ProSPEC: Proactive Security Policy Enforcement for Containers

ABSTRACT

By providing lightweight and portable support for cloud native applications, container environments have gained significant momentum lately. A container orchestrator such as Kubernetes can enable the automatic deployment and maintenance of a large number of containerized applications. However, due to its critical role, a container orchestrator also attracts a wide range of security threats exploiting misconfigurations or implementation flaws. Moreover, enforcing security policies at runtime against such security threats becomes far more challenging, as the large scale of container environments implies high complexity, while the high dynamicity demands a short response time. In this paper, we tackle this key security challenge to container environments through a proactive approach, namely, *ProSPEC*. Our approach leverages learning-based prediction to conduct the computationally intensive steps (e.g., security verification) in advance, while keeping the runtime steps (e.g., policy enforcement) lightweight. Consequently, ProSPEC can ensure a practical response time (e.g., less than 10 ms in contrast to 600 ms with one of the most popular existing approaches) for large container environments (up to 800 Pods). We demonstrate the deployability by integrating ProSPEC with Kubernetes, one of the most popular container orchestrators.

1 INTRODUCTION

Container environments are becoming increasingly popular for delivering microservices with increased scalability, reliability and observability [41]. In such environments, container orchestrators (e.g., Kubernetes [18]) are typically employed to ease the deployment and maintenance of large amounts of containerized applications.[1] However, the central role of such orchestrators also renders them attractive to various security threats that exploit misconfigurations or vulnerabilities to cause breaches of security policies. Furthermore, security is typically an afterthought in the deployment of containerized applications and security policy breaches are usually detected after the fact, which could result in irreversible damages (e.g., denial of service or information leakage) [10, 12].

To that end, enforcing security policy at runtime (i.e., verifying user requests against a given security policy and denying those requests causing a breach) can prevent such irreversible damages caused by attacks. However, runtime security policy enforcement can be challenging for container environments due to their sheer scale (which implies high complexity) combined with the very short life cycle of containers (which demands short response times). Evidently, applying Open Policy Agent (OPA)/Gatekeeper [5] (the former is an open-source policy engine, and the latter the go-to solution for using OPA for Kubernetes admission control [18]) for runtime security policy enforcement in large container environments may face some practical challenges as follows.

- First, such tools may cause prohibitive runtime delay for a relatively large container environment (e.g., OPA/Gatekeeper can cause up to 600 ms delay in a Kubernetes cluster of 800 Pods, as shown in our experiments in Section 5.2).
- Second, the reactive nature of those solutions (i.e., all the efforts are only started after a user request is already received) implies a fundamental bottleneck that leaves little room for further performance improvement to keep up with the ever-increasing size and complexity of container environments.
- Third, existing proactive approaches to reduce response time, such as verifying replicated states [5] (instead of actual system states) may cause severe security issues. Specifically, the small delay in replicating the states can cause a temporary inconsistency between the actual and the replicated states, which can be exploited by a malicious user to bypass security policies, as we will show through an example in Section 2.2.

In this paper, we tackle those key challenges through a proactive approach, namely, *ProSPEC*. Our key idea is to perform computationally intensive verification steps in advance (i.e., before the actual events occur) to keep the runtime enforcement steps lightweight with a practical response time. Specifically, we first learn a predictive model from historical data (i.e., logs of past events) to enable the prediction of future events. Then, we utilize this model to predict imminent critical events (which may violate a security policy) and proactively start the verification of that policy based on those hypothetical events. Finally, once the actual events occur, we enforce the security policy based on the pre-computed verification results through efficient operations such as list searching. Consequently, ProSPEC can make runtime decisions with negligible delay even for large container environments.

In summary, our main contributions are as follows:

- To the best of our knowledge, this is the first work offering proactive security policy enforcement at runtime for containers. ProSPEC can ensure security policy enforcement for large container environments with a practical response time (e.g., less than 10 ms for 800 Pods in contrast to 600 ms with one of the most popular existing approaches).
- We study the dependency relationships among container management events and build the first predictive model for container events. Such a model may enable other proactive security solutions (beyond security policy enforcement).
- ProSPEC can be easily integrated into legacy policy enforcement engines without major modification as verification and enforcement are decoupled in ProSPEC.
- ProSPEC is integrated with the de facto standard orchestrator, Kubernetes [18], with the provision of porting it to other orchestrators (e.g., Docker Swarm [15], OpenShift [22]).

The rest of this paper is organized as follows: Section 2 presents the preliminaries and models. Section 3 details the ProSPEC approach. Sections 4 and 5 provide the implementation details and experimental evaluation, respectively. Section 6 summarizes the related work, and finally, Section 7 concludes the paper and considers future works.

[1] A study shows 91% of respondents use Kubernetes and 83% of them in production [8].

2 BACKGROUND AND MOTIVATION

This section provides a background on containerization and security policy compliance, presents the motivation, and defines our threat model.

2.1 Background

Containerization. Cloud computing environments present numerous advantages including scalability, reliability and observability for application deployment. Frameworks such as OpenStack [1] allow companies to deploy their own cloud infrastructure over virtual machines (VMs) [41]. However, due to the lack of portability and the significant overhead imposed by VMs (the operating system), containerization has recently become a preferred option for dynamic and quickly evolving environments.

As demonstrated in the ETSI container environment [6] shown in Fig. 1, 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 and run directly at the OS level, thus resulting in much faster deployments and less resource consumption. As a hypervisor manages resources and VMs in hardware virtualization, a container orchestrator (e.g., Kubernetes) indirectly manages containers (i.e., via a container runtime environment such as Docker) through their entire life cycle, including scheduling, deployment, patch and deletion. Depending on the orchestrator, containers can be managed and gathered in group (e.g., in Pods, in the case of Kubernetes).

Security Policy Compliance. In a container environment, security policy compliance means to first verify requests made to the orchestrator against a set of security policies, and then enforce the decision based on the verification result (i.e., allow or deny). As shown in Fig. 1, a policy compliance tool ensures the security by verifying the requests and enforcing the decisions.

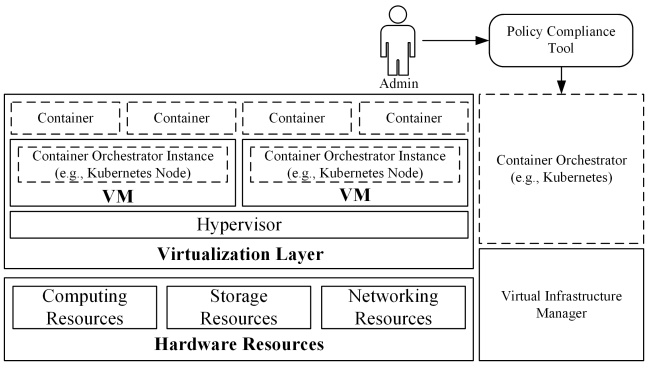


Figure 1: The ETSI architecture of a container environment [6]

2.2 Motivating Example

To make our discussion more concrete, our motivating example will be based on Kubernetes [18] as the container orchestrator and OPA/Gatekeeper [5] as the policy compliance tool. A key limitation of OPA/Gatekeeper lies in its reactive nature, i.e., it can only start the data collection and policy verification after a user request has already been received. Consequently, a user would have to experience both the *Data Collection Delay* (which grows linearly in the amount of data required) and the *Policy Verification Delay* (which depends on the number of policies and their complexity), which could become prohibitive for large container environments.

As a remedy for such undesirable delays, OPA/Gatekeeper employs *data replication* by monitoring resource changes in the Kubernetes cluster and keeping its own copy of the cluster state for faster data collection and verification. However, the data replication causes an unavoidable delay that can lead to inconsistencies between the actual state of a Kubernetes cluster and the state replicated by OPA/Gatekeeper. As we will show next, such inconsistencies can be exploited by adversaries to bypass security policies.

Specifically, our attack scenario is based on a real-world vulnerability in Kubernetes, CVE-2020-8554 [10]. This vulnerability allows an adversary to set the *externalIP* field of a newly created Kubernetes Service to be identical to the *IP* address of an existing resource such as a Pod (normally, OPA/Gatekeeper would only allow a new Service to be set to an *IP* address not already used in the Kubernetes cluster). The attacker can then employ this Service to intercept the traffic directed to that resource (e.g., to eavesdrop sensitive information).

