, PerfSPEC first checks the parameters of that event against the watchlists built for that policy. Then, based on whether the requested parameters are present in, or absent from the watchlists, PerfSPEC takes action to either *allow* or *deny* the request, according to the watchlist rule (e.g., whitelist or blacklist). Note that for any inaccuracy in the watchlist built for an event (e.g., wrong event prediction, incomplete predictive model, etc.), PerfSPEC would simply fall back to the *intercept-and-check* mode, whose impact will be evaluated through experiments in Section 6.

Example 5. Fig. 8 depicts an example of the *runtime* phase. We consider a scenario where a vulnerability (CVE-2020-8554 [5]) can be exploited to intercept the traffic between two resources (man-in-the-middle attack), similarly to Section 1. To prevent that vulnerability, a security policy can be specified as: *Services should not be allowed to use an external IP address identical to any existing IPs*. Following the *ranking* phase, that particular policy has been previously identified as the most expensive one in terms of response time and resources, and the probabilistic predictive model, built during the *learning* phase, is the same as shown in Fig. 6b. Critical events (i.e., events that trigger the verification) are events impacting Services IP addresses, i.e., `Create Service` and `Patch Service`.

At runtime, for the first intercepted event `Create Pod` with its IP address, `192.168.1.1`, PerfSPEC looks up in the predictive model and predicts `Create Service` as the next event, indeed considered critical. It then adds the IP address of the newly created Pod (`192.168.1.1`) to the policy watchlist (blacklist). For the second intercepted event, `Create Service` with an *externalIP* value of `192.168.1.1`, PerfSPEC denies the request as this requested IP is in the watchlist. Similarly, the third intercepted event will be allowed as the *external IP* of the Service, `192.168.0.8`, is not present in the watchlist. The fourth event will be denied as it tries to modify the *externalIP* to `192.168.1.1`, IP that is in the watchlist. Note that PerfSPEC avoids inconsistencies between the watchlist and the actual state of the cluster (as shown in Section 1) as each request is held until the pre-computation step is over.

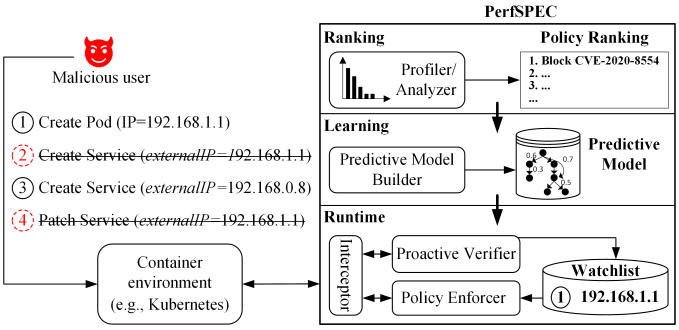


Fig. 8. PerfSPEC preventing CVE-2020-8554

4 Dataset Generation

In this section, we present our dataset of Kubernetes events and detail the method performed to collect data.

Characteristics. The dataset contains 16,548 entries of 95 unique events. Fig. 9 represents a detailed distribution of methods and resources of samples. The annotations on the plot count the number of times a sample appears in the dataset (note that only samples with a count greater than 20 are annotated). For instance, the most frequently observed event is `update secrets` with 2,234 occurrences, but we can see that Pods are the most created resources, with the event `create pods` observed 1,020 times.

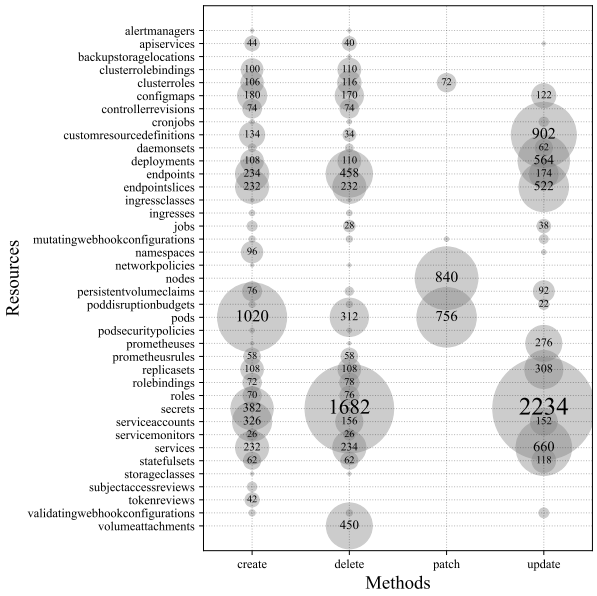


Fig. 9. Detailed distribution of samples in our dataset

Methodology. To generate events, we adopt the 60 most popular Helm charts available from ArtifactHub [34] that we deploy and delete at regular intervals to simulate administrative operations. As these charts are commonly used by users to deploy applications and services in Kubernetes, we consider that the sequences of operations performed for each deployment and deletion are realistic and representative. We save the events using Kubernetes audit logs as described in Section 5. We employed this dataset to build predictive models as described in Section 3 and to obtain the experiment results in Section 6. Fig. 10 depicts an excerpt of model generated from this dataset using the Bayesian learning

approach for the policies *Block CVE-2020-8554* and *Unique Ingress*, both involving the critical event `create_services`. The policies are further detailed in Section 6.

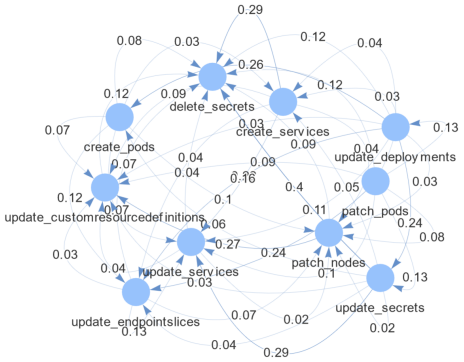


Fig. 10. Excerpt of our predictive model for the policies *Block CVE-2020-8554* and *Unique Ingress*, both involving the critical event `create_services`. Each node represents an event and each directed edge represents the probability of two events happening one after the other

5 Implementation and Integration

This section describes the implementation and integration details of PerfSPEC.

5.1 PerfSPEC Implementation

Fig. 11 shows the high-level architecture of PerfSPEC through its three major phases: *ranking*, *learning* and *runtime*. The first component performs the *ranking* phase using four modules. The `policy watcher` and `counter` modules continuously track the creation/update/deletion of security policies and their usage frequency, respectively. The `profiler` module measures the response time of each security policy, and the `analyzer` module ranks all policies from the most expensive to the least expensive, according to metrics selected by the user. All four modules access and share data through the `policy registry`. The second component implements the *learning* phase. To that end, first the `log collector` continuously collects historical data from the Kubernetes cluster, then the `log processor` prepares the data and extracts Kubernetes events. Finally, the `predictive model learner` module leverages three different learning techniques (namely, Bayesian Network, n-gram and LSTM), builds the predictive model and saves it. The third and last component is for the *runtime* phase. First, the `interceptor` module captures live events and determines which module to call next. If the event is not critical, the `proactive verifier` module predicts future events, pre-computes verification results, and stores them in the `watchlist` database. Otherwise, if the event is critical, the `policy enforcer` module enforces a decision based on the content of the `watchlist`. We elaborate on those three phases in the following.

5.1.1 Implementation of the Ranking Phase

This section details the implementation of the *ranking* phase.

- The `policy registry` is a portable and lightweight database using *SQLite* [35].
- The `policy watcher` module is developed in Python and uses HTTP requests to communicate with the Kubernetes API and keep track of the existing policies. The *First_observed*

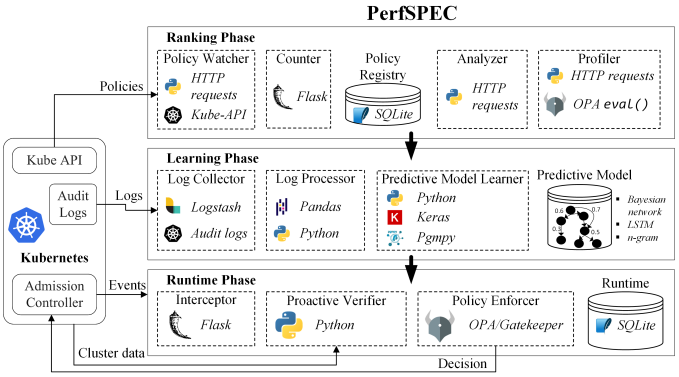


Fig. 11. PerfSPEC architecture

timestamp is expressed as the Linux timestamp (i.e., the number of seconds spent since January 1st 1970) of the first deployment of the policy as observed by the `policy watcher`.

