Proactive Security Policy Enforcement for Containers

Hugo Kermabon-Bobinnec

A Thesis
in
The Concordia Institute
for
Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Master of Applied Science (Information Systems Security) at
Concordia University
Montréal, Québec, Canada

December 2022

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Hugo Kermabon-Bobinnec**
Entitled: **Proactive Security Policy Enforcement for Containers**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair and Examiner
Dr. Mohammad Mannan

_____ Examiner
Dr. Arash Mohammadi

_____ Thesis Supervisor
Dr. Lingyu Wang

_____ Co-supervisor
Dr. Suryadipta Majumdar

Approved by _____
Dr. Zachary Patterson, Graduate Program Director

_____ _____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Proactive Security Policy Enforcement for Containers

Hugo Kermabon-Bobinnec

By providing lightweight and portable support for cloud native applications, container environments have recently gained significant momentum. 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 thesis, we tackle this key security challenge to container environments through a novel proactive approach. Our proposed 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, this approach 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 (e.g., up to 800 Pods). We demonstrate its deployability by integrating our solution with Kubernetes, one of the most popular container orchestrators.

Acknowledgments

I would like to express my deepest appreciation to my thesis co-supervisors, Dr. Suryadipta Majumdar and Dr. Lingyu Wang. Their endless support is in for a big part of this journey and I could not have done it without their help.

I would like to extend my sincere appreciations to my labmates Sima Bagheri and Mahmood Gholipourchoubeh. Their enthusiastic company inside and outside the lab has made this whole journey quite enjoyable. Special thanks to Dr. Yosr Jarraya for her precious pieces of advice.

My deepest thanks to my parents and my brother, for their unconditional support, and to Gilles, for accompanying me in this transatlantic journey.

I dedicate this thesis to my dear friend Florian. Repose en paix.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Problem Statement

Container environments are becoming increasingly popular for delivering microservices with increased scalability, reliability and observability [89]. In such environments, container orchestrators (e.g., Kubernetes [44]) 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) [5, 16].

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

[1] A study shows 91% of respondents use Kubernetes and 83% of them in production [12].

response times). Evidently, applying Open Policy Agent (OPA)/Gatekeeper [64] (the former is an open-source policy engine, and the latter the go-to solution for using OPA for Kubernetes admission control [44]) 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 further demonstrated through experiments in Section 4.3.2).
- 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 [64] (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 replicated states, which can be exploited by a malicious user to bypass security policies, as shown through an example in Section 2.2.

In this thesis, 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.

1.2 Thesis Contribution

In summary, the main contributions of this thesis 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 more practical response time (e.g., less than 10 ms for 800 Pods in contrast to 600 ms with OPA/Gatekeeper [64], one of the most popular existing approaches), making it suitable for low latency appications such as 5G's Ultra Reliable and Low Latency Communications [84].
- 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).
- In the design of ProSPEC, the verification step is deliberately decoupled from the enforcement step to enable easy integration with legacy policy enforcement engines.
- ProSPEC is integrated with the *de facto* standard orchestrator, Kubernetes [44], with the provision of porting it to other orchestrators (e.g., Docker Swarm [22], OpenShift [66]) with limited effort.

1.3 Research Gap

In summary, ProSPEC mainly differs from the state-of-the-art works as follows. First, many existing solutions leverage *Linux Security Features* or *Linux Security Modules* to address issues specific to the operating system (e.g., lack of process isolation, co-residency

detection, container escape) while leaving aside more generic security policies. ProSPEC provides a proactive security policy enforcement solution that prevents security compliance breaches. Second, there exists solutions that enhance security for specific orchestrator (e.g., Kubernetes) or container runtimes (e.g., Docker). ProSPEC instead embraces a larger range of concerns by focusing on technology-agnostic security policy enforcement (e.g., OPA) while still proving the possibility of integration with Kubernetes. Finally, many existing works on proactive security and policy enforcement were designed with VM-based cloud infrastructure (e.g., OpenStack) and are not designed to tackle the dynamicity and complexity of container-based environments. Instead, ProSPEC brings proactive security policy enforcement to container environments.

1.4 Related Publications

Conference Paper. Our work for proactive security policy enforcement for containers has been published as an article in a peer-reviewed conference's proceedings:

> ProSPEC: Proactive Security Policy Enforcement for Containers. Hugo Kermabon-Bobinnec, Mahmood Gholipourchoubeh, Sima Bagheri, Suryadipta Majumdar, Yosr Jarraya, Makan Pourzandi and Lingyu Wang. *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy (CODASPY'22)*, Apr 25-27, 2022, Baltimore-Washington DC Area, USA. (Acceptance ratio 30/111$\approx$27%)

1.5 Authors' Contribution

The student co-authors'contributions to the aforementioned article are as follows: Hugo Kermabon–Bobinnec contributed to the motivation, approach and design, implementation

as well as experiments on the impact of cluster size, threshold and predictions on response time. Mahmood Gholipourchoubeh contributed to the experiments on learning time and model accuracy (whose corresponding sections have been excluded from this thesis).

1.6 Outline

The rest of this thesis is organized as follows: Chapter 2 provides the necessary background for this dissertation. Chapter 3 covers the literature related to this thesis. We present our solution for proactive security policy enforcement for containers in Chapter 4, while Chapter 5 presents extensions related to predictive model learning and rule proactivization priority. In Chapter 6, we present two actively used deployments that emerged from this thesis. We conclude in Chapter 7.

Chapter 2

Background and Motivation

This chapter 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 [67] allow companies to deploy their own cloud infrastructure over virtual machines (VMs) [89]. 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 [26] 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 the requests that are placed 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.