As shown in Fig. 2, a `Create Pod` request (1) is made to the Kube-API server. The Kubernetes admission webhook receives the request and forwards it to the admission controller (i.e., OPA/Gatekeeper) for verification (2). OPA/Gatekeeper compares the request against its pre-defined security policy and allows the `Create Pod` request (3). In (4), the Pod is created in the Kubernetes cluster with the *IP* address `192.168.1.1`. Shortly after, while OPA/Gatekeeper is replicating the cluster state, the malicious user makes a `Create Service` request to the Kube-API server (5) after α time which is less than the data replication delay (i.e., β). Therefore, OPA/Gatekeeper's replicated cluster state is not yet updated with the freshly created Pod. When processing (6), OPA/Gatekeeper does not detect any policy breach and allows the Service creation with an *externalIP* equal to the existing *IP* address `192.168.1.1` (7). As a result, in the actual cluster, a Service exists with the same *externalIP* address as a Pod, giving it the ability to intercept that Pod's traffic (8). Thus, the data replication delay leads to the security policy bypass, and represents a critical security issue in the Kubernetes cluster.

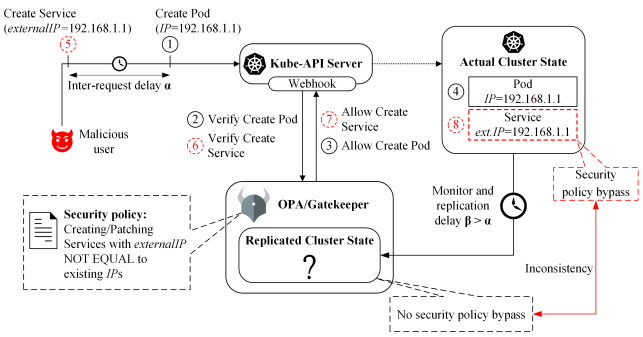


Figure 2: Policy bypass due to data replication delay

In summary, the reactive nature of OPA/Gatekeeper can imply significant runtime delays for normal users, whereas the data replication solution used by OPA/Gatekeeper to reduce such delays could lead to severe security breaches such as policy bypass. Those limitations motivate us to depart from such a reactive solution

and propose instead a proactive approach that performs the data collection and verification *before* user requests arrive.

2.3 Threat Model

In-scope Threats. The in-scope threats include both external attackers and insiders (such as other users of the cluster) with malicious intents. We assume that the container environment may have implementation flaws, misconfigurations, or vulnerabilities that may allow such adversaries to violate given security policies. As ProSPEC focuses on Kubernetes API requests, we limit our scope to attacks that involve sequences of operations directed through the Kubernetes API server interface. Like most existing works on security verification, we assume the integrity of ProSPEC and the Kubernetes system (with its API requests, events, audit logs, and database records), protected with existing trusted computing techniques such as remote attestation [25, 31].

Out-of-scope Threats. As ProSPEC focuses on providing security compliance at the front line of the cluster, i.e., Kubernetes control plane, other related issues such as Docker container security and detecting specific attacks or intrusions are out of the scope of this paper. The out-of-scope threats also include attacks that can completely bypass the Kubernetes API server interface, and attacks that do not involve any Kubernetes API requests. Moreover, as with most works on security verification, we do not consider attackers who can temper with (either through attacks or by using insider privileges) the Kubernetes system or the ProSPEC solution itself.

3 PROSPEC APPROACH

Fig. 3 shows an overview of our approach, which contains two major phases: *offline* and *runtime*. During the *offline* phase, ProSPEC builds a predictive model that captures the (probabilistic) dependency relationships among events in the container environment to enable prediction of future events. During the *runtime* phase, ProSPEC first conducts proactive verification against security policies (provided by ProSPEC users, such as administrators) for predicted future events by utilizing the built models, and then enforces those proactive verification results when actual events occur. In the following, we elaborate on both phases.

3.1 The Offline Phase

In this section, we first informally define our predictive model and then describe how ProSPEC builds this model.

Defining Predictive Model. Our predictive model is to capture the probabilistic dependencies among management events in a container environment; which will be used in subsequent steps of the ProSPEC approach. This model is represented as a directed graph where nodes indicate container events, edges indicate their transitions, and labels on edges indicate the probabilities of a transition. This model includes two types of dependencies between events in a container environment: (i) *inter-resource dependency*: the dependencies among its different resources, and (ii) *intra-resource dependency*: the dependencies within one resource. For example, Fig. 4a shows an example of inter-resource dependency relationship in Kubernetes [18], a major container orchestrator, where a Pod resource cannot be created unless a Namespace resource exists, and

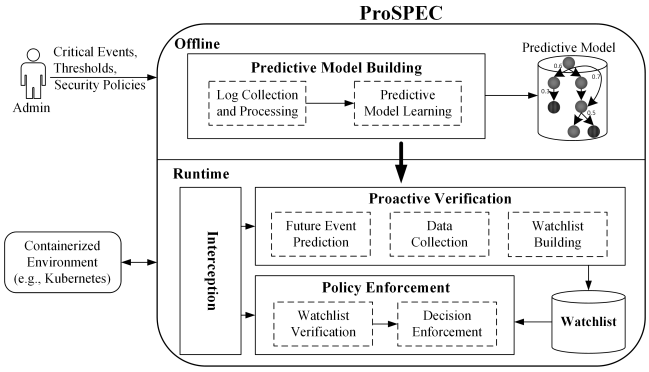
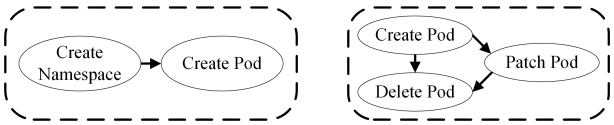


Figure 3: Overview of the ProSPEC approach

Fig. 4b shows an example of intra-resource dependency relationship where, for instance, a `delete` event on a Pod resource can only be performed after that Pod is created. Note that similar management events and their dependencies also exist for other container environments beyond Kubernetes, as discussed in Section 4.2.



(a) Inter-resource **(b) Intra-resource**

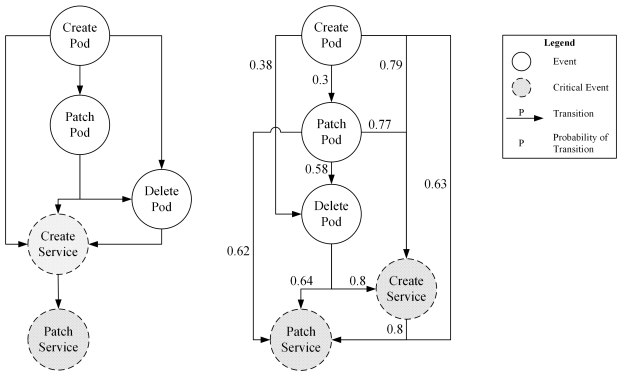
Figure 4: An excerpt of event dependencies in Kubernetes

Building Predictive Models. To learn the aforementioned dependencies and build predictive models, ProSPEC first collects and processes historical container events (e.g., event logs) from the orchestrator (e.g., Kubernetes), and then leverages probabilistic learning methods (e.g., Bayesian network) to build the predictive models. Those steps are detailed in the following and Algorithm 1 summarizes how we build the predictive model.

Algorithm 1 ProSPEC offline phase

```
1:  Input: Raw event logs
2:  Output: Predictive model
3:  procedure BuildModel(RawEventLogs)
4:      for each line ∈ RawEventLogs do
5:          Parse the line;
6:          Extract fields and type the event accordingly;
7:      end for
8:      Build the event sequences;
9:      Build structure from the event sequences;
10:     for each node ∈ structure do
11:         for each other node do
12:             if other node is reachable from node then
13:                 Add the transition to structure;
14:             end if
15:         end for
16:     end for
17:     Run Bayesian Learning on structure with event sequences;
18: end procedure
```

Collecting and Processing Logs. This step is to collect and process logs from a container environment and prepare the inputs for learning predictive models (in the next step). First, ProSPEC collects event logs at the orchestrator level (e.g., from Kubernetes). To that end, based on the available sources of logs in the orchestrator, ProSPEC collects those logs. Second, to enable learning from collected logs, ProSPEC processes the log entries, identifies the event



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

Figure 5: ProSPEC predictive models

types and extracts meaningful sequences of events. To that end, it may need separating and removing system-initiated events from management events, and identifying event types and resources from API calls, as detailed in Section 4.2 for Kubernetes.