- The `counter` module is an HTTP server implemented with Flask [36] that receives information about a policy and the corresponding Kubernetes input request (in `JSON` format) every time a policy is verified. To do so, the OPA source code of the `Driver.Query()` function [37] (written in Go) is modified in order to make an HTTP POST query to the `counter` when necessary.
- The `profiler` benefits from the proximity of our solution with OPA/Gatekeeper to run the `opa eval` command directly inside the OPA container and obtain results as precise as possible. Specifically, it runs the command `opa eval --data` *policy_file* `--data SampleInputData --profile --metrics` over 100 iterations to average the results. Additionally, it leverages Linux's `htop` tool [38] to estimate CPU and memory consumption during profiling. Performance profiles are returned as text files and averaging is done by parsing the text fields in Python arrays using *numpy*.
- Finally, the `analyzer` module performs mathematical operations with Python and returns the policies' ranking and performance profiles. It also parses the policy code using Python to identify the critical events and potential lines of code to be proactivized. All database queries are done using Python and the *sqlite3* library.

5.1.2 Implementation of the Learning Phase

The three main modules of this component include `log collector`, `log processor`, and `predictive model learner`, as detailed below.

- The `log collector` module is responsible for collecting event logs from the container environment. To that purpose, PerfSPEC first enables the Kubernetes audit logs feature (see Section 5.2).
- The `log processor` module extracts the events from the historical data for the *predictive model learner*. It reads the audit log file (in `JSON` format), extracts the fields `objectRef[resource]` and `method` from the logs and stores them in a CSV file by leveraging Logstash [39], a popular log processor. Afterwards, using the Python data analysis toolkit *pandas* v1.2.4 [40] and our own code, it processes each entry with event typing that maps the pair (method, resource) to a string `method_resource` (event

type). Extracted events are written to a file.
- Finally, the `predictive model learner` module implements three different learning approaches, namely Bayesian network, n-gram and LSTM. The specifics of these implementations and their data preparation steps are detailed below.

Bayesian Network. To enable learning using Bayesian network, PerfSPEC first learns the structure of such model. To do so, a sliding window of size two is applied on the extracted events, to create sub-sequences composed of two consecutive elements. All using Python, sub-sequences occurrences are counted and only the most frequent sub-sequences are kept to build the structure, as Bayesian learning requires a limited amount of nodes to perform in a reasonable time. We then leverage the `BayesianModel` and `MaximumLikelihood` classes of the Python library for learning and inference in Bayesian networks, *pgmpy* v0.1.14 [41], to learn the probabilities. The obtained model is saved in a database.

N-gram. The n-gram model is implemented in Python using a dictionary data structure. The keys of the dictionary are the input sub-sequences represented as a single event in the case of 2-gram and as a 2-tuple in the case of 3-gram. The values of that dictionary are also dictionaries representing the probability of each event to happen (keys are predicted events and values are probabilities). Events in the dataset are first tokenized (i.e., we map an integer value with each unique event) using the *nltk* (Natural Language ToolKit) Python library [42]. The count and probability calculations of transitions are done using Python *numpy*.

LSTM. The LSTM models are implemented in Python using the *Keras* framework [43]. The architecture of our model consists of three layers sequentially organized: (i) an LSTM layer of depth 256, with a recurrent dropout rate of 0.2; (ii) a second LSTM layer of depth 128, with a recurrent dropout of 0.2; and (iii) a dense layer with a softmax activation.

We choose an architecture with relatively small depth (two LSTM units) as we are dealing with short sequences of events (two or three depending on the chosen window size). Also, as the vocabulary is composed of only 95 unique events, we choose a relatively small width of LSTM (256 for the first cell and 128 for the second). We use Dropout directly in the LSTM cell with a rate of 20%. Dropout layers randomly assign a percentage of input values to 0 in order to prevent the model from overfitting. The last layer is a dense layer, i.e., a fully connected layer, used with a softmax activation function as we are considering multiple potential events for prediction. Finally, as the output of the LSTM model returns a probability of appearance for each event in the vocabulary, we choose the one with highest probability as the predicted event using the *argmax* function. We train the model using a batch size of 256 samples over 30 epochs. The associated loss function is the categorical cross-entropy. We use the Adam optimizer with Keras's default parameters during our training.

5.1.3 Implementation of the Runtime Phase

The three main modules of this component are `interceptor`, `proactive verifier` and `policy enforcer`, as detailed below.

- The `interceptor` module aims at intercepting runtime event requests made to Kubernetes. To that end, PerfSPEC leverages the Kubernetes admission controller mechanism to intercept the requests received by the Kube-API server. The choice to use an admission controller ensures the portability of our solution and its independence from a specific orchestrator, since equivalent mechanisms are implemented in other orchestrators (as discussed in Section 5.2). The `interceptor` module runs as a local web server using the micro web framework Flask [36], and is registered as an admission controller in Kubernetes. The so-built webhook receives requests from the Kubernetes API server and processes them to determine if it is critical or not. If the event is not critical, the event is released and the `proactive verifier` module is queried. Otherwise, the event is put on hold and the `policy enforcer` module is called.
- The `proactive verifier` module is to incrementally build the watchlist for a security policy. Particularly, it considers the intercepted non-critical event, predicts the most probable next event and checks if that predicted event is critical for some policies. To do so, it queries the `Runtime` database for policies that consider the next predicted event critical as follows:
  `SELECT Policy FROM PolicyThreshold INNER JOIN Model ON Model.FutureEvent = PolicyThreshold.CriticalEvent WHERE ((Model.CurrentEvent =` *CurrentEvent*`) AND (Model.Probability >= PolicyThreshold.Threshold))`.
  For policies returned by this query, the `proactive verifier` module starts to collect data to build (or update, if it already exists) the content of watchlists. Specifically, it collects the required data defined with the policies using HTTP(S) requests to the cluster. As an example, for the policy *Block CVE-2020-8554* mentionned in Section 3.4, it gets the IP addresses of existing Pods by querying the API server with the following URI: `https://kubernetes.default.svc:6443/api/v1/pods` (as per the Kubernetes API reference [3]). Data collected that way (in `JSON` formatting) is then processed and fields required by the watchlists (e.g., IP addresses of existing Pods) are extracted. Finally, the `proactive verifier` module writes the collected features to the *PolicyWatchlist* table in the row corresponding to the policy.
- The `policy enforcer` module takes care of watchlist verification and decision enforcement and integrates OPA/-Gatekeeper [6] as later detailed in Section 5.2. Note that it is always possible to instead implement PerfSPEC `policy enforcer` module independently from OPA/Gatekeeper (i.e., as a standalone Kubernetes admission controller verifying the watchlists and directly returning decision to Kubernetes). However, there are several advantages in integrating PerfSPEC with OPA/Gatekeeper, such as preserving the features offered by the latter while bringing in the performance of proactive solution to existing policies.

5.2 Integration of PerfSPEC with Kubernetes

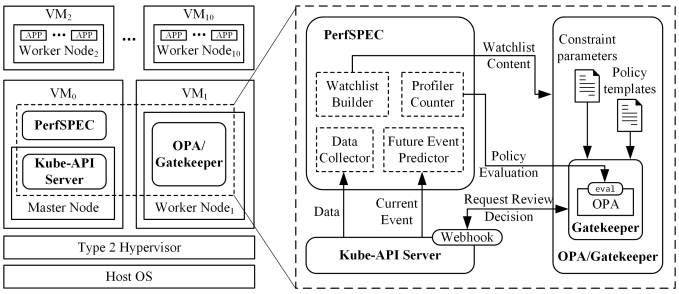
This section details the integration of PerfSPEC.