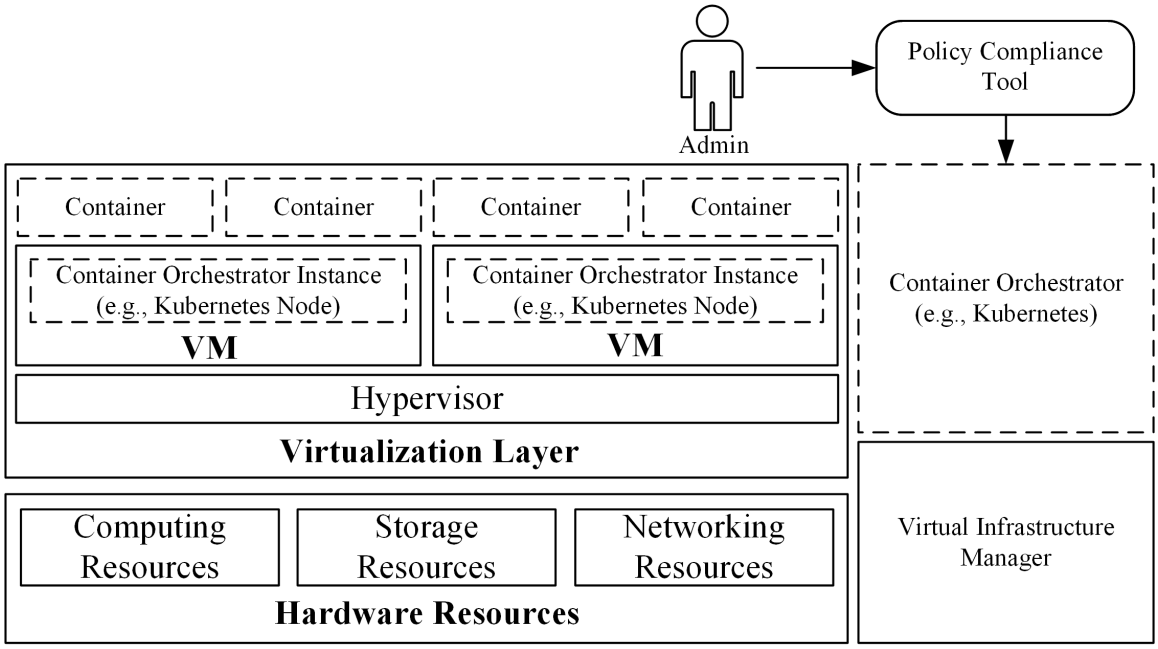


Figure 1: The ETSI architecture of a container environment [26]

Predictive Model.

Predictive models are used to predict future events by analyzing or mining patterns from historical data [31]. It relies on the assumption that events happening in an environment are correlated, meaning that given the past events in a system, different events are more or less likely to happen in the future. Fig. 2 shows an example of predictive model based on four arbitrary events and probability. For instance, after the door is closed, it will be locked

90% of the time (i.e., there is 90% chance the door will be locked after it is closed). On the other hand, there is a 5% chance the door will be reopened after it is closed.

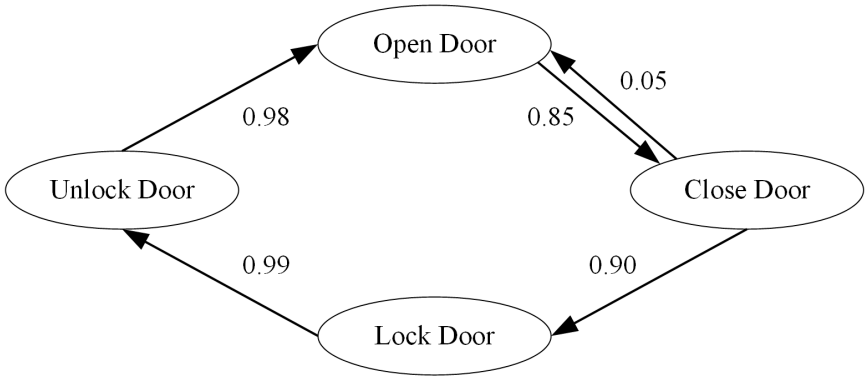


Figure 2: An example of predictive model

2.2 Motivation

To make our discussion more concrete, our motivating example will be based on Kubernetes [44] as the container orchestrator and OPA/Gatekeeper [64] 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 [16]. 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. 3, 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). Due to this vulnerability (CVE-2021-43979 [17], which was discovered by us), the data replication delay leads to security policy bypass and represents a critical security issue in the Kubernetes cluster.

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.