Learning Predictive Models. This step is to learn predictive models (including their nodes, edges, and labels of edges) from the sequences of events to enable proactive security policy enforcement during the runtime phase. Each predictive model is built in three steps:

- First, ProSPEC identifies the nodes and edges of the model from the sequences of events. To that end, it extracts the unique event types from the sequences and identifies them as the nodes of the model. Afterwards, it extracts all immediate transitions between event-pairs from sequences and identifies them as edges between those event nodes. For instance, Fig. 5a shows an excerpt of the output with such nodes and edges for Kubernetes.
- Subsequently, to further include the non-immediate transitions (i.e., transitions from one event to another through one or more intermediate transitions), ProSPEC utilizes a Breadth-First Search (BFS) algorithm [27] to determine each node's ability to reach non-adjacent nodes and if so, includes these transitions as additional edges in the model obtained from the previous step. Fig. 5b shows an example of such model with its additional edges.
- Finally, ProSPEC learns the labels of the edges from the sequences of events. To that end, it establishes probabilistic dependencies by leveraging existing Bayesian network learning techniques [36] where the conditional probabilities indicate the likelihood for a (immediate or non-immediate) transition to occur, and those probabilities are used as the labels of the corresponding edges representing the transitions in our model. In the end, ProSPEC builds the predictive model that will be utilized during the runtime phase, as demonstrated in Fig. 5b.

Example 2. Fig. 6 shows an example of applying ProSPEC's offline phase on a subset of logs. In (1) and (2), the log collection and processing module 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 identifies three sequences: "`Create Pod`, `Delete Pod`, `Create Service`", "`Create Pod`, `Create Service`, `Patch Service`" and "`Create Pod`, `Patch Service`". Next, sequences are built in (4), and in (5), ProSPEC identifies the unique four nodes of the model: `Create Pod`, `Delete Pod`, `Create Service`, and `Patch Service`. In (6), it identifies five edges from the immediate transitions: (`Create Pod`, `Delete Pod`), (`Delete Pod`, `Create Service`), (`Create Pod`, `Create Service`), (`Create Service`, `Patch Service`), and (`Create Pod`, `Patch Service`). In (7), using the BFS algorithm, it finds a non-immediate transition: `Delete Pod` to `Patch Service` (through `Create Service`), and adds an additional edge: (`Delete Pod`, `Patch Service`). Finally, in (8), it learns the conditional probabilities for transitions given the event sequences identified in (4).

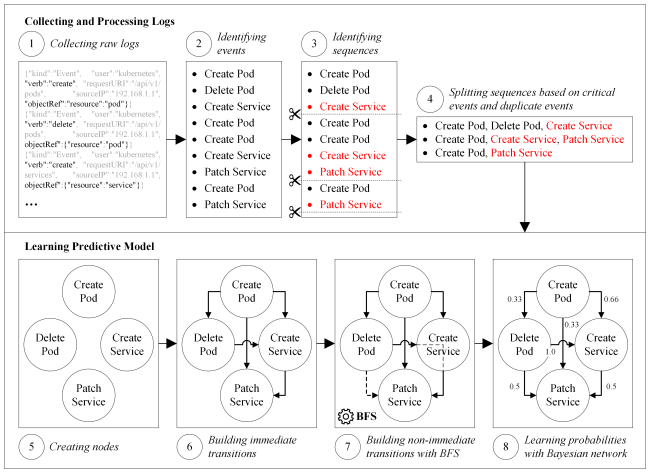


Figure 6: An example of offline learning

3.2 The Runtime Phase

This section describes how ProSPEC intercepts events, conducts proactive verification, and enforces security policies at runtime. Those steps are detailed in the following, and Algorithm 2 summarizes the runtime phase of ProSPEC.

Interception. ProSPEC intercepts runtime events requested by users when such events are sent to a container orchestrator (e.g., Kubernetes), and provides the details of those events to the following runtime steps. To that end, it initially blocks the execution of an event to determine if the requested event is critical (which may potentially breach a security policy). If the event is critical, then ProSPEC keeps the blocking till it completes the *policy enforcement* step. If the event is not critical, then ProSPEC releases the blocking to allow Kubernetes to execute the event, and then ProSPEC conducts the *proactive verification* step for this non-critical event.

Proactive Verification. ProSPEC conducts proactive verification for future events that are predicted based on the intercepted event. Precisely, it first identifies the highly probable (which have a prediction probability higher than a chosen threshold) future critical events from the current event using the predictive model. Second, for such predicted events, it collects the existing resource data related to each security policy from the orchestrator. Finally, it builds a watchlist (e.g., a blacklist of parameters that may lead to a policy breach) by verifying the collected resource data against each policy. As ProSPEC blocks critical events until pre-computation is over (which still causes less delay to users than an *intercept-and-check*

solution, as our experiments show in Section 5), it can eliminate the kind of attack windows demonstrated in Section 2.2.

Policy Enforcement. ProSPEC enforces security policies at runtime based on the watchlists built in the previous step. To that end, if an intercepted event is determined to be critical w.r.t. a security policy, then ProSPEC first checks the requested parameter(s) of that critical event against the watchlist(s) of the policy. Second, based on whether the requested parameters are present in or absent from the watchlist(s), ProSPEC takes the enforcement decision of *allow* or *deny*, according to the watchlist rule (e.g., whitelist or blacklist). Note that in cases where the watchlist is not correctly built for an event (e.g., wrong event prediction, incomplete predictive model, etc.), ProSPEC would simply fall back to the *intercept-and-check* mode (i.e., it will perform verification and enforcement after the event has arrived) whose impact will be evaluated through experiments in Section 5.