Kubernetes Background. We provide a necessary background on Kubernetes as follows:

- *Kubernetes Basics.* Kubernetes [3] is a container orchestrator that runs, manages, and coordinates the deployments of containerized applications. A Kubernetes cluster is composed of a master Node responsible for controlling and managing a set of worker Nodes, each hosting multiple Pods running the applications. The Kube-API server is used to perform any operation on the cluster that queries or modifies the state of Kubernetes resources (e.g., Pods, Services, etc.). In the following, we describe the admission control mechanism and the logging mechanism in Kubernetes, which will later be utilized in the integration of PerfSPEC.
- *Admission Control.* An admission controller in Kubernetes intercepts the requests made to the Kube-API server and performs validation, mutation, or both in order to protect clusters against malicious activities. Particularly, OPA/Gatekeeper [6] is a cloud-native project that leverages an admission controller (namely, Gatekeeper) and the Open Policy Agent (OPA) (a general-purpose policy engine that decouples decision-making from policy enforcement) to validate user requests made to the Kube-API server with respect to policies specified by the admin. When a request is made to the Kube-API server, Gatekeeper internally uses OPA to verify the intercepted request against the policies. Based on the response from OPA, Gatekeeper enforces the decision (i.e., *allow*, *deny*, or *mutate* the request).
- *Logging.* Events in Kubernetes can be collected in audit logs containing detailed events and attributes (e.g., resource-name, resource-type, method, etc.). A Kubernetes event can be described as a method (such as create, delete, update, get, etc.), a resource (such as a user account, a Deployment, a Pod, a Service, etc.), as well as more or less details depending on the audit policy. The API documentation [3] details resources and their associated methods. By knowing the method and the resource used in a request made to the Kubernetes API, one can deduct the corresponding event.

Integration with Kubernetes. Fig. 12 illustrates the integration of PerfSPEC with Kubernetes. Particularly, Fig. 12a provides a high-level overview of the integration including the deployment of a Kubernetes testbed, and Fig. 12b highlights the key integration aspects including how particularly PerfSPEC interacts with the Kube-API server and OPA/Gatekeeper. We provide details in the following.

- First, Fig. 12a shows the deployment of our Kubernetes testbed with PerfSPEC. The physical hardware of our cloud infrastructure is composed of one physical rack-mount server with 2x Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz and 128GB of DDR4-2933 running Debian 10. The container environment is deployed over 11 VMs (managed by VirtualBox 7.0) where one VM (eight vCPUs and 32GB RAM) is hosting the Kubernetes master Node, and ten other VMs (four vCPUs and 8GB RAM each) are used as worker Nodes. Each VM is running Ubuntu 20.04 and we use Python 3.9 for all programming tasks. Additionally, we use Kubernetes v1.23.14 through the `kubectl` CLI and the `kubeadm` tool for creating the cluster. The container runtime used is Containerd [21] v1.6.12, as the usage of Docker is deprecated in recent Kubernetes versions.
- Second, Fig. 12b shows the interaction between PerfSPEC, OPA/Gatekeeper and Kubernetes. PerfSPEC interacts with the Kube-API server, first to watch for policies during the *ranking* phase, then to intercept current runtime events and collects data needed for verification during the *runtime* phase. Our solution also interacts directly with OPA/-Gatekeeper to evaluate the performance of policies and their usage frequency during the *ranking* phase. During the *runtime* phase, PerfSPEC creates constraint parameters for OPA/Gatekeeper's policies based on the watchlist contents. The proper enforcement (deny or allow decision) is performed through PerfSPEC by providing OPA/Gatekeeper with a policy constraint including the watchlist content.
- This choice of integration presents several advantages: (i) Different policies can be quickly leveraged/removed by applying/deleting the corresponding OPA/Gatekeeper constraints. (ii) Widely-used OPA/Gatekeeper's features are preserved while bringing PerfSPEC's proactive advantages. (iii) PerfSPEC remains as much decoupled as possible from Kubernetes.



(a) High-level overview (b) Detailed view

Fig. 12. Integration of PerfSPEC with Kubernetes

Although our implementation is based on Kubernetes, PerfSPEC can be adapted to other container orchestrators (e.g., Docker Swarm [17], OpenShift [18]). The container-related concepts on which PerfSPEC relies on are not specific to Kubernetes, and are also implemented in Docker Swarm and OpenShift. Table 3 gives examples of similitude between different container orchestrator concepts. Even though the concept of admission control is partially absent from Docker Swarm, it is still possible to enable fine-grain control by leveraging a third-party solution such as OPA [44]. The usage of API calls in all these orchestrators greatly facilitates the access to in-cluster resources. Therefore, the adaption to those orchestrators is practically feasible.

TABLE 3
An excerpt of equivalent terminologies and concepts among three main container orchestrators

Kubernetes [3]	Docker Swarm [17]	OpenShift [18]
Cluster	Swarm	Cluster
Pod	Task	Pod
Event	(Docker) Event	(OpenShift) Event
Namespace	Stack	Project
Admission Control	Third-party Plug-in	Admission Plug-in

5.3 Challenges in Implementation and Integration

Enabling Kubernetes Audit Logs for Model Learning. As Kubernetes audit logs (that are used for our model learning) are disabled by default, enabling that feature requires some efforts as follows. First, the audit log option has to be enabled by setting `--audit-log-path` to a directory with sufficient write permissions. Second, as audit logs are verbose by default, the resources (e.g., Pods) and methods (e.g., Create, Update) to be logged have to be limited by specifying an audit

policy file. Also, this change requires to restart the cluster.

Kubernetes API Reachability. OPA/Gatekeeper and PerfSPEC require access to the Kubernetes API for different reasons (such as, obtaining list of enforced policies, performing proactive verification, accessing existing Kubernetes resources). Since OPA/Gatekeeper runs inside a container and PerfSPEC is external to the cluster, none of them can directly reach the Kubernetes API. To overcome this issue, we add a `kube-proxy` sidecar container to the OPA/Gatekeeper deployment, and we use `kube-proxy` in the master node to give PerfSPEC an access to the Kubernetes API.

OPA Profiling Tool Reachability. The PerfSPEC *ranking* phase requires access to the OPA/Gatekeeper container to accurately profile the policy performance. Because PerfSPEC is external to Kubernetes, and OPA/Gatekeeper runs in a container, we use the Kube-API CLI (`kubectl exec -it`) to execute the profiling commands in the container.

PerfSPEC Reachability. The PerfSPEC *ranking* phase needs to receive input from OPA/Gatekeeper. To that end, we modify and recompile the OPA/Gatekeeper code to query PerfSPEC with such information every time a policy is used. Specifically, upon enforcement of a policy, OPA/Gatekeeper makes an HTTP POST request to our solution endpoint, implemented using a Flask web server and exposed in the Kubernetes cluster using a ClusterIP as endpoint.

Intercepting Events at Runtime. As PerfSPEC aims at reducing the policy verification and enforcement time, we find a solution to minimize the delay between the time when user requests reach the Kubernetes API and the time when those requests can be intercepted. To that end, we register PerfSPEC as a Kubernetes admission controller such that it can intercept the requests as early as OPA/Gatekeeper.

Feeding Watchlist Contents to OPA/Gatekeeper. We use OPA/Gatekeeper for watchlist verification and policy enforcement (as discussed in Section 5.2). However, OPA/Gatekeeper does not offer the possibility to simply pass policy parameters (e.g., watchlist content) as inputs. To overcome that issue, we develop a method for encoding the PerfSPEC watchlists content in the YAML format of a standard Constraint file of OPA/Gatekeeper. We can then feed such encoded watchlists to OPA/Gatekeeper as specified by the Constraint CRD (e.g., command `kubectl apply -f constraint.yaml`).

Measuring Response Time. Even though our experiments require to measure the response time at the OPA/Gatekeeper-level, being run on a container, it is impractical to access the process and attach a debugger from outside the cluster. To overcome, we enable a metrics logging feature in the OPA/Gatekeeper source code that reports response time.

6 Experimental Evaluation

In this section, we evaluate our solution through experiments.

6.1 Ranking Phase Evaluation

In the following, we evaluate the *ranking* phase of PerfSPEC. To conduct our analysis, we leverage a collection of 31 OPA/Gatekeeper security policies obtained from two major policy libraries. Of these, 26 policies have been sourced from the OPA/Gatekeeper official policy library [14], four policies come from the Red Hat Communities of Practice public repository [15], and one has been contributed by us to address CVE 2020-8554, as outlined in Section 1. We categorize those policies into five distinct profiles, namely: *Network*, *Image Security*, *Resource Usage*, *Access Control* and *Storage*.