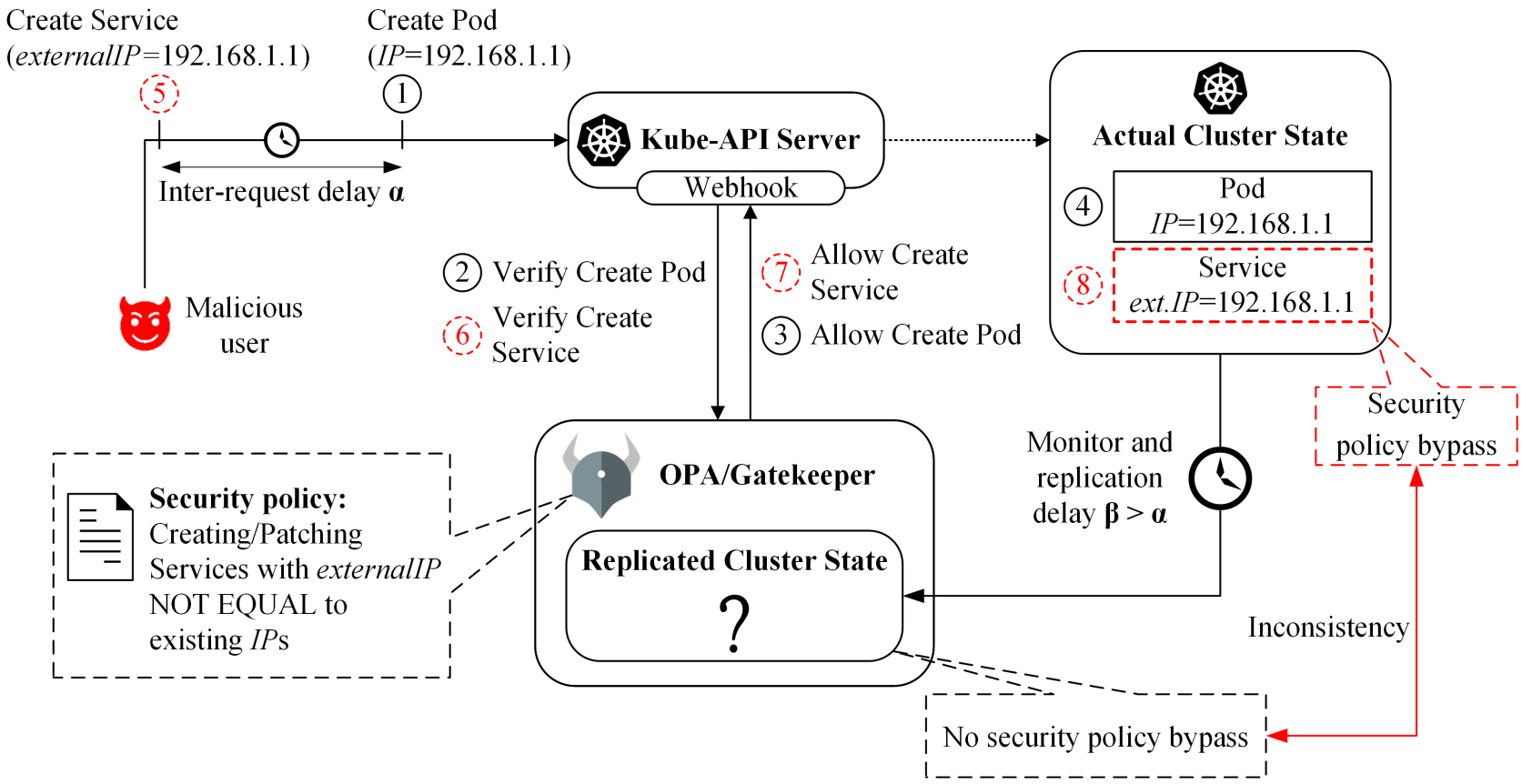


Figure 3: Policy bypass due to data replication delay

2.3 Threat Model

In-scope Threats. The in-scope threats include both external attackers and malicious insiders (such as co-located users in a cluster). 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 (e.g, [57, 83]), 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 [6, 49].

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 thesis. 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 (e.g, [57, 83]), we do not consider attackers who can tamper with (either through attacks or by using insider privileges) the Kubernetes system or the ProSPEC solution itself. We are not interested in detecting specific attacks; we focus instead on policy breaches that can result from various causes (e.g., attacks, misconfigurations, vulnerabilities, etc.).

Chapter 3

Related Work

In this section, we review the literature on container security and security policy compliance.

3.1 Container Security

Linux-based Solutions. Container security use cases are: i) attacks from applications to containers, ii) attacks from one container to another, iii) attacks from a container to the host and iv) attacks from the host to a container. The researches regarding container security take advantage of *Linux security features (LSF)*, *Linux security modules (LSM)*, and *Hardware-based solutions* [75] to protect containers against attacks.

The authors in [42, 30] take advantage of the LSFs *Namespace* and *CGroup* to prevent escape attack and power attack. Particularly, authors in [42] use namespace inspection to look for differences between the namespace of a container and the namespace of a program it spawned. They detail two ways processes can escape their original container: by exploiting arbitrary kernel code execution and mounting the `init` process filesystem in the container, then switching the container process to the host namespaces, or by modifying the virtual dynamic shared object (vDSO) through exploit such as CVE-2016-5195

(a.k.a, DirtyCow). In both cases, the resulting process namespace will be different from the container's it has spawned from. To detect such hints of container escape attacks, the authors enumerate processes in the system using the `pstree` command then compare the namespace ID of each container's `init` process with the ID of their spawned processes.

The authors in [30] present side-channels that adversaries could use to launch a power attack on a cloud infrastructure and propose an two-steps approach to mask the side channels and reinforce the container isolation model, thwarting such attacks. Particularly, they look for Linux pseudo-files that might not have been namespaced properly (and thus might leak kernel data from the host to the container) by comparing their versions accessed by both a containerized process and a host process. Authors then use this information to determine whether or not different containers are hosted in the same host and target co-resident container to effectively launch a power attack.

The LSF *Seccomp* is used by the authors in [33, 48, 20, 34, 23, 70] to limit the number of system calls used by the container in order to reduce the attack surface. Specifically, [33, 20, 34] use static code analysis on the application to build a list of necessary system calls and block the others, while [23, 70] aim for the same goal using dynamic code analysis. *Seccomp* is used in the Docker context (using `--security-opt seccomp`) to whitelist and/or blacklist system call in selected containers. Also, [48] divides the application runtime in two steps and restricts different system calls in both steps, reducing the attack surface even more. Authors make the difference between a container's boot phase and runtime (long-term) phase by analyzing the usage of system call over time. Therefore, they create two different *Seccomp* profiles, one for each phase. At runtime, *Seccomp* is updated with the second profile using a loadable kernel module. Even though they prevent the use of system calls that are not needed by the container applications, these approaches still leave a large number of frequent system calls needed by the application available to a potential attacker.

The authors in [76] use the LSM *AppArmor* and propose *security namespace* to isolate the security profile for each single container and independently define its own security policies. Precisely, instead of leveraging kernel security frameworks (*AppArmor* and *IMA*) policies system-wide (as it is by default), they introduce their usage in containers as autonomous entities. This way, each container can manage and use its own security framework for processes it hosts. Conflicts between system-wide and container-specific policies are handled by an Policy Engine specifically developed for that application. *DockerSec* [54] is based on *AppArmor* and enforces different policies to protect both the Docker engine and the containers. It leverages static analysis and runtime behaviour analysis to construct *AppArmor* profiles specific to each container, in addition to a profile for the runC process and the Docker engine (a.k.a, Docker daemon).

Use cases i, ii, iii rely on *LSFs* and *LSMs*. Hardware-based solutions such as *Intel SGX* and *virtual TPM* are used for use case iv [38]. ProSPEC differs from these approaches as it focuses on improving specific policies enforcement and does not leverage LSF, LSM, nor acts at the hardware level.

Container Security Verification. The Container Security Verification Standard (CSVS) is a framework established by OWASP [68] to formulate the security requirements for developing containerized applications. There are several works (e.g., [1, 19, 55]) on container security, particularly aiming at verifying the security of container images (e.g., [1]) and checking their integrity (e.g., [19, 55]). 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, [1] is a vulnerability-centric approach to identify and assess vulnerabilities in Docker containers images and proposes an OWASP aligned checklist of security use cases, compliant with the NIST [61] guidelines. It firsts verifies the integrity and content

of the container base images, performs some safety checks on the Docker image configuration files, then perform authentication and authorization checks on the image itself. Both [19, 55] propose solutions for containers integrity attestation covering the entire life cycle of the containers and their underlying images. By using Trusted Platform Module (TPM), [19] can verify all three host machine, container engine and the running containers, additionally offering the capability to detect which container is compromised. On the other hand, [55] preserves the privacy of the containers and the host from a remote untrusted integrity verifier. While these solutions aim at ensuring the containers and their images are safe, ProSPEC instead focuses on mitigating the potential impact of unsafe containers by leveraging policy enforcement.