Algorithm 2 ProSPEC 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 threshold then
10:             Start pre-computation;
11:         end if
12:     end if
13: end procedure
```

Example 1. Fig. 7 shows an example of our runtime phase. For this example, we consider the same scenario as in Section 2.2, where CVE-2020-8554 [10] can be exploited to perform a man-in-the-middle attack and data theft. To prevent the threat before the vulnerability can be patched, suppose a security policy is specified as: *creating/patching Services should not be allowed to use an externalIP address identical to any existing IPs.* The critical events for this policy are: `Create Service` and `Patch Service`. The probabilistic predictive model is built offline and is the same as shown in Fig. 5b.

At runtime, for the first intercepted event `Create Pod` with its *IP* address, `192.168.1.1`, ProSPEC predicts the next critical event, `Create Service`, using the predictive model, and adds the *IP* address `192.168.1.1` to the watchlist (blacklist) as it is now used for the Pod. For the second intercepted event, `Create Service` with an *externalIP*, `192.168.1.1`, ProSPEC denies the request as this requested *IP* is in the watchlist (a policy breach). Similarly, the third event will be allowed as its Service *externalIP*, `192.168.0.8`, is not in the watchlist, whereas the fourth event will be denied as it modifies the *externalIP* to `192.168.1.1`, *IP* that is in the watchlist. Note that ProSPEC avoids inconsistencies between the watchlist and the actual state of the cluster (as shown in Section 2.2) since the second request is blocked until the pre-computation is done.

4 IMPLEMENTATION AND INTEGRATION

This section first details the implementation of ProSPEC, then describes its integration with Kubernetes, and finally discusses the challenges tackled during these steps.

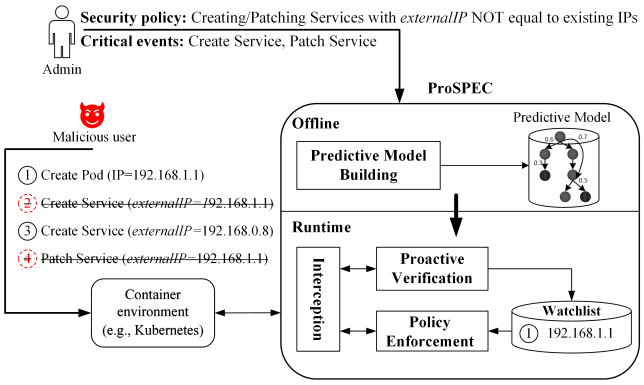


Figure 7: ProSPEC preventing CVE-2020-8554

4.1 ProSPEC Implementation

Fig. 8 shows the high-level architecture of ProSPEC illustrating how it manages its inputs, and the two major modules: *offline learning* and *runtime enforcement*. We elaborate on those in the following.

Management of the ProSPEC Inputs. ProSPEC takes several inputs (e.g., configurations, logs, policies, critical events, threshold values, and watchlists definitions) from a container environment as well as its administrators. To manage those inputs, ProSPEC maintains a database using *SQLite* [30] for its portability and simplicity with four different tables, `PolicySettings`, `PolicyThreshold`, `PolicyWatchlist`, and `Model` as follows.

- 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 proactive attribute for enabling or disabling the proactive feature for that policy.
- The `PolicyThreshold` table stores the critical events and their threshold 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 threshold value for that critical event.
- The `PolicyWatchlist` table stores the actual watchlists content pre-computed by ProSPEC 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 predictive models for each policy, 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, as well as the probability of that transition.

Implementation of the Offline Learning Module. The three main components of this module include `log collector`, `log processor`, and `predictive model learner`, as detailed below.

- The `log collector` and `log processor` components are responsible for collecting event logs from a container environment and preparing them for the learning tool. To that purpose, ProSPEC first enables the Kubernetes audit logs feature (see Section 4.2). Afterwards, to process the audit log file and extract only the required information from the raw JSON audit logs, it

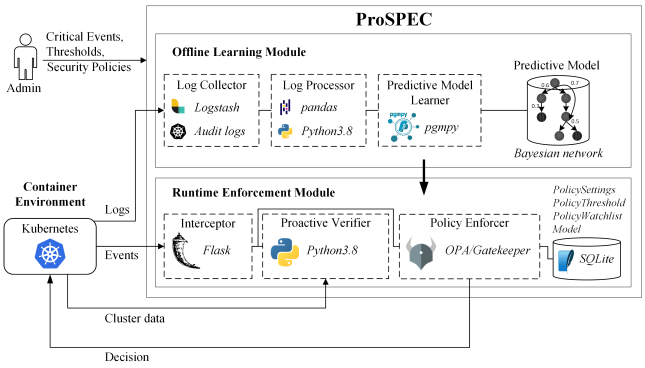


Figure 8: The architecture of our ProSPEC implementation

leverages Logstash [21], a popular log processor. Moreover, it extracts the fields `receivedRequestTimestamp`, `user[username]`, `objectRef[resource]` and `verb` from the logs and stores them in a CSV file, where each line represents a log entry. Furthermore, using the Python data analysis toolkit pandas v1.2.4 [40] and our own code, it processes each of those lines with event typing that maps the pair (verb, resource) to a string `verb_resource` (event type). Finally, ProSPEC splits events into sequences in a way that it avoids any cycle (or repeated events in a sequence) and ends with a critical event (if any in that sequence), as the predictive model is a directed acyclic graph. More precisely, it ensures each sequence always begins with a non-critical event and cuts the current sequence after it sees one or more critical events in a row, or a repetition of the existing events.

- The `predictive model learner` component is to learn the predictive model, which is represented as a Bayesian network. The event sequences are used as described in Section 3.1. ProSPEC follows a standard iterative implementation [27] of the BFS algorithm using a queue to check the reachability of nodes. It also leverages the `BayesianModel` and `MaximumLikelihood` classes of the Python library for learning and inference in Bayesian networks, pgmpy v0.1.14 [24], to learn the probabilities. The obtained model is stored in the `Model` table in our database.

Implementation of the Runtime Phase. The three main components of this module include `interceptor`, `proactive verifier`, and `policy enforcer`, as detailed below.

- The `interceptor` component aims at intercepting runtime event requests to a container orchestrator (e.g., Kubernetes). To that end, ProSPEC leverages the Kubernetes admission controller mechanism to intercept the requests sent to 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 4.2). The `interceptor` component runs as a local web server using the micro web framework Flask [29], and is registered as an admission controller in Kubernetes. The so-built webhook receives requests from the Kubernetes API server, processes them to extract useful data and places the events in a FIFO queue.
- The `proactive verifier` component is to incrementally build the watchlist for a security policy. Particularly, it takes the first intercepted event in the FIFO queue and queries in the ProSPEC database as follows:
  `SELECT Policy FROM PolicyThreshold INNER JOIN Model ON Model.FutureEvent = PolicyThreshold.CriticalEvent WHERE ((Model.CurrentEvent = CurrentEvent) AND (Model.Probability >= PolicyThreshold.Threshold))`.
  For the policies that are selected in this way, this component collects the needed data to build or update the watchlist. For each event that requires pre-computation, the component receives the corresponding policy(ies) and starts collecting the required data defined with the policies using HTTP(S) requests to the cluster. For instance, to gather the *IP* addresses of Pods required in Section 3.2, the `proactive verifier` component would query the API server with the following URI: https://localhost:6443/api/v1/pods (following the Kubernetes API reference [19]). Collected data in the JSON format is further processed to extract the interesting features (e.g., Pods *IP* addresses). Then the `proactive verifier` component writes the collected features to the *PolicyWatchlist* table.
- The `policy enforcer` component is for watchlist verification and decision enforcement; which integrates OPA/Gatekeeper [5] and will be detailed in the next section. Note that it is always possible to implement ProSPEC independently from OPA/Gatekeeper, as a registered admission controller that verifies the watchlists and enforces the policies. However, integrating ProSPEC with OPA/Gatekeeper presents several advantages, including preserving the features offered by OPA/Gatekeeper while bringing advantages of a proactive solution to existing policies.

4.2 ProSPEC Integration with Kubernetes

We first present background information about Kubernetes and then detail the integration of ProSPEC with Kubernetes.

Kubernetes Background. In the following, we provide a background on Kubernetes (including its basics, its admission controller mechanism and event logs), which will later be necessary in discussing ProSPEC integration.