Evaluation of the Dataset of Policies. In this experiment, we evaluate the response time of policies enforced by OPA/Gatekeeper (i.e., without our proactive approach) in our collection. Fig. 13 depicts the results of ranking the policies according to their response time using the PerfSPEC *ranking* phase. We group the result for the five profiles (i.e., all policies) under the *General* profile (Fig. 13 top left corner). Results show that the distribution of response time loosely follows a power law. More precisely, we observe that in most profiles (including the *General* profile), around 20% of the policies are responsible for approximately 80% of the total response time of the profile. Only the *Image Security* and the *Storage* profiles do not seem to follow that power law. For the *Storage* profile, the interpretation of results is difficult as only two policies are evaluated. Also, for the *Image Security* profile, it can be observed that the average response time of its policies are much lower than for other profiles in any case (approximately 1.5 ms against more than 10 ms for other profiles). We can conclude that, in the general case, making only a minority of top policies proactive would lead in major savings in total response time, therefore showcasing the importance of using PerfSPEC *ranking* phase.

Impact of PerfSPEC's Ranking on Policy Response Time. In this experiment, the quantitative impact of the *ranking* phase of PerfSPEC on the overall response time is evaluated. Fig. 14 illustrates the gain in response time as a function of the number of policies proactivized for the *Network* profile. We evaluate the policies response time for three different sizes of cluster: small (10 resources), medium (100 resources) and large (1,000 resources). We compare the optimal ranking achieved by PerfSPEC with both a random ranking (averaged over 100 runs) and a manual ranking performed by five graduate students familiar with Kubernetes, but not familiar with individual policies. According to the observations, by utilizing the policy ranking mechanism provided by PerfSPEC, it is possible to achieve an optimal average response time by proactively addressing merely four policies, while it takes students six policies to achieve the same gain. Conversely, proactivizing policies in a random manner leads to a much lower rewards, on average, for the same level of effort. When four or less policies are proactivized, using the manual ranking offers barely the same gain in response time as ranking the policies randomly, showcasing how difficult it is to do a meaningful manual ranking solely based on prior knowledge such as policy description and code, without automatic profiling and ranking tool. When the size of the cluster is substantial, the positive influence of our ranking in reducing the effort needed to reach lower average response time is more significant. For instance, in clusters of large size, proactivizing only four policies reduces the overall response time from 99 ms to less than 2 ms using our ranking, but reduces it to only 66 ms with a random ranking, and 51 ms with a manual ranking. This represents an improvement of 98% using our ranking versus less than 34% and 49%, respectively.

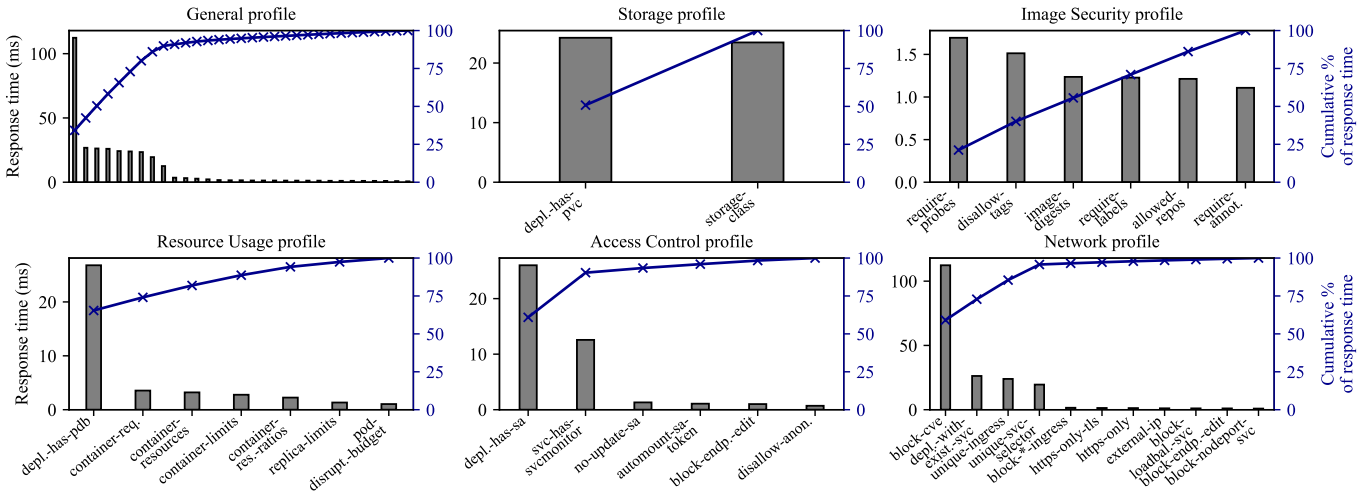


Fig. 13. Overall evaluation of the dataset of policies used for the ranking experiments

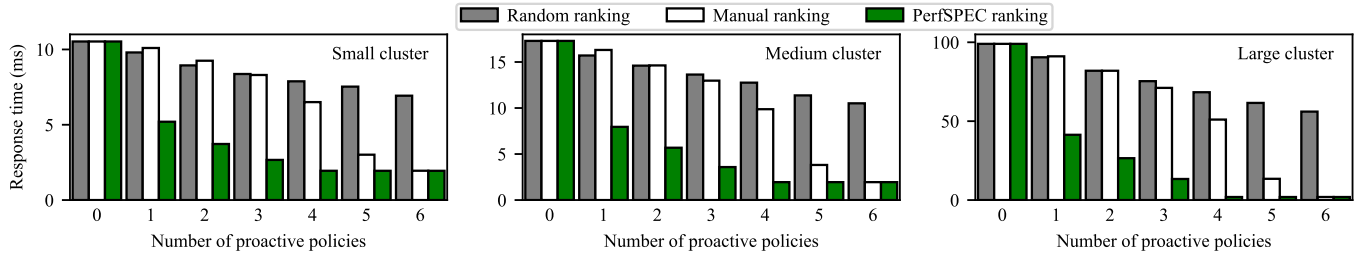


Fig. 14. Impact of ranking on average response time of policies (*Network* profile)

6.2 Learning Phase Evaluation

In this section, we evaluate the *learning* phase of PerfSPEC.

Offline Learning Time. We measure the offline learning time, i.e., the time required for deriving a predictive model from historical data. Specifically, we measure the time to extract the event sequences from the processed logs and to learn the predictive model using the Bayesian network library *pgmpy* [41]. The *log processing* task, done by Logstash, is not considered as it is performed in parallel of the logs collection.

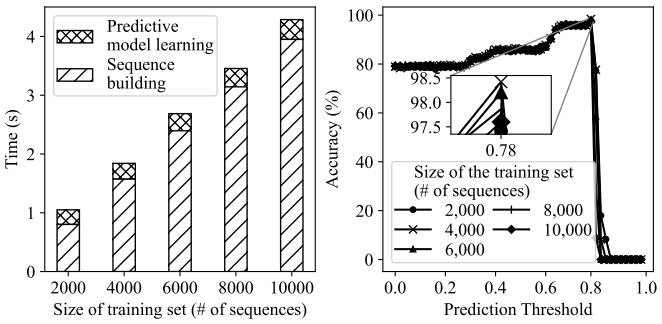


Fig. 15. Learning time and accuracy of our predictive model using the Bayesian learning technique

Fig. 15 shows the time required by the *predictive model builder* module of PerfSPEC to build sequences from the logs and learn a predictive model, while the number of event sequences varies from 2,000 to 10,000. We can see that the time required to perform both of those offline learning steps shows an upward linear trend. The linear trend is less pronounced for the predictive model learning than for the sequence building, as the time needed for the former is much less than that is needed for the latter. For instance, the time required for model learning increases almost linearly from 248 ms to 337 ms with the increasing number of sequences, whereas the time required for sequence building is growing from 801 ms to 3,950 ms. This has a practical implication since the more expensive sequence building only needs to be performed once, while the less expensive model learning may need to be repetitively performed (e.g., when new event sequences are added to the training data). Finally, the overall time reaches about 4 seconds for 10,000 event sequences, which is reasonable especially considering this is an offline step performed only periodically.