Kubernetes Security. There are several research initiatives proposing different solutions addressing different security aspects in Kubernetes. According to authors in [74], the security best practices for Kubernetes are as follows: (i) API-based authentication and authorization request through authentication plugins 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 of the cluster. (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., [77, 27, 78]) 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* [77] provides a system-call level security attack detection approach while *Falco* [27] offers an online anomaly detection tool for containerized applications. The latter uses a kernel module to intercept system calls and matches them against user-defined rules to generate alerts. *KubAnomaly* [78] is a learning-based anomaly detection system, providing runtime monitoring capabilities in Kubernetes. Also, OPA [64] is a

security policy engine and, Gatekeeper as its sidecar, is an enforcement tool designed for Kubernetes. ProSPEC differs from those works as it proactively prevents policy violations which better protects the security of the container environment.

Additionally, some works focus on container runtime security. *Aqua Security* [4] provides comprehensive security detection and protection solutions to improve the overall security of Kubernetes and containers. Its products include container runtime security, risk evaluation, automatic CIS benchmarking, penetration testing tools as well as resources admission control by leveraging OPA. Authors in [86] proposed an analysis of three existing Kubernetes container runtimes (i.e., runC, gVisor, Kata) and evaluated the security-performance trade-off. Results show that the default runtime runC outperforms security-enhanced solutions gVisor and Kata thus demonstrate the need to propose other efficient security solutions for Kubernetes. The scope of ProSPEC is different: we assume the integrity of the container runtime (i.e., the container engine) and instead focus on the security compliance across the entire container environment.

3.2 Security Policy Enforcement

There are several proactive security compliance verification works (e.g., [9, 67, 57, 56]) for non-containerized virtual environments (e.g., IaaS such as OpenStack [67]).

For instance, Weatherman [9] and Congress [67] verify security policies in clouds using graph-based and Datalog-based models, respectively. Moreover, in [89], 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* [57] and *Proactivizer* [58] are proactive security auditing solutions for cloud environments. In contrast to our work, those works are not specifically designed to tackle the 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. Authors in [28] propose an algebra for anomaly-free firewall policies for OpenStack. Many state-based formal models (e.g., [73, 51, 52, 24]) are proposed for program monitoring. All those approaches are not specifically designed to tackle problems specific to containerized environments.

Chapter 4

ProSPEC: Proactive Security Policy Enforcement for Containers

4.1 Methodology

Fig. 4 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.

4.1.1 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. We assume that the historical data is sound and represents the behaviour of the environment correctly, and we will evaluate our solution when this is not the case in Section 4.3. 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. 5a shows an example of inter-resource dependency relationship in Kubernetes [44], a major container orchestrator, where a Pod resource cannot be created unless a Namespace resource exists, and Fig. 5b 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 4.2.2.

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 [44]), 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.

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.

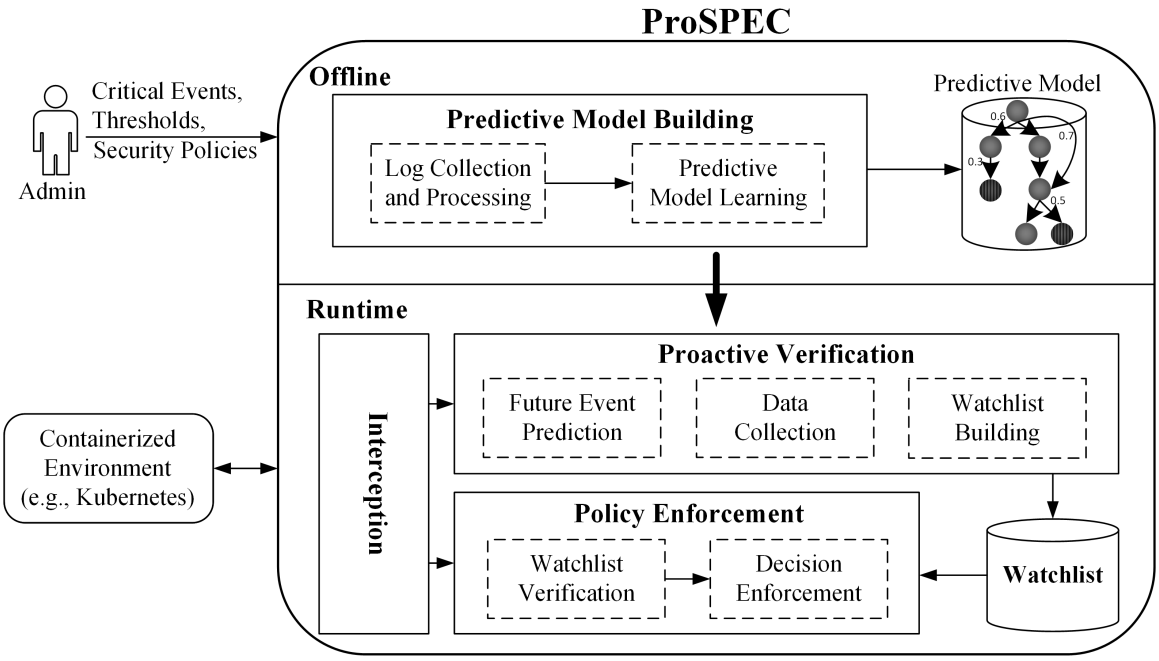


Figure 4: Overview of the ProSPEC approach

Second, to enable learning from collected logs, ProSPEC processes the log entries, identifies the event types and extracts meaningful sequences of events (*line 4 to line 8*). 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 4.2.2 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 (*line 9*). Afterwards, it extracts all immediate transitions between event-pairs from sequences and identifies them as edges between those event nodes (*line 9*). For instance, Fig. 6a shows an excerpt of the output with such nodes and edges for Kubernetes.

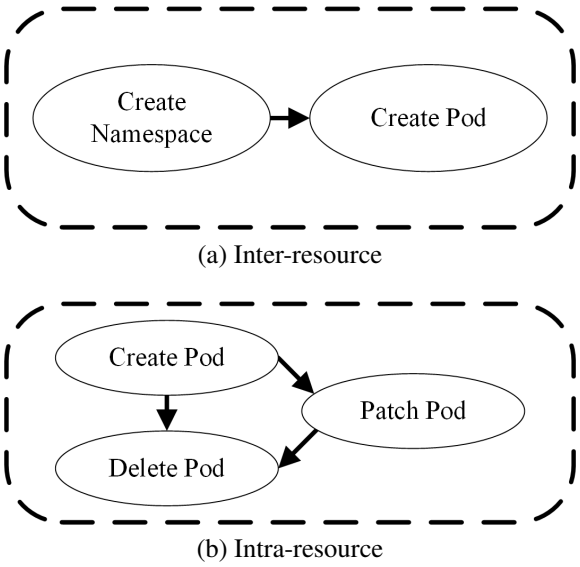


Figure 5: An excerpt of event dependencies in Kubernetes

- 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 [15] 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 (*line 10 to line 16*). Fig. 6b shows an example of such model with its additional edges.