- *Kubernetes Basics.* Kubernetes [18] is a container orchestrator that runs, manages, and coordinates the deployments of containerized applications. In Kubernetes, a cluster contains a master Node responsible for controlling and managing a set of worker Nodes containing multiple Pods that run the applications. Any operation on the cluster that queries or modifies the state of Kubernetes resources (e.g., Pods, Services, etc.) is first received by the Kube-API server, which applies them by communicating with the worker Nodes. In the following, we describe the admission controller mechanism and the event logs in Kubernetes, which will later be utilized in the integration of ProSPEC.
- *Admission Controller.* An admission controller in Kubernetes aims at intercepting the requests to the Kube-API server and performing validation, mutation, or both in order to protect clusters against malicious user activities. Particularly, OPA/Gatekeeper [5] 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 to the Kube-API server with respect to pre-defined policies. When a request is made to

the Kube-API server, Gatekeeper uses OPA as a library to verify the intercepted request against a set of pre-defined policies. Based on the response from OPA, Gatekeeper enforces the decision (i.e., allows or denies the request).

- *Event Logs.* There are three different sources for capturing Kubernetes event logs: (i) management operation (e.g., Kubernetes command-line interface (CLI) history), (ii) event object (e.g., `kubectl get events` command) with the life span of one hour, and (iii) audit logs containing detailed events and attributes (e.g., resource-name, resource-type, operation, etc.). There are 71 types of resources in Kubernetes v1.20.2, and for each of them there are up to eight possible operations. Table 1 shows an example of Kubernetes events for three sample resources.

Table 1: An excerpt of Kubernetes events

Kubernetes resources	Operations
Namespace	create, delete, get, list, patch, update, watch
Pod	create, delete, delete collection, get, list, patch, update, watch
Service	create, delete, delete collection, get, list, patch, update, watch

Integration with Kubernetes. Fig. 9 illustrates the integration of ProSPEC with Kubernetes. Particularly, Fig. 9a provides a high-level overview of the integration including the deployment of a Kubernetes testbed, and Fig. 9b highlights the key integration aspects including how particularly ProSPEC is integrated with the Kube-API server and OPA/Gatekeeper, as detailed below.

- First, Fig. 9a shows the deployment of the Kubernetes testbed with ProSPEC. The physical hardware of our cloud 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. On top of that, VirtualBox6.1 is running as a Type-2 hypervisor. The container environment is deployed over 11 VMs 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 Lubuntu 20.04 and we use Python 3.8 for all programming tasks. Additionally, we use Kubernetes v1.20.2 through the `kubectl` CLI and the `kubeadm` tool for creating the cluster. The standard and recommended installation steps [17] are followed to deploy the cluster, including disabling swap partitions. The container environment is Docker v20.10.
- Second, Fig. 9b shows the integration of ProSPEC with OPA/Gatekeeper on the Kubernetes deployment. ProSPEC collects necessary data as well as intercepts current runtime events with the help of the Kube-API server. Afterwards, based on the watchlist contents, ProSPEC creates a constraint parameter for OPA/Gatekeeper. Additionally, the policy template is defined and applied in advance. The proper enforcement is performed through ProSPEC by applying 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 constraints. (ii) Widely-used OPA/Gatekeeper's features are preserved while bringing ProSPEC's proactive advantages. (iii) ProSPEC remains as much decoupled as possible from Kubernetes.

Adapting with Other Container Orchestrators. Although our implementation is to fit with Kubernetes, ProSPEC can be adapted to other container orchestrators (e.g., Docker Swarm [15], OpenShift [22]). The container-specific concepts on which ProSPEC relies on are not too specific to Kubernetes and are also implemented in Docker Swarm and OpenShift. Table 2 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 [14]. The usage of API in all these orchestrators greatly facilitates the access to in-cluster resources. Therefore, the adaption to those orchestrators can be possible with a minimal effort (that will be explored in future work).

Table 2: An excerpt of equivalent terminologies and concepts among three main container orchestrators

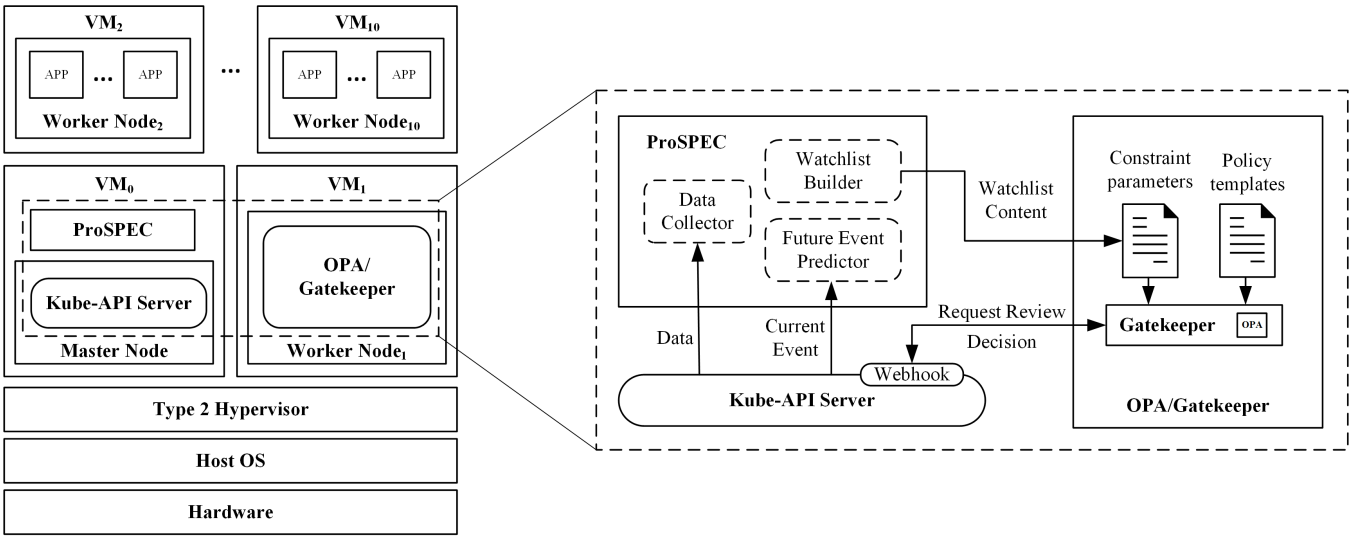
Kubernetes [18]	Docker Swarm [15]	OpenShift [22]
Cluster	Swarm	Cluster
Pod	Task	Pod
Event	(Docker) Event	(OpenShift) Event
Namespace	Stack	Project
Admission Control	Third-party Plug-in	Admission Plug-in

4.3 Challenges

We describe various challenges that were encountered and addressed during our implementation and integration of ProSPEC as follows.

Enabling Kubernetes Audit Logs for Model Learning. After exploring all available options of event logs in Kubernetes (as discussed in Section 4.2), we choose Kubernetes audit logs to train our model and learn dependencies among events. Those audit logs represent the best source of information for monitoring the events in the cluster since they provide enough granularity and details for us to obtain the information needed for some policies (e.g., the relationship between a Pod and the Service exposing it). However, working with Kubernetes audit logs requires some efforts as follows. First, the audit log option is disabled by default in Kubernetes, and enabling it requires setting the `-audit-log-path` flag in the `kube-apiserver.yaml` file. Second, a directory with sufficient write permissions must also be specified. Third, as audit logs are verbose by default, to limit the logging to specific resources (e.g., Pods, Services) and verbs (e.g., Create, Delete, Patch, Update), we need to enable audit log filtering by specifying an audit policy file. Finally, the cluster must be restarted after enabling the audit logs and then Kubernetes will start to append all the received requests to a log file in JSON format in the specified folder. More details on Kubernetes audit logs can be found in [20].

Accessing Kubernetes API. The Kubernetes API can only be accessed from inside the cluster network, or by the `kubectl` CLI. However, as OPA/Gatekeeper is running inside a container and ProSPEC is running outside the cluster, both have no direct access to the Kubernetes API and must be given another way to reach it in order to read the cluster state for policy verification. To overcome that issue, we modify the OPA/Gatekeeper container image



(a) High-level overview of the integration

(b) Detailed view of the integration

Figure 9: Showing both (a) high-level overview and (b) detailed view of the integration of ProSPEC with Kubernetes

and deploy a sidecar container running a Kubernetes API proxy, `kube-proxy`. Similarly, we run `kube-proxy` in the master VM to give ProSPEC an access to the Kubernetes API.

Intercepting Events at Runtime. As ProSPEC aims at reducing the policy verification and enforcement time, we need to find a solution to minimize the delay between the time when user requests made to the cluster reach the Kubernetes API and the time when those requests can be intercepted. To that end, we register ProSPEC as a Kubernetes admission controller such that it can intercept the requests as early as other admission controllers such as OPA/Gatekeeper. As Kubernetes admission controllers must use the TLS protocol in their communication, we sign ProSPEC certificate using the Kubernetes root certificate. More details on admission controller registration in Kubernetes are found in [16].

Feeding Watchlist Contents to OPA/Gatekeeper Constraints. We use OPA/Gatekeeper for watchlist verification and policy enforcement in our implementation (as discussed in Section 4.2). 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 ProSPEC watchlist content in the YAML format of a standard constraint file of OPA/Gatekeeper. We can then feed such encoded watchlists to OPA/Gatekeeper through a Custom Resource Definition (CRD) predefined by OPA/Gatekeeper (e.g., command `kubectl apply -f constraint.yaml`).