Model Accuracy at Runtime. This series of experiments aims to evaluate the relationship between the accuracy, prediction threshold values, and size of the training dataset during the PerfSPEC *runtime* phase. The selected prediction threshold for the critical events determines whether a pre-computation will be initiated or not. Since the accuracy of our model depends on the correct prediction of critical events, it is essential to demonstrate the impact of the prediction threshold value on the overall accuracy. Note that the optimal accuracy and corresponding prediction threshold value may vary across different predictive models. The accuracy is measured for different sizes of dataset, from 2,000 to 10,000 event sequences. During training, 80% of each dataset is utilized, while the remaining 20% is used for testing. We define the accuracy as the number of correct predictions divided by the total number of predictions made.

Furthermore, Fig. 15 shows the accuracy as a function of prediction threshold values for different sizes of dataset. We find that the best rate for the utilized model (as depicted in Fig. 6b) reaches 98.4% for a prediction threshold value of 0.78 and a training dataset of 4,000 sequences. Nevertheless, slight variations between different training sets are noted, indicating that a size of training set larger than 2,000 does

not significantly improve the accuracy. Prediction threshold values that are above 0.78 results in an accuracy of 0%, as no pre-computation is ever done. On the other hand, varying the threshold from 0 to 0.78 results in increasingly better accuracy values. Higher threshold values eventually mean that our solution will only perform pre-computations when the predicted events are highly likely to happen.

Comparison of Learning Techniques. In this experiment, we compare the three different learning techniques offered by PerfSPEC and highlight their pros and cons. We focus our interest on three different metrics, i.e., model accuracy, time needed for learning (offline learning time), and time needed for prediction at runtime (runtime inference time). We consider a prediction threshold value of zero (i.e., we always perform pre-computation) for these measurements, and the same accuracy calculation method has been employed for the three models in order to ensure the consistency of the results.

The results of these experiments are reported in Table 4. Overall, both LSTMs and n-gram present a better prediction accuracy than the Bayesian approach. Particularly, the LSTM models with a window size of one event and two events both score more than 90% accuracy, with the drawback of larger learning time (more than 20 seconds compared to 4.3 seconds for the Bayesian network and less than 70 ms for the n-grams). Additionally, LSTM models both depict a larger inference time of around 60 ms while other models typically take less than one millisecond. On the other hand, the 3-gram approach (i.e., a window size of two events) proves particularly efficient both from the accuracy and the timing point of view.

TABLE 4
Comparison of different predictive model learning approaches

Approach	Size of Window	Accuracy	Offline Learning Time	Runtime Inference Time
Bayesian Network	*N.A.*	79.7%	4.29 s	1e-4 s
LSTM	1	92.3%	24.01 s	0.06 s
	2	97.6%	32.75 s	0.07 s
n-grams	1	88.2%	0.01 s	1e-4 s
	2	97.3%	0.07 s	2e-4 s

6.3 Runtime Phase Evaluation

In this section, we evaluate the performance of our runtime phase where we consider the following policies.

- Policy *Block CVE-2020-8554* [16] is to block CVE-2020-8554 [5]. It prevents the creation of Services using the same *externalIP* as an already existing resource.
- Policy *Deployment with Existing Service* [45] ensures that Kubernetes Deployments are exposed with a corresponding Service. This can help avoid misconfigurations where a set of resources is not correctly exposed and ends up being unreachable by other resources.
- Policy *Unique Ingress* [46] verifies that Ingress rules are unique. It helps to mitigate misconfigurations where two different Ingress rules are applied to the same resources and potentially resulting in undefined behaviour.

Impact of the Size of the Cluster on Response Time. This set of experiments measures the response time of PerfSPEC. The response time is measured as the duration between the time PerfSPEC receives a critical request and the time PerfSPEC returns an enforcement decision to Kubernetes. As specified in Section 5.3, the response time is measured directly at the decision engine level (i.e., OPA/Gatekeeper) to avoid any overhead due to external factors. For this experiment, we consider an environment that is not under stress (i.e., we have enough time to pre-compute between two requests). Therefore, the prediction threshold for the pre-computation is set to zero (i.e., we always choose to pre-compute for critical events). A scenario in a stressed environment is presented later in this section.

Fig. 16 shows the comparison of the response time between PerfSPEC and OPA/Gatekeeper to enforce three different policies (taken from the *Network* profile described in Section 6.1), namely, *Block CVE-2020-8554* [5], *Deployment with Existing Service* [45] and *Unique Ingress* [46]. Particularly, to enforce the policy *Block CVE-2020-8554*, PerfSPEC maintains a near-constant response time of less than 15 ms when we vary the size of the cluster (# of Pods) for both a single request of one resource and a batch request of 100 resources. Conversely, it can be seen that the response time using OPA/Gatekeeper grows almost linearly in the size of the cluster. This behaviour can partially be explained by the reactive nature of OPA/Gatekeeper, i.e. performing the time-consuming operation of gathering the IP addresses of all the existing Pods at runtime. For the largest size of cluster (800 Pods here) and for one resource, OPA/Gatekeeper takes up to 580 ms, whereas PerfSPEC takes only 15 ms (which is close to 40 times faster). The zoomed inset shows the PerfSPEC response times on a more precise scale for a single request and a batch request, measured at 7 ms and 10 ms, respectively.

Similarly, we compare the response time between PerfSPEC and OPA/Gatekeeper to enforce the *Unique Ingress* policy when we vary the size of the cluster (# of Ingress rules, as dictated by this policy) for both a single resource and a batch request of 100 resources. Although the response times of both PerfSPEC and OPA/Gatekeeper grow almost linearly, PerfSPEC still outperforms OPA/Gatekeeper in all cases (e.g., for the largest cluster, 15 ms by PerfSPEC vs. 29 ms by OPA/-Gatekeeper). Additionally, as discussed in Section 1, the delay caused by OPA/Gatekeeper (mainly due to its replication step) leads to inconsistencies between the replicated state and the actual state of the cluster (which may be exploited for security policy bypass). Whereas, PerfSPEC not only reduces the delay by up to 50% but also avoids the need for state replication and its security implications. Note that the response time for policy *Block CVE-2020-8554* is relatively longer than that for the *Unique Ingress* policy, because Pod objects are much more complex and their Kubernetes descriptions contain more details.

Those figures also show the impact of the type of the requests (either a single request for one resource, or a batch request for 100 resources) on the response time. In the case of policy *Block CVE-2020-8554*, the additional delay induced by the batch request is negligible with respect to the response time. In the case of the *Unique Ingress* policy, the additional 4 ms due to processing the batch request represents an overhead of about 50%. In both cases, we can see that the impact of batch request on OPA/Gatekeeper and PerfSPEC is similar, and PerfSPEC outperforms OPA/Gatekeeper for both types of requests.

Impact of Wrong Predictions on Response Time. The second set of experiments is to measure the impact of wrong predictions by our predictive model on the response time of PerfSPEC. For this purpose, we consider the case where a critical

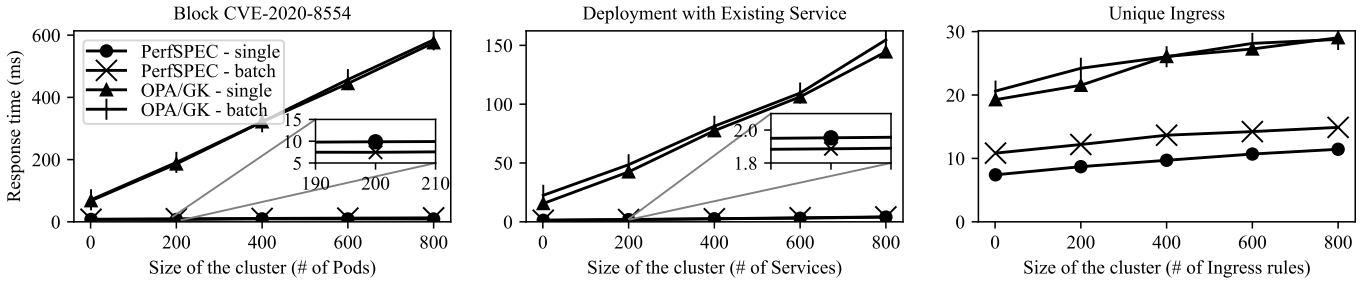


Fig. 16. Impact of the size of cluster on the response time

event occurs without being predicted by PerfSPEC, which has an impact on the response time as PerfSPEC would fall back to the intercept-and-check mode in this case (as described in Section 3). We measure the overall response time (which includes both the pre-computation time measured at PerfSPEC level and the verification time measured at OPA/Gatekeeper level). For this experiment, we vary the rate of wrong predictions in the model and use policy *Block CVE-2020-8554* for enforcement. We simulate 10,000 correctly predicted events and vary the rate of wrong predictions from 5% to 40% (note a rate of wrong predictions of more than 40% is unlikely in practice) by injecting unexpected events randomly into the event sequences.