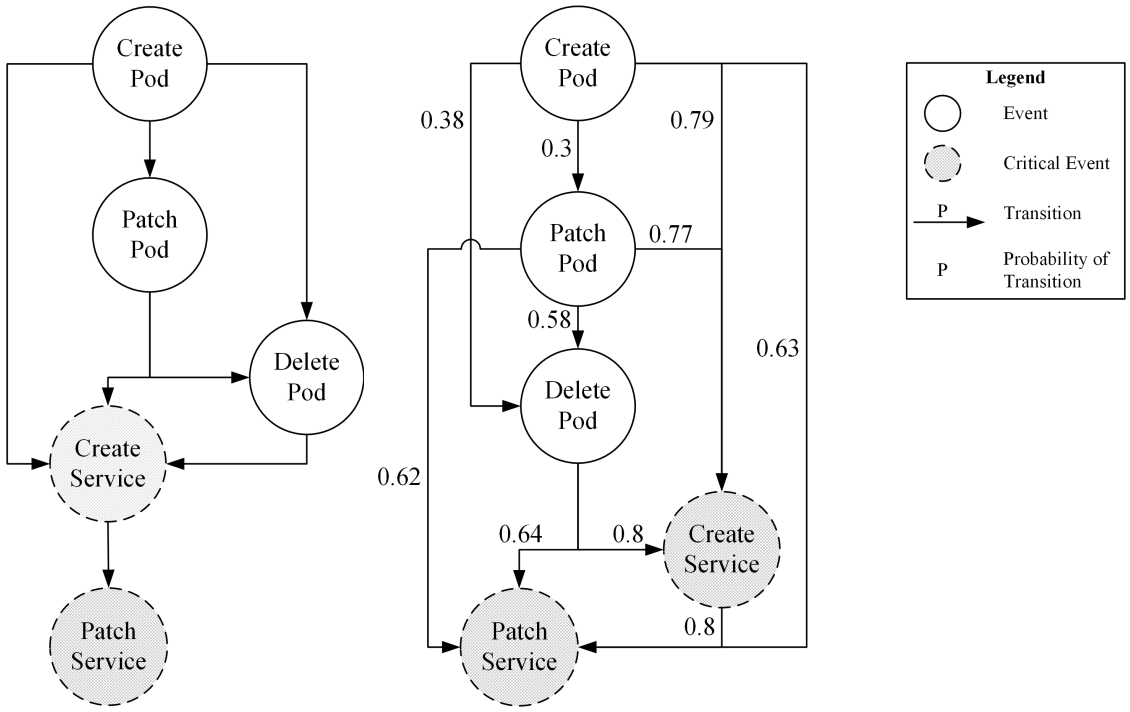
- Finally, ProSPEC learns the labels of the edges from the sequences of events (*line 17*). To that end, it establishes probabilistic dependencies by leveraging existing Bayesian network learning techniques [60] where the conditional probabilities indicate the likelihood for a (immediate or non-immediate) 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 (in Section 4.1.2), as demonstrated in Fig. 6b.

Example 1. Fig. 7 shows an example of applying ProSPEC's offline phase on a subset of logs. In step (1) and step (2), the log collection and processing module collects