Learning Model Structure. To learn the structure of our predictive model, we first investigated regular Directed Acyclic Graph (DAG) structure learning approaches such as MMHC [39] or Constraint-Based estimation [36]. However, they could not serve our purpose, as they are not able to capture the chronological order between events in sequences and subsequently led to wrong edge direction problem. To overcome this challenge, ProSPEC performs structure learning by first deriving the direct (immediate) dependencies between two events from the audit logs per sequence, then applying a Breadth-first search (BFS) algorithm to derive the conditional edge between nodes, and finally using the Maximum Likelihood Estimation (MLE) for parameter learning with the conditional predictive model (as shown in Fig. 5b).

Measuring Response Time. In our experiments, the response time measurement is performed at the admission controller level (i.e., OPA/Gatekeeper) to avoid external biases, as discussed in Section 5. However, as OPA/Gatekeeper runs in a container, it is impractical to access the process and attach a debugger from outside the cluster. To overcome this challenge and ensure the accuracy of our measurement, we modify the OPA/Gatekeeper source code (note such modification is only needed for our experiments and not required for deploying ProSPEC) to include a metrics logging feature and we rebuild the container image. This way, the response time is available in the easily accessible container logs.

Minimizing Other Networking Effects in Efficiency Measurement. As one of the main objectives of our experiments is to measure the response time, we want to avoid any perturbations that would affect these measures, such as network congestion. To that end, the physical network between Nodes is simulated through VirtualBox internal network interfaces. In Kubernetes, the network model is managed through a Container Network Interface (CNI) plugin. We select Calico [7] for our Kubernetes cluster as it is referred as one of the best network overlays in terms of performances [4]. The recommended deployment for the cluster with 50 Nodes or less is applied as specified in the documentation except that we set the `IP_AUTODETECTION_METHOD` parameter to match the internal network interface, to avoid BGP failure.

Avoiding Inconsistencies among Cgroups. Control groups (`cgroups`) is a Linux Kernel feature that allows control and isolation of hardware resources used by processes, typically used for containers. However, inconsistencies might arise between Docker containers `cgroups` and Kubernetes `cgroups` [13]. As a solution, we

set up the Docker `cgroups-driver` to `systemd` as recommended in Kubernetes documentation.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Settings

Environment. The experimental environment is set up on our Kubernetes testbed described in Section 4.2. For all the experiments, ProSPEC is running on the master VM and OPA/Gatekeeper inside a container on a worker Node. To simulate real-world environments [9], the size of the Kubernetes cluster is varied for different experiments with up to 800 Pods and 800 Ingress rules, and the size of the input requests varies between a single critical resource and a set of 100 critical resources in a batch (here resources mean Kubernetes Services for our sample policies, as discussed later).

Security Policies. For our experiments, we use two sample security policies based on real-world use-cases [10, 11]: *Policy 1* (presented in our motivating example in Section 2.2) is inspired by a real-world vulnerability CVE-2020-8554 [10], and *Policy 2* is designed to prevent common misconfigurations in Kubernetes [11].

- **Policy 1.** This policy prevents a malicious user from intercepting traffic to other resources by creating Services with an *externalIP* identical to the *IP* address of an existing Pod in the cluster. To enforce this policy, ProSPEC dynamically maintains a blacklist containing the *IPs* of all existing Pods in the cluster, and prevents Services from exposing such existing *IPs*.
- **Policy 2.** This policy prevents a common Kubernetes misconfiguration called Ingress rules conflicts [11] in which deploying multiple Ingress rules to manage external access to Services can potentially lead to service failure and/or data exposure. To enforce this policy, ProSPEC dynamically maintains a blacklist containing all Ingresses hostname rules in the cluster to prevent any Ingress rules conflict from happening.

5.2 Experimental Results

In the following, we evaluate the performance of ProSPEC in terms of response time, impact of wrong predictions, impact of threshold, offline learning time and rate of correct predictions.

Impact of Cluster Size on Response Time. The first set of experiments shown in Fig. 10 measures the response time of ProSPEC. The response time is measured as the duration between the time ProSPEC receives a critical request and the time ProSPEC returns an enforcement decision to Kubernetes. As specified in Section 4.3, the response time is measured directly at the decision engine level (i.e., OPA/Gatekeeper) to avoid any overhead due to external factors.

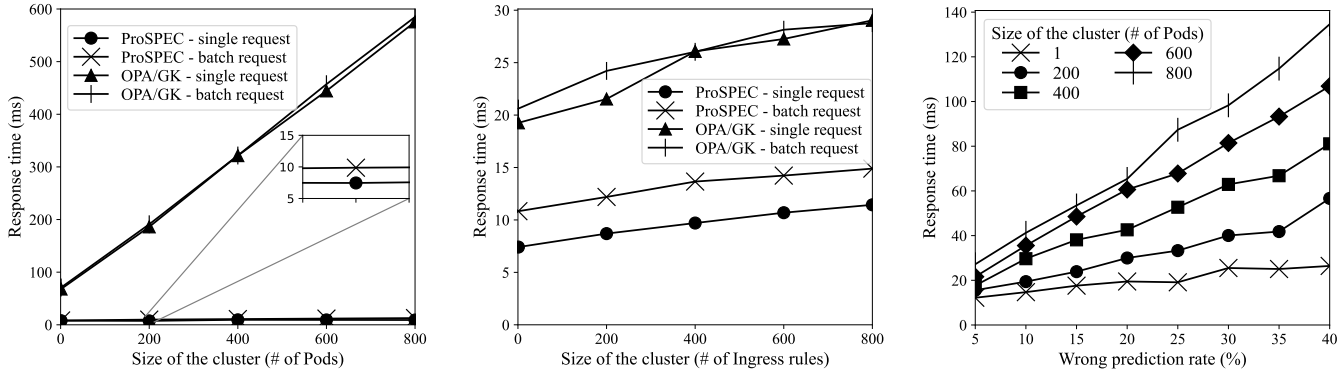
Particularly, Fig. 10a shows the comparison of the response time between ProSPEC and OPA/Gatekeeper to enforce *Policy 1* when we vary the size of the cluster (# of Pods) for both a single request of one resource and a batch request of 100 resources. As shown in the figure, for ProSPEC, we observe a near-constant response time (lower than 15 ms). On the other hand, it can be seen that the response time for OPA/Gatekeeper grows almost linearly in the size of the cluster. This is mostly due to the reactive nature of OPA/Gatekeeper, i.e., it performs the time-consuming operation of gathering the *IP* addresses of all the existing Pods at runtime. For the largest size of cluster and for one resource, OPA/Gatekeeper takes up to 580 ms, whereas ProSPEC takes only 15 ms (which is close to 40 times faster). The zoomed inset shows the ProSPEC response times on a more precise scale for a single request and a batch request, measured at 7 ms and 10 ms, respectively.

Fig. 10b shows the comparison of the response time between ProSPEC and OPA/Gatekeeper to enforce *Policy 2* 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 ProSPEC and OPA/Gatekeeper grow almost linearly, ProSPEC still outperforms OPA/Gatekeeper in all cases (e.g., for the largest cluster, 15 ms by ProSPEC vs. 29 ms by OPA/Gatekeeper). Additionally, as discussed in Section 2.2, 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, ProSPEC 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 1* is relatively longer than that for *Policy 2*, 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 1*, the additional delay induced by the batch request is negligible with respect to the response time. In the case of *Policy 2*, the additional 4 ms delay to the response time 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 ProSPEC is similar, and ProSPEC 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 ProSPEC. For this purpose, we consider the case where a critical event occurs without being predicted by ProSPEC, which has an impact on the response time as ProSPEC 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 ProSPEC level and the verification time measured at OPA/Gatekeeper level). For this experiment, we vary the rates of wrong predictions in the model and use *Policy 1* 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. 10c shows the average overall response time (incurred in pre-computation as well as in verification by ProSPEC for enforcing *Policy 1*) in case of different (simulated) wrong predictions rates. As a baseline, in Fig. 10a (without simulated errors), the response time is around 12 ms for 800 Pods. In contrast, Fig. 10c shows that, even with a 40% error rate, the response time of ProSPEC still stays below 140 ms for 800 Pods, which is better than the performance of OPA/Gatekeeper in the same environment (580 ms, see Fig. 10a). As the error rate is likely much lower in reality (see Fig. 12b), we can conclude that wrong predictions will not significantly affect the effectiveness of ProSPEC.

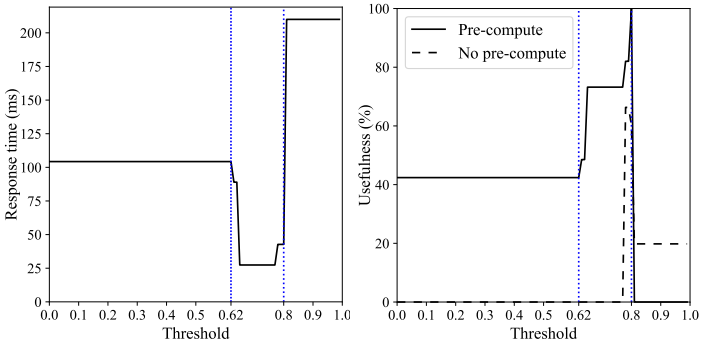


(a) Impact of # of Pods on *Policy 1*

(b) Impact of # of Ingress rules on *Policy 2*

(c) Impact of wrong predictions on *Policy 1*

Figure 10: Impact of the size of cluster and wrong predictions rate on the response time



(a) Impact of threshold on response time

(b) Impact of threshold on pre-computation efficiency

Figure 11: Impact of threshold (dashed vertical lines show the minimum (0.62) and maximum (0.8) transition probabilities to a critical event in the predictive model) with 200 Pods and enforcing *Policy 1*