Fig. 17 shows the average overall response time (incurred in pre-computation as well as in verification by PerfSPEC for enforcing policy *Block CVE-2020-8554*) in case of different (simulated) wrong predictions rates. As a baseline, in Fig. 16 (without simulated errors), the response time is around 12 ms for 800 Pods. In contrast, Fig. 17 shows that, even with a 40% error rate, the response time of PerfSPEC still stays below 125 ms for 800 Pods, which is better than the performance of OPA/Gatekeeper in the same environment (580 ms, see Fig. 16). As the error rate is likely much lower in reality (see Fig. 15), we can conclude that wrong predictions will not significantly affect the effectiveness of PerfSPEC.

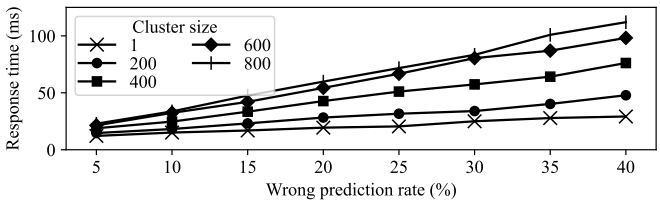


Fig. 17. Impact of wrong event predictions on the response time for policy *Block CVE-2020-8554*

Impact of Prediction Threshold on Response Time. The third set of experiments is to measure the impact of different prediction threshold values (as described in Section 3) on the response time as well as on the pre-computation efficiency of PerfSPEC. We realize these experiments in a stressed environment, i.e., requests arrive one after another without delay. As a result, since the pre-computing step is blocking, spending time to pre-compute for one request can impact the response time of the next request. We vary the value of the critical events prediction threshold from 0.0 to 1.0 and measure the response time of three policies (namely, *Block CVE-2020-8554*, *Deployment with Existing Service* and *Unique Ingress*) for a cluster size of 200 resources and single requests. The *pre-compute usefulness* is measured as the ratio of the number of pre-computations that are useful (in the sense that the predicted events eventually happen) over the total number of pre-computations. The *no pre-compute usefulness* is the ratio of the number of times we make the correct decision to not pre-compute (in the sense that the event eventually does not happen) over the total number of times we do not pre-compute. In this experiment, we deliberately avoid traditional accuracy metrics (e.g., precision, recall), as use of those metrics might be misinterpreted as the accuracy of PerfSPEC security; whereas this experiment instead measures the usefulness of its pre-computation step.

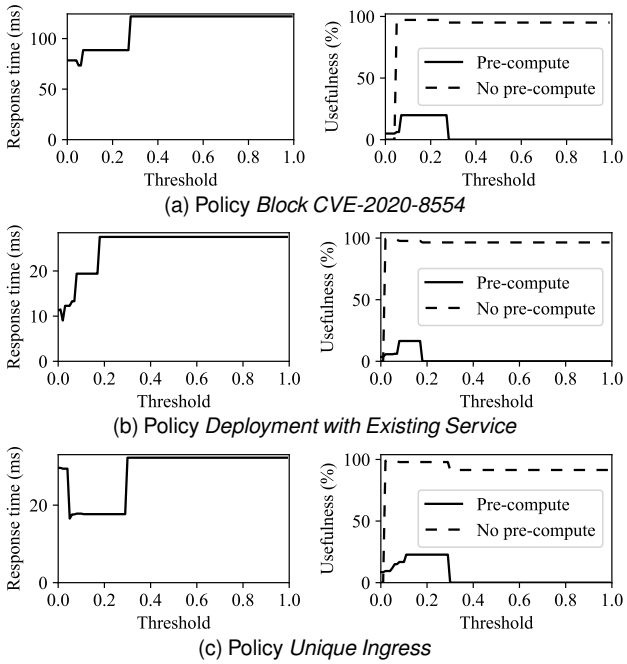


Fig. 18. Impact of prediction threshold on response time and pre-computation efficiency for three policies

Fig. 18a, Fig. 18c and Fig. 18b show the response time of PerfSPEC and the aforementioned usefulness metrics as functions of the prediction threshold. As an example, Fig. 18a shows the average response time stays almost constant for threshold values above 0.23 (0.23 being the highest transition probability to a critical event existing in the predictive model for this policy). For values above 0.23, the average response time is the highest at more than 120 ms, since we never pre-compute and have to perform the verification at runtime under such prediction threshold values. For threshold values below 0.03, the response time peaks at 76 ms, since we always pre-compute but often unnecessarily (for events that will not happen). Between values of 0.03 and 0.23, we observe the lowest response time. Precisely, a threshold value of 0.05 reduces the average response to a minimum of 68 ms. Similar behaviours can be observed in Fig. 18b and Fig. 18c. Thus, if necessary, an optimal prediction threshold value can be determined based on the given policy and training data. Note

that, in the general case (i.e., the environment is not under stress and we have enough time to pre-compute between two requests), the prediction threshold value should be zero.

Impact of the Rate of Critical Events. This fourth set of experiments investigates further the impact of the rate of critical events on the efficiency of our solution. Fig. 19 shows the amount of pre-computation that is missed, the average additional delay per critical event, and the corresponding relative overhead, as a function of the rate of critical events in our environment (per minute). Our solution uses event prediction to pre-compute policy results for critical events. As the rate of critical events in the system increases, PerfSPEC is left with a smaller time window to pre-compute policy results before the next critical event occurs. As a result, some pre-computations can eventually be missed as the critical event happens before the pre-computation step is over. In such cases, PerfSPEC still enforces the policy by computing the actual policy results on the fly, resulting in additional delay. Note that this delay in pre-computation depends on the Kubernetes admission controller and is not an end-user choice. We vary the rate of critical events between 5 per minute (measured during normal cluster operation) and 210 per minute (extreme cluster usage), for the three policies previously studied. In each case, the rate of pre-computation misses follows a logarithmic increase from less than 1% (enough time to pre-compute) to between 10% and 17% of missed pre-computations in the extreme case. These incurs relatively low overhead on the response time with at most 1 ms (12%) for the *Block CVE-2020-8554* and *Unique Ingress* policies, and 0.3 ms (2.3%) for the *Deployment with Existing Service* policy.

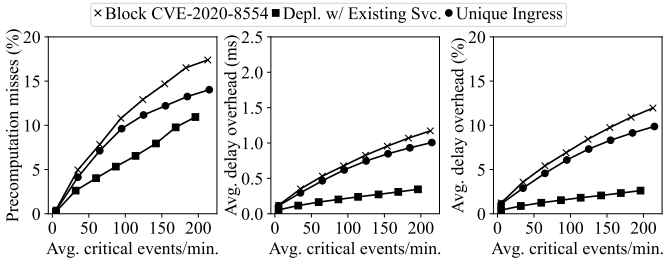


Fig. 19. Impact of the rate of critical events per minute on pre-computation misses (left), average additional delay in time (center) and average additional delay in percentage (right)

CPU/Memory Consumption. The fifth set of experiment reports on the resource consumption of our solution at runtime. Using Linux's `pidstat` [47] command, we measure the average CPU (% of both user and system CPU time) and memory consumption (virtual set size, in MB) of each of our PerfSPEC's runtime modules. Table 5 reports the results measured during the deployment of the 60 most popular Helm charts used to collect our dataset (described in Section 4), with all 31 policies enforced (described in Section 6.1). We additionally report metrics when using OPA/Gatekeeper alone (without our solution). The CPU and memory consumption of the *interceptor* module (registered as an admission webhook) remain relatively low (4.22% CPU, and 262.5 MB memory) as it does not actively process data. The *proactive verifier* can do expensive data collection and pre-computation (we observed peak CPU usage at 22.1%), however this does not concern every single event, making the average resource consumption quite low as well (4.38% CPU, and 115.2 MB memory). Finally, when used together with our solution, OPA/Gatekeeper's CPU consumption is greatly reduced (from 8.14% to 3.15%) since results are pre-computed by PerfSPEC, therefore reducing the processing load on OPA. On the other hand, the memory consumption rises from 839.6 MB to 1018.4 MB, as results are stored in memory to be rapidly accessible during policy verification. Overall, using PerfSPEC over OPA/Gatekeeper results in an increase in CPU and memory consumption of less than 4% and 560 MB, respectively. This is acceptable given the benefits in response time brought by our solution.