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
```

and extracts the events "`Create Pod`, `Delete Pod`, `Create Service`, `Create Pod`, `Create Pod`, `Create Service`, `Patch Service`, `Create Pod`, `Patch Service`". Then, in step (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 step (4), and in step (5), ProSPEC identifies the unique four nodes of the model: `Create Pod`, `Delete Pod`, `Create Service`, and `Patch Service`. In step (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 step (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 step (8), it learns the conditional probabilities for transitions given the event sequences identified in step (4).



(a) Structure based on immediate transitions

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

Figure 6: ProSPEC predictive models

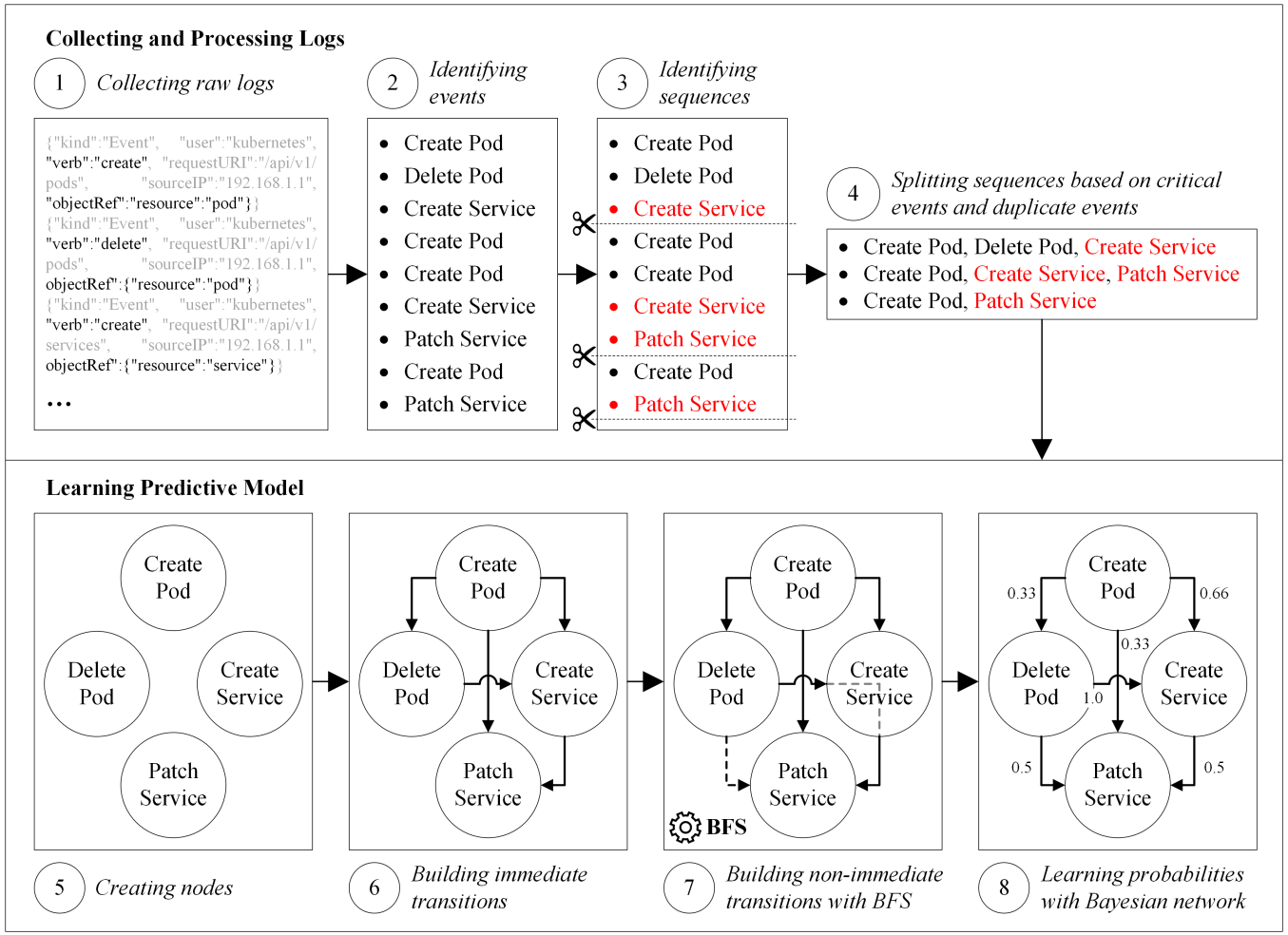


Figure 7: An example of offline learning

4.1.2 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 (*line 5*). 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 (*line 8 to line 11*).

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 (*line 8 and line 9*). Second, for such predicted events, it collects the existing resource data related to each security policy from the orchestrator (*line 10*). 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 (*line 10*). 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 4.3), 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 with respect to 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 4.3.

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 2. Fig. 8 shows an example of our runtime phase. For this example, we consider the same scenario as in Section 2.2, where CVE-2020-8554 [16] 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. 6b.

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

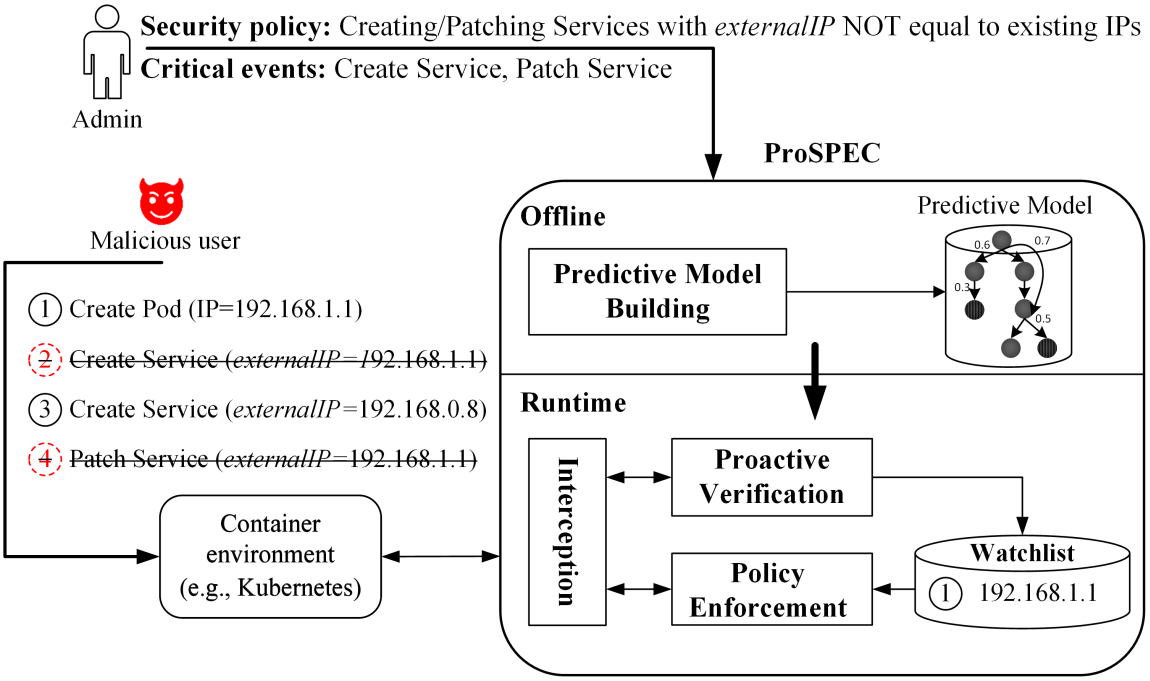


Figure 8: ProSPEC preventing CVE-2020-8554

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.

4.2 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.

4.2.1 ProSPEC Implementation

Fig. 9 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* [36] 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`

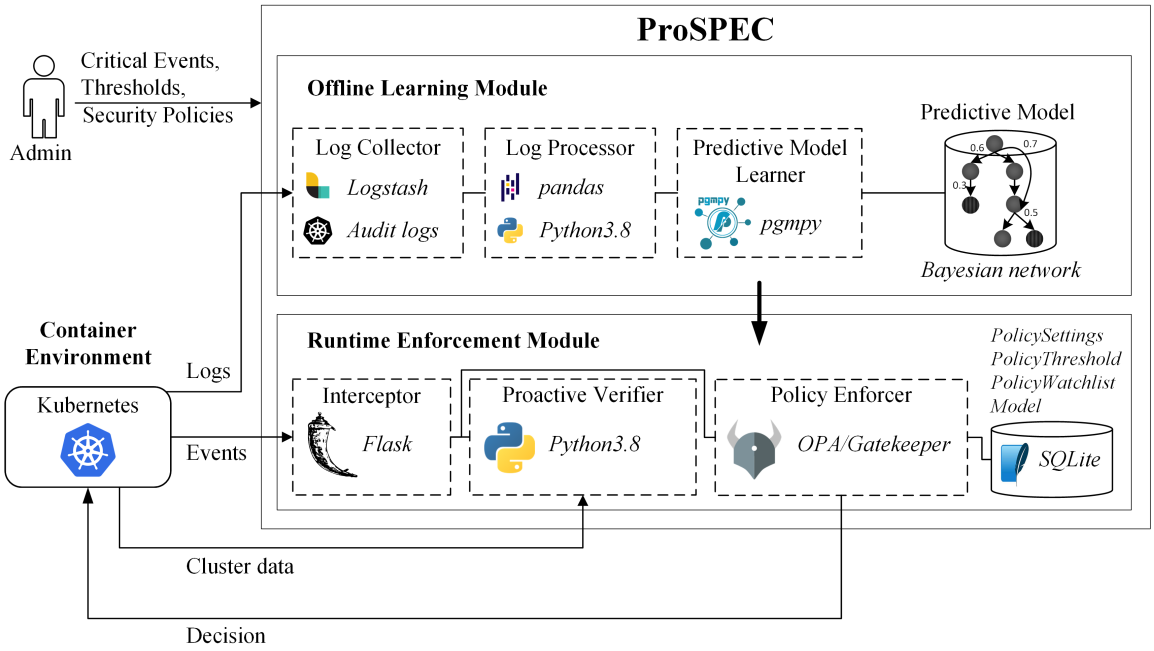


Figure 9: The architecture of our ProSPEC implementation

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 4.2.2). Afterwards, to process the audit log file and extract only the required information from the raw JSON audit logs, it leverages Logstash [53], 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* [88] 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 4.1.1. ProSPEC follows a standard iterative implementation [15] 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* [2], 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 4.2.2). The `interceptor` component runs as a local web server using the micro web framework *Flask* [35], 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 4.1.2, the `proactive verifier` component would query the API server with the following URI: `https://localhost:6443/api/v1/pods` (following the Kubernetes API reference [45]). 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 [64] 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.

4.2.2 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 [44] 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 [64] 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.