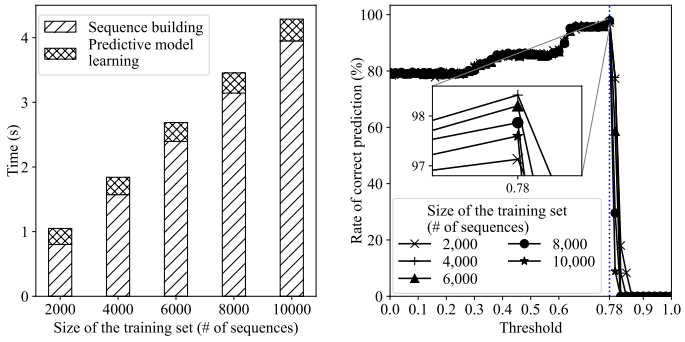
Impact of Threshold on Response Time. The third set of experiments is to measure the impact of different threshold values (as described in Section 3) on the response time as well as on the pre-computation efficiency of ProSPEC. For this experiment, we vary the value of the critical events threshold from 0 to 1 to measure its impact on enforcing *Policy 1* for 200 Pods and single requests (i.e., one resource per request). 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) to 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 ProSPEC security; whereas this experiment measures the usefulness of its pre-computation step.

Fig. 11a and Fig. 11b show the response time of ProSPEC and the aforementioned usefulness metrics as functions of the threshold.

Fig. 11a shows the average response time stays almost constant for threshold values below 0.62 or above 0.8, respectively (0.62 and 0.8 are the lowest and highest transition probabilities to a critical event existing in the used model, as in Fig. 5b). For threshold values above 0.8, the average response time is the highest at more than 200 ms, since we never pre-compute and have to perform the verification at runtime under such threshold values. For threshold values below 0.62, the response time peaks at more than 100 ms, since we always pre-compute but often unnecessarily (for non-critical events). Between threshold values of 0.62 and 0.8, we observe the lowest response time. Precisely, a threshold value between 0.65 and 0.78 reduces the average response to a minimum of 27 ms.

Fig. 11b shows the pre-computation efficiency under different threshold values. For the threshold values below 0.62, the usefulness of pre-computation decreases to around 40%, because that range of threshold values will always trigger pre-computation for every event (the lowest transition probability in the used model is 0.62, as shown in Fig. 5b). On the other hand, the *no pre-compute usefulness* stays at 0% as we always pre-compute. Once the threshold values pass 0.62, we can observe a sharp increase in both the *pre-compute usefulness* and *no pre-compute usefulness*. The *pre-compute usefulness* increases as pre-computation is mostly for the probable next events under such threshold values. The *no pre-compute usefulness* increases more sharply as we do not pre-compute for events with low transition probabilities under such threshold values. For the specific model and policy used in this experiment, a threshold value of 0.8 allows us to reach 100% usefulness, meaning we neither waste nor miss any pre-computation operation. Above the threshold value of 0.8, the *pre-compute usefulness* drops to 0% as in the given model there is no transition with higher probabilities and hence no pre-computation is triggered. On the other hand, the *no pre-compute usefulness* drops to around 20% as we miss necessary pre-computations under such threshold values. Finally, we can identify an optimal threshold value of 0.78 under which Fig. 11a shows the lowest response time, and Fig. 11b shows the maximum overall efficiency (in terms of both the *pre-compute usefulness* and *no pre-compute usefulness*). Thus, an optimal threshold value can be determined based on the given policy and training data.

Offline Learning Time. The fourth set of experiments is to measure the offline learning time, i.e., the time required for deriving the predictive model from event logs. The measured time in this experiment includes the time to build the event sequences as well as



(a) Measured time for different learning steps: sequence building and model learning

(b) The correct predictions rates of our model for different thresholds and # of sequences

Figure 12: Learning time and rate of correct predictions of our predictive model (dashed vertical line shows peak rate)

the time to learn the predictive model using the Bayesian network library pgmpy [24]. The log processing task, performed by Logstash, is not considered in this experiment as in practice it is supposed to be performed in parallel, while the audit logs are collected from the container environment.

Fig. 12a shows the time required by the *predictive model building* module of ProSPEC to sequence the logs and build 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 increasing from 801 ms to 3,950 ms under similar numbers of sequences. This has a practical implication since the more expensive sequence building only needs to be performed once for each event sequence, 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. The following experiment shows the impact of the training data size on the rate of correct predictions.

Rate of Correct Predictions at Runtime. The fifth set of experiments is to assess the relation between the rate of correct predictions, threshold values and size of the training set. During the ProSPEC runtime phase, the chosen threshold for the critical events dictates if a pre-computation will be triggered or not. As the model accuracy will rely on whether we correctly predict critical events or not, it is thus important to show that the chosen threshold has an impact on the overall rate of correct predictions of the model. Note that the best accuracy and corresponding threshold may vary based on different predictive models. The rate of correct predictions is measured for different datasets by varying the number of event sequence from 2,000 to 10,000. 80% of each dataset is used during the training and the remaining is used for testing. For each threshold value, we define the rate of correct predictions as the number of successful predictions over the number of total predictions.

Fig. 12b shows the rate of correct predictions as a function of threshold values for different datasets. We find that the best rate for the used model (as in Fig. 5b) reaches 98.4% for a threshold value of 0.78 and a training dataset of 4,000 sequences. However, small differences between different training sets are observed; specifically, it shows that a training set larger than 2,000 does not significantly improve the rate of correct predictions. We can establish a correlation between this result and the response time or pre-computation efficiency in Figs. 11a, and 11b. Indeed, for the threshold values between 0.62 and 0.8, we not only observe better pre-computation efficiencies and the lowest average response time, but also an increase in the rate of correct predictions.

6 RELATED WORK

In this section, we review relevant works in container environment security and security policy compliance.

Container Security Verification. There are several works (e.g., [23, 28, 32]) on container security, particularly aiming at verifying the security of container images (e.g., [23]) and checking their integrity (e.g., [28, 32]). However, those works focus on a single security aspect such as developing vulnerability-free container images or integrity attestation, and do not propose a solution for the verification and enforcement of security polices at runtime. For instance, [23] is a vulnerability-centric approach to identify and assess vulnerabilities in Docker containers images. Both [28, 32] propose solutions for containers integrity attestation covering the entire life cycle of the containers and their underlying images.

Kubernetes Security. There exist solutions addressing different security aspects in Kubernetes. According to [37], the security best practices for Kubernetes are as follows: (i) API-based authentication and authorization request through authentication plugin and policies., (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, to keep it updated with latest security fixes, (iv) logging/monitoring the cluster, and (v) continuous security compliance. ProSPEC subscribes to the latter by proposing a proactive and efficient security compliance solution for container environments. In contrast, most of the existing works (e.g., [2, 3, 38]) propose reactive solutions that can only detect security policy violations after they occur, which may expose the system to large attack windows and thus higher security risks. For example, *Sysdig* [3] provides a system-call level security attack detection approach while *Falco* [2] offers an online anomaly detection tool for containerized applications. *KubAnomaly* [38] is a learning-based anomaly detection system, providing runtime monitoring capabilities in Kubernetes. Also, OPA is a security policy engine and, Gatekeeper as its sidecar, is an enforcement tool designed for Kubernetes [5]. ProSPEC differs from those works as it proactively prevents policy violations.