TABLE 5
CPU and memory usage of our solution compared to OPA/Gatekeeper

	PerfSPEC		OPA/Gatekeeper	
Modules	**CPU (%)**	**Memory (MB)**	**CPU (%)**	**Memory (MB)**
Interceptor Webhook	4.22	262.5	N/A	N/A
Proactive Verifier	4.38	115.2		
Runtime Policy Enforcer	3.15	1018.4	8.14	839.6

7 Discussion

This section discusses different aspects of PerfSPEC.

Efforts to Ensure Efficiency. As mentioned in Section 1, forcing synchronization might be a solution to the current replication issue in OPA/Gatekeeper. However, the required efforts to force synchronization at runtime appears to be inefficient (as depicted in Fig. 16). Alternatively, our solution incurs pre-computational efforts (which are performed offline) without causing any significant delay in response at runtime. Furthermore, to minimize the pre-computation effort in PerfSPEC, we adopt several mechanisms as follows: (i) We use an adjustable prediction threshold for each policy so that only the relevant events that are above the threshold are considered as a threat. Fig. 18 shows that using a threshold of 0 (i.e., considering all future events as a potential threat) indeed results in higher response time and lower usefulness, however, there exists an optimal threshold providing the lowest response time and highest usefulness. (ii) We use Bayesian network as a fast prediction model and evaluate its efficiency in inference in Table 4. (iii) Using our ranking phase, we selectively apply our proactive computing solution on policies that have a large impact on the cluster response time and/or the resource consumption.

Impact on Security. The ranking of the policies does not have any direct impact on the security of the system and all policies (regardless of their ranking) will always be enforced. The ranking helps to schedule the proactive verification of different policies where more (computationally) expensive policies are prioritized (as depicted in Fig. 14), so that the overall response time of our enforcement step can remain minimum (as shown in Fig. 16). As a result, if a policy is not proactivized, then we adopt an intercept-and-check approach to verify that policy at runtime before enforcement (which only affects the response time of our solution).

8 Related Work

In this section, we review relevant works.

Container Security Verification. Several works (e.g., [48], [49], [50]) on container security aim at verifying the security of container images (e.g., [48]) or their integrity (e.g., [49], [50]). For instance, [48] identifies and assesses vulnerabilities in Docker containers images, while both [49], [50] propose

solutions to attest the integrity of containers during their entire life cycle. Unlike them, PerfSPEC instead aims at enhancing the performances of security policy verification, including image security verification but also broader aspects such as network policy, role-based access control, etc.

Kubernetes Security. There exist solutions addressing different security aspects in Kubernetes. Authors in [51] give five security best practices for Kubernetes as follows: (i) API-based authentication and authorization request, (ii) network-specific and Pod-specific policies, restricting network communications and applying least privilege context to Pods, respectively, (iii) continuous security patches for the cluster, (iv) logging/-monitoring the cluster, and (v) continuous security compliance. PerfSPEC subscribes to the latter by proposing a proactive and efficient security compliance solution for container environments. In contrast, most existing works (e.g., [52], [53], [54]) propose reactive solutions (i.e., after-the-fact), giving the attacker a larger attack window. For example, *Sysdig* [52] provides a system-call level security attack detection approach while *Falco* [53] offers an online anomaly detection tool for containerized applications. *KubAnomaly* [54] is a learning-based anomaly detection system, providing runtime monitoring capabilities in Kubernetes. PerfSPEC differs from those works as it proactively prevents policy violations.

Security Policy Compliance. There are several proactive security compliance verification works (e.g., [22], [27], [55], [56]) for non-container environments (e.g., OpenStack [55] clouds). For instance, *Weatherman* [27] and Congress [55] verify security policies in OpenStack clouds using graph-based and Datalog-based models, respectively. Moreover in [1], a proactive protection approach for potential security breaches in cloud is proposed. Unlike our automated learning of predictive model, those require the user to provide future plans, which are not always accessible. *LeaPS* [22] and *Proactivizer* [57] are proactive security solutions for cloud environments. In contrast to our work, those are not specifically designed to tackle complexity and challenges of container environments. Additionally, all those works do not take into account the ranking of policies to decide the most efficient way to reduce response time and resource consumption.

A preliminary version of this paper introducing the basic idea of proactive security policy enforcement in containerized environments has been published in [8]. In this paper, we extend our previous work by enhancing the overall usability and scalability of our proactive security policy enforcement approach. First, we make it possible for large environments to adopt our proactive solution by designing a way to automatically evaluate and rank the performance of policies in Kubernetes clusters. Second, we improve our predictive model by refining our previous learning approach and introducing two more learning techniques. Third, we generate the first dataset of Kubernetes events generated from realistic deploy and tear-down operations. Fourth, we design and implement a new system architecture to integrate those new components into our framework based on Kubernetes. Finally, we extensively test our solution and its extensions on a collection of publicly available security policies, and with our new dataset.

In summary, our work mainly differs from the state-of-the-art works as follows. First, PerfSPEC automatically assesses the performance of security policies and ranks them to efficiently reduce the overall response time while keeping required proactivization effort low. Second, it provides a proactive security policy enforcement solution designed for container environments that prevents security compliance breaches and enhances response time. Third, PerfSPEC automatically captures dependencies among events in container environments and learns a predictive model to anticipate future critical events. Finally, it is integrated with one of the most popular policy enforcement frameworks, OPA/Gatekeeper, while offering the benefit of a proactive solution.

9 Conclusion

In this paper, we proposed PerfSPEC, a performance assessment and proactive security policy enforcement solution for container environments. We automated the process of profiling security policy enforcement and determining the most efficient way to improve their overall response time. We then leveraged learning techniques to derive a predictive model that captures dependencies among events in container environments. PerfSPEC utilized this model to predict future critical events and efficiently prevent security policy violations for large container environments with a practical response time (e.g., less than 15 ms for 800 Pods compared to 600 ms with one of the most popular existing approaches). Additionally, we implemented PerfSPEC and integrated it with Kubernetes.

Limitations and Future Work. First, PerfSPEC neither retrains nor tunes the model based on historical compliance and changes in user behavior. A future direction is to support dynamic online learning of the predictive model. Second, PerfSPEC is integrated with Kubernetes. A future direction is to integrate it with other container orchestrators. Finally, our solution is currently automating the process of assessing policies performance and ranking them, although it still requires some manual effort to identify critical events and where to bring proactivization within a policy itself. In the future, we plan to explore static policy analysis to identify performance bottlenecks in order to automate these tasks.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and suggestions. This work was supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under the Industrial Research Chair in SDN/NFV Security, and the Canada Foundation for Innovation under JELF Project 38599.

References

[1] S. S. Yau, A. B. Buduru, and V. Nagaraja, "Protecting critical cloud infrastructures with predictive capability," in *CLOUD*, 2015.
[2] D.-H. Luong, H.-T. Thieu, A. Outtagarts, and Y. Ghamri-Doudane, "Cloudification and Autoscaling Orchestration for Container-Based Mobile Networks toward 5G: Experimentation, Challenges and Perspectives," in *IEEE Vehicular Technology Conference*, 2018.
[3] "Kubernetes," 2021. [Online]. Available: https://kubernetes.io
[4] "'Azurescape' Kubernetes Attack Allows Cross-Container Cloud Compromise," 2021. [Online]. Available: https://threatpost.com/azurescape-kubernetes-attack-container-cloud-compromise/169319
[5] "CVE-2020-8554: Man in the Middle," 2020. [Online]. Available: https://blog.champtar.fr/K8S_MITM_LoadBalancer_ExternalIPs
[6] "Open Policy Agent/Gatekeeper," 2019. [Online]. Available: https://open-policy-agent.github.io/gatekeeper

[7] "AWS Container Security 2020 Survey," 2022. [Online]. Available: https://aws.amazon.com/blogs/containers/results-of-the-2020-aws-container-security-survey

[8] H. Kermabon-Bobinnec, M. Gholipourchoubeh, S. Bagheri, S. Majumdar, Y. Jarraya, M. Pourzandi, and L. Wang, "ProSPEC: Proactive Security Policy Enforcement for Containers," in *ACM CODASPY*, 2022.