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

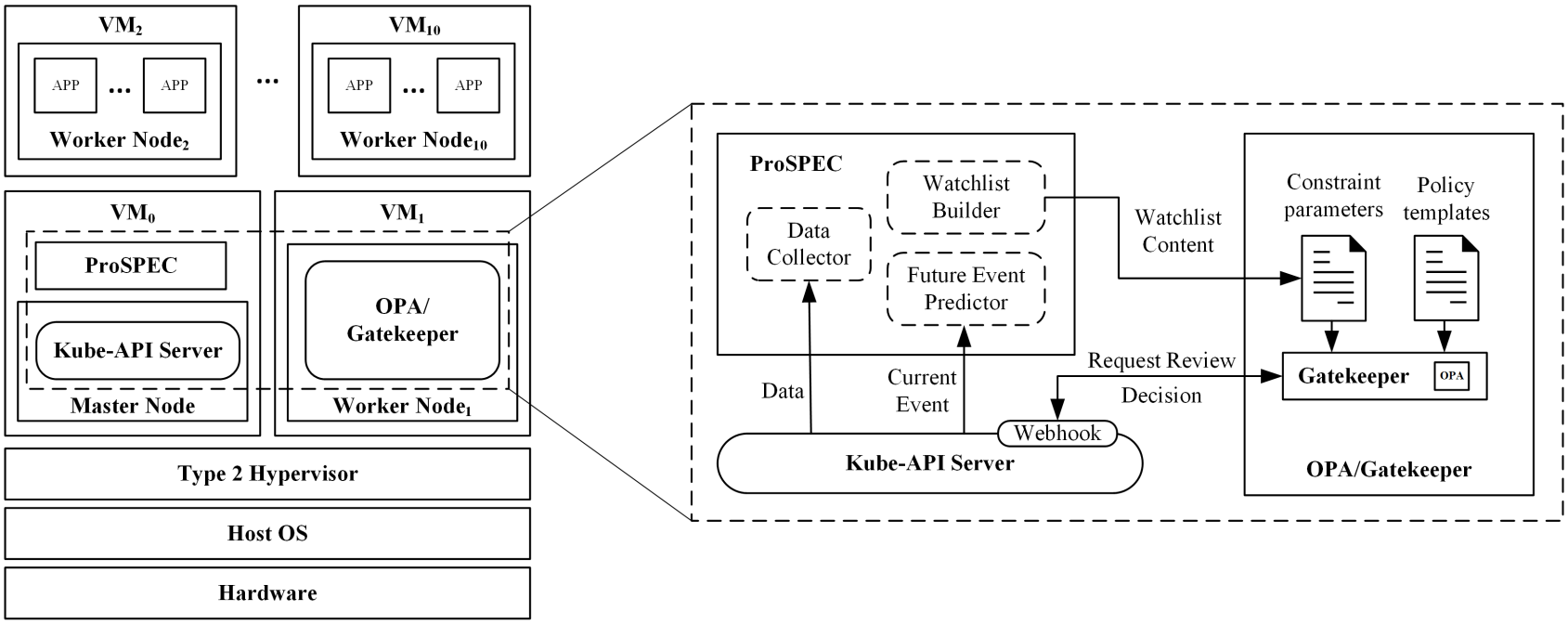
Table 1: An excerpt of Kubernetes events

Integration with Kubernetes. Fig. 10 illustrates the integration of ProSPEC with Kubernetes. Particularly, Fig. 10a provides a high-level overview of the integration including the deployment of a Kubernetes testbed, and Fig. 10b highlights the key integration aspects including how particularly ProSPEC is integrated with the Kube-API server and OPA/Gatekeeper, as detailed below.

- First, Fig. 10a 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 [40] are followed to deploy the cluster, including disabling swap partitions. The container environment is Docker v20.10.

- Second, Fig. 10b 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 watch-list 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 watch-list 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.



(a) High-level overview of the integration **(b) Detailed view of the integration**

Figure 10: Showing both (a) high-level overview and (b) detailed view of the integration of ProSPEC with 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 [22], OpenShift [66]). 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 [21]. 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).

Kubernetes [44]	**Docker Swarm [22]**	**OpenShift [66]**
Cluster	Swarm	Cluster
Pod	Task	Pod
Event	(Docker) Event	(OpenShift) Event
Namespace	Stack	Project
Admission Control	Third-party Plug-in	Admission Plug-in

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

4.2.3 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 4.2.2), 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 [46].

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 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 [25].

Feeding Watchlist Contents to OPA/Gatekeeper Constraints. We use OPA/Gatekeeper

for watchlist verification and policy enforcement in our implementation (as discussed in Section 4.2.2). 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) pre-defined 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 [81] or Constraint-Based estimation [60]. 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. 6b).

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 4.3. 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 [10] for our Kubernetes cluster as it is referred as one of the best network overlays in terms of performances [7]. 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` [14]. As a solution, we set up the Docker `cgroups-driver` to `systemd` as recommended in Kubernetes documentation [47].

4.3 Experimental Evaluation

4.3.1 Experimental Settings

Environment. The experimental environment is set up on our Kubernetes testbed described in Section 4.2.2. 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 [13], 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 [16, 79]: *Policy 1* (presented in our motivating example in Section 2.2) is inspired by a real-world vulnerability CVE-2020-8554 [16], and *Policy 2* is designed to prevent common misconfigurations in Kubernetes [79].

- **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*.

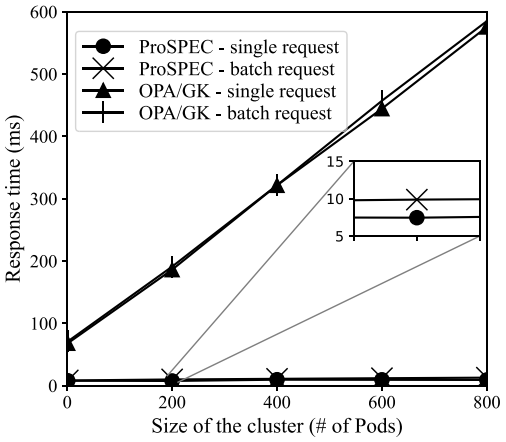