Security Policy Compliance. There are several proactive security compliance verification works (e.g., [1, 26, 33, 34]) for non-container environments (e.g., OpenStack [1] clouds). For instance, Weatherman [26] and Congress [1] verify security policies in clouds

using graph-based and Datalog-based models, respectively. Moreover in [41], a proactive protection approach for potential security breaches in cloud is proposed. Unlike our automated learning of predictive model, those works rely on manual inputs of future plans. *LeaPS* [34] and *Proactivizer* [35] are proactive security auditing solutions for cloud environments. In contrast to our work, those works are not specifically designed to tackle complexity and challenges of container environments such as supporting container-specific events, capturing dependencies among diverse types of resources, and deriving a predictive model from those dependencies.

In summary, ProSPEC mainly differs from the state-of-the-art works as follows. First, ProSPEC provides a proactive security policy enforcement solution designed for container environments that prevents security compliance breaches. Second, ProSPEC 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, Kubernetes, while offering the benefit of a proactive solution.

7 CONCLUSION

In this paper, we proposed ProSPEC, a proactive security policy enforcement solution for container environments. We leveraged learning techniques to derive a predictive model that captures dependencies among events in container environments. ProSPEC 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 ProSPEC and integrated it with Kubernetes, a popular container orchestrator.

Limitations and Future Work. First, ProSPEC does not retrain, tune the model nor adjust the thresholds based on historical compliance and changes in user behavior. Our future work will support dynamic online learning of the predictive model. Second, in ProSPEC, identification of critical events, security policies, as well as event typing are still manually performed. As future work, this process can be automated by leveraging supervised machine learning approaches. Third, currently ProSPEC is integrated with Kubernetes. In the future, we will integrate it with other container orchestrators, such as Docker Swarm [15] and OpenShift [22]. Finally, ProSPEC is currently at the deployment phase, that can be extended as a proactive security solution for containers after their deployment.

REFERENCES

[1] 2015. *OpenStack Congress.* Retrieved July 09, 2021 from https://wiki.openstack.org/wiki/Congress/
[2] 2018. *Falco.* Retrieved June 15, 2021 from https://falco.org/
[3] 2018. *Sysdig.* Retrieved June 15, 2021 from https://sysdig.com/
[4] 2019. *Benchmark results of Kubernetes CNI over 10Gbit/s network.* Retrieved July, 2021 from https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-august-2020-6e1b757b9e49
[5] 2019. *Open Policy Agent/Gatekeeper.* Retrieved July, 2021 from https://open-policy-agent.github.io/gatekeeper/
[6] 2019. *Report on the Enhancements of the NFV architecture towards"Cloud-native" and "PaaS", ETSI GR NFV-IFA 029.* Technical Report. ETSI.
[7] 2020. *Calico: Open source networking solution for Kubernetes.* Retrieved August 09, 2021 from https://docs.projectcalico.org/
[8] 2020. *Cloud Native Computing Foundation 2020 Annual Report.* Retrieved September, 2021 from www.cncf.io/cncf-annual-report-2020/
[9] 2020. *Cloud Native Computing Foundation 2020 Survey Report.* Retrieved September, 2021 from www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf
[10] 2020. *CVE-2020-8554: Man in the middle in Kubernetes.* Retrieved July 10, 2021 from https://blog.champtar.fr/K8S_MITM_LoadBalancer_ExternalIPs/
[11] 2020. *Torin Sandall, OPA: Top 5 Kubernetes Admission Control Policies.* Retrieved July 20, 2021 from https://thenewstack.io/open-policy-agent-the-top-5-kubernetes-admission-control-policies/
[12] 2021. *'Azurescape' Kubernetes Attack Allows Cross-Container Cloud Compromise.* Retrieved October, 2021 from https://threatpost.com/azurescape-kubernetes-attack-container-cloud-compromise/169319/
[13] 2021. *Container Runtimes.* Retrieved June, 2021 from https://kubernetes.io/docs/setup/production-environment/container-runtimes/#cgroup-drivers
[14] 2021. *Docker authorization with OPA.* Retrieved August 19, 2021 from www.openpolicyagent.org/docs/latest/docker-authorization/
[15] 2021. *Docker Swarm.* Retrieved September 15, 2021 from https://docs.docker.com/engine/swarm/
[16] 2021. *Dynamic Admission Control.* Retrieved September 30, 2021 from https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/
[17] 2021. *Installing kubeadm.* Retrieved June, 2021 from https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/
[18] 2021. *Kubernetes.* Retrieved September 15, 2021 from https://kubernetes.io
[19] 2021. *Kubernetes API Reference.* Retrieved September 20, 2021 from https://v1-18.docs.kubernetes.io/docs/reference/
[20] 2021. *Kubernetes Audit Logs.* Retrieved September 09, 2021 from https://kubernetes.io/docs/tasks/debug-application-cluster/audit/
[21] 2021. *Logstash CVE-2020-8554.* Retrieved June 13, 2021 from www.elastic.co/logstash/
[22] 2021. *OpenShift.* Retrieved September 15, 2021 from https://docs.openshift.com/
[23] Waheeda Syed Shameem Ahamed, Pavol Zavarsky, and Bobby Swar. 2021. Security Audit of Docker Container Images in Cloud Architecture. In *ICSCCC*. IEEE.
[24] Ankur Ankan and Abinash Panda. 2015. pgmpy: Probabilistic graphical models using python. In *SCIPY*. Citeseer.
[25] Mihir Bellare and Bennet Yee. 1997. *Forward integrity for secure audit logs.* Technical Report. Citeseer.
[26] Sören Bleikertz, Carsten Vogel, Thomas Groß, and Sebastian Mödersheim. 2015. Proactive security analysis of changes in virtualized infrastructures. In *ACSAC*.
[27] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms.* MIT press. 594–602 pages.
[28] Marco De Benedictis and Antonio Lioy. 2019. Integrity verification of Docker containers for a lightweight cloud environment. *Future Generation Computer Systems* 97 (2019), 236–246.
[29] Miguel Grinberg. 2018. *Flask web development: developing web applications with python.* " O'Reilly Media, Inc.".
[30] Richard D Hipp. 2020. SQLite. https://www.sqlite.org/index.html
[31] Min Li, Wanyu Zang, Kun Bai, Meng Yu, and Peng Liu. 2013. MyCloud: supporting user-configured privacy protection in cloud computing. In *ACSAC*.
[32] Wu Luo, Qingni Shen, Yutang Xia, and Zhonghai Wu. 2019. Container-IMA: a privacy-preserving integrity measurement architecture for containers. In *RAID*.
[33] Suryadipta Majumdar, Yosr Jarraya, Taous Madi, Amir Alimohammadifar, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. 2016. Proactive verification of security compliance for clouds through pre-computation: Application to OpenStack. In *ESORICS*. Springer.
[34] Suryadipta Majumdar, Yosr Jarraya, Momen Oqaily, Amir Alimohammadifar, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. 2017. LeaPS: Learning-based proactive security auditing for clouds. In *ESORICS*. Springer.
[35] Suryadipta Majumdar, Azadeh Tabiban, Meisam Mohammady, Alaa Oqaily, Yosr Jarraya, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. 2019. Proactivizer: Transforming existing verification tools into efficient solutions for runtime security enforcement. In *ESORICS*. Springer.
[36] Richard E Neapolitan et al. 2004. *Learning bayesian networks.* Vol. 38. Pearson Prentice Hall Upper Saddle River, NJ. 550 pages.
[37] Md Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, and Akond Rahman. 2020. XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices. In *SecDev*. IEEE.
[38] Chin-Wei Tien, Tse-Yung Huang, Chia-Wei Tien, Ting-Chun Huang, and Sy-Yen Kuo. 2019. KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches. *Engineering Reports* 1, 5 (2019), e12080.
[39] Ioannis Tsamardinos, Laura E Brown, and Constantin F Aliferis. 2006. The max-min hill-climbing Bayesian network structure learning algorithm. *Machine learning* 65, 1 (2006), 31–78.
[40] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *SCIPY*, Stéfan van der Walt and Jarrod Millman (Eds.).
[41] Stephen S Yau, Arun Balaji Buduru, and Vinjith Nagaraja. 2015. Protecting critical cloud infrastructures with predictive capability. In *CLOUD*. IEEE.