[9] A. Sriraman and T. F. Wenisch, "{μTune}:{Auto-Tuned} threading for {OLDI} microservices," in *USENIX OSDI*, 2018.

[10] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *ASPLOS*, 2021.

[11] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, "Serverless computing for container-based architectures," *Future Generation Computer Systems*, vol. 83, pp. 50–59, 2018.

[12] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. R. Yu, and T. Huang, "When serverless computing meets edge computing: Architecture, challenges, and open issues," *IEEE Wireless Communications*, vol. 28, no. 5, pp. 126–133, 2021.

[13] "CVE-2021-43979," 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-43979

[14] "OPA/Gatekeeper Policy Library," 2022. [Online]. Available: https://github.com/open-policy-agent/gatekeeper-library

[15] "RedHat Rego Policies," 2022. [Online]. Available: https://github.com/redhat-cop/rego-policies

[16] Abir Hamzi, "Stop CVE-2020-8554," 2020. [Online]. Available: https://github.com/AbirHamzi/gatekeeper-library/tree/CVE-2020-8554/library/cve/CVE-2020-8554

[17] "Docker Swarm," 2021. [Online]. Available: https://docs.docker.com/engine/swarm

[18] "OpenShift," 2021. [Online]. Available: https://docs.openshift.com

[19] ETSI, "Report on the Enhancements of the NFV architecture towards "Cloud-native" and "PaaS", ETSI GR NFV-IFA 029," ETSI, Tech. Rep., 2019.

[20] "Docker," 2021. [Online]. Available: https://docker.com

[21] "Containerd," 2022. [Online]. Available: https://containerd.io

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

[23] K. W. Ullah, A. S. Ahmed, and J. Ylitalo, "Towards Building an Automated Security Compliance Tool for the Cloud," in *IEEE TrustCom*, 2013.

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

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

[26] S. Majumdar, G. S. Chawla, A. Alimohammadifar, T. Madi, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi, "ProSAS: Proactive security auditing system for clouds," *IEEE TDSC*, vol. 19, no. 4, pp. 2517–2534, 2021.

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

[28] F. Doelitzscher, C. Fischer, D. Moskal, C. Reich, M. Knahl, and N. Clarke, "Validating cloud infrastructure changes by cloud audits," in *IEEE World Congress on Services*, 2012.

[29] "How to use OPA/Gatekeeper," 2023. [Online]. Available: https://open-policy-agent.github.io/gatekeeper/website/docs/howto

[30] R. E. Neapolitan *et al.*, *Learning Bayesian networks*. Pearson Prentice Hall Upper Saddle River, NJ, 2004, vol. 38.

[31] M. Jardino, "Multilingual stochastic n-gram class language models," in *IEEE ICASSP*, 1996.

[32] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[34] "Helm charts at ArtifactHub," 2022. [Online]. Available: https://artifacthub.io

[35] R. D. Hipp, "SQLite," 2020. [Online]. Available: https://sqlite.org/index.html

[36] Grinberg, Miguel, *Flask web development: developing web applications with Python*. O'Reilly Media, Inc., 2018.

[37] "OPA/Gatekeeper source code," 2023. [Online]. Available: https://github.com/open-policy-agent/gatekeeper/blob/master/vendor/github.com/open-policy-agent/frameworks/constraint/pkg/client/drivers/rego/driver.go#L244

[38] "Linux's htop," 2020. [Online]. Available: https://man7.org/linux/man-pages/man1/htop.1.html

[39] "Logstash," 2021. [Online]. Available: https://elastic.co/logstash

[40] Wes McKinney, "Data Structures for Statistical Computing in Python," in *SCIPY*, 2010.

[41] A. Ankan and A. Panda, "pgmpy: Probabilistic graphical models using python," in *SCIPY*, 2015.

[42] Bird, Steven and Loper, Edward and Klein, Ewan, *Natural Language Processing with Python*. O'Reilly Media, Inc., 2009.

[43] Chollet, Francois and others, "Keras," 2015. [Online]. Available: https://keras.io

[44] "Docker authorization with OPA," 2021. [Online]. Available: https://openpolicyagent.org/docs/latest/docker-authorization

[45] "Deployment with Existing Service policy," 2023. [Online]. Available: https://github.com/redhat-cop/rego-policies/tree/master/policy/ocp/requiresinventory

[46] "Unique Ingress Host policy," 2022. [Online]. Available: https://github.com/open-policy-agent/gatekeeper-library/tree/master/library/general/uniqueingresshost

[47] "pidstat Linux command," 2022. [Online]. Available: https://linux.die.net/man/1/pidstat

[48] W. S. S. Ahamed, P. Zavarsky, and B. Swar, "Security Audit of Docker Container Images in Cloud Architecture," in *ICSCCC*, 2021.

[49] M. De Benedictis and A. Lioy, "Integrity verification of Docker containers for a lightweight cloud environment," *Future Generation Computer Systems*, vol. 97, pp. 236–246, 2019.

[50] W. Luo, Q. Shen, Y. Xia, and Z. Wu, "Container-IMA: a privacy-preserving integrity measurement architecture for containers," in *RAID*, 2019.

[51] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices," in *SecDev*, 2020.

[52] "Sysdig," 2018. [Online]. Available: https://sysdig.com

[53] "Falco," 2018. [Online]. Available: https://falco.org

[54] C.-W. Tien, T.-Y. Huang, C.-W. Tien, T.-C. Huang, and S.-Y. Kuo, "KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches," *Engineering Reports*, vol. 1, no. 5, p. 12080, 2019.

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

[56] 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.

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

Hugo Kermabon-Bobinnec Hugo Kermabon-Bobinnec is a Ph.D. student at the Concordia Institute for Information Systems Engineering, Concordia University. He previously obtained his M.A.Sc in Information Systems Security from Concordia University. Currently, his research interests include OS security, cloud-computing, and container security.

Sima Bagheri Sima Bagheri received her B.Sc. and M.Sc. degrees from Shahid Beheshti University, Tehran, Iran. Since 2020, she has been a research assistant at Concordia University. Her research interests include cloud computing and security, applied machine learning in security, and usable security and privacy.

Mahmood GholipourChoubeh Mahmood GholipourChoubeh earned his second Master's degree in Information Systems Security from Concordia University in 2023. He previously obtained a Master's degree in Computer Engineering from the University of Tehran, Iran, in 2013. His research interests include 5G/edge security, SDN/NFV/network security, and privacy.

Suryadipta Majumdar Suryadipta Majumdar is currently an Associate Professor in the Concordia Institute for Information Systems Engineering, Concordia University, Montreal. He received his Ph.D. on cloud security auditing from Concordia University. His research mainly focuses on cloud security, Software Defined Network (SDN) security and IoT security.

Yosr Jarraya Yosr Jarraya is a Master Researcher at Ericsson since 2016 focusing on security and privacy in cloud, SDN and NFV. She received a Ph.D. in electrical and computer engineering from Concordia University, Montreal, in 2010. She co-authored two books and over 40 research papers in peer-reviewed international journals and conferences.

Lingyu Wang Lingyu Wang is a professor in the Concordia Institute for Information Systems Engineering at Concordia University, Montreal. He holds the NSERC/Ericsson Industrial Research Chair in SDN/NFV Security. He received his Ph.D. degree in Information Technology in 2006 from George Mason University. His research interests include SDN/NFV security, cloud computing security, software security, and privacy.

Makan Pourzandi Makan Pourzandi received a M.Sc. in parallel computing from Ecole Normale Supérieure de Lyon, France, and a Ph.D. in computer science from the University of Lyon I, France. He is currently a Researcher with Ericsson, Canada. He has over 15 years of experience in security for telecom systems, cloud computing, distributed systems security, and software security.