- **Policy 2.** This policy prevents a common Kubernetes misconfiguration called Ingress rules conflicts [79] in which deploying multiple Ingress rules to manage external access to Services can potentially lead to service failure and/or data exposure. To enforce this policy, ProSPEC dynamically maintains a blacklist containing all Ingress hostname rules in the cluster to prevent any Ingress rules conflict from happening.

4.3.2 Experimental Results

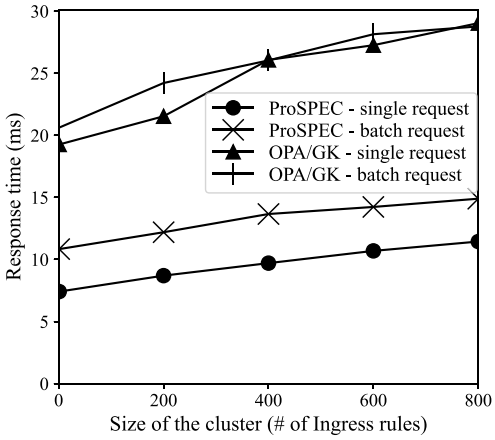
In the following, we evaluate the performance of ProSPEC in terms of response time, impact of wrong predictions, impact of threshold, offline learning time and rate of correct predictions.

Impact of Cluster Size on Response Time. The first set of experiments shown in Fig. 11 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.

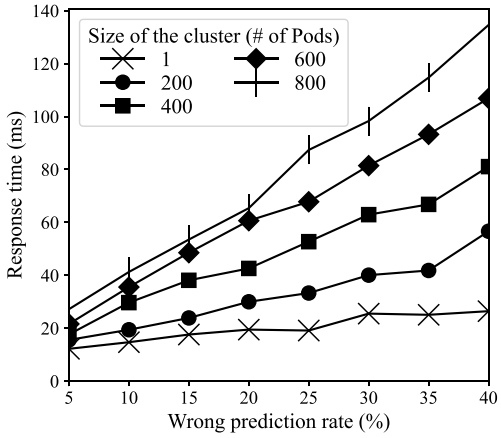
As specified in Section 4.2.3, the response time is measured directly at the decision engine level (i.e., OPA/Gatekeeper) to avoid any overhead due to external factors.



(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 11: Impact of the size of cluster and wrong predictions rate on the response time

Particularly, Fig. 11a 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. 11b 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.

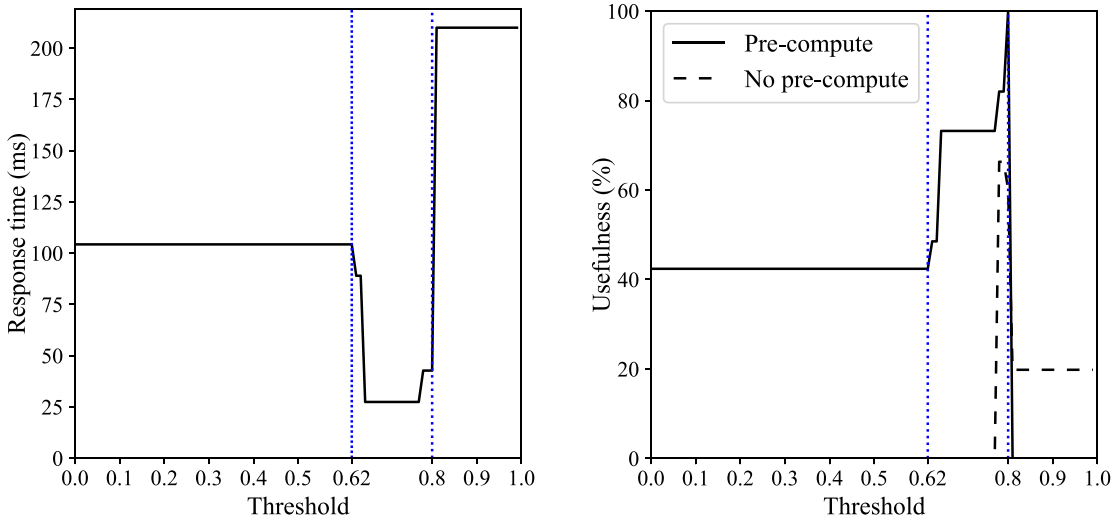
Impact of Wrong Predictions on Response Time. The second set of experiments is to measure the impact of wrong predictions by our predictive model on the response time of ProSPEC. Even though we assume that the historical data is sound and events can be

predicted correctly, wrong predictions can happen for different reasons, such as (i) an adversary purposefully introducing abnormal behaviour in the historical data to bias the predictive model (ii) a sudden change of behaviour in the environment due to updates (iii) a complex environment where it might be difficult to learn trends, or (iv) the lack of training data 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 4.1). We measure the overall response time (which includes both the pre-computation time measured from ProSPEC and the verification time measured from OPA/Gatekeeper).

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. 11c 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. 11a (without simulated errors), the response time is around 12 ms for 800 Pods. In contrast, Fig. 11c 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. 11a). As the error rate is likely much lower in reality, we can conclude that wrong predictions will not significantly affect the effectiveness of ProSPEC.

Impact of Threshold on Response Time. The third set of experiments is to measure the impact of different threshold values (as described in Section 4.1) 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).



(a) Impact of threshold on response time

(b) Impact of threshold on pre-computation efficiency

Figure 12: 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*

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. 12a and Fig. 12b show the response time of ProSPEC and the aforementioned usefulness metrics as functions of the threshold. Fig. 12a 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. 6b). 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 as we only pre-compute for events that are most likely to happen. Precisely, a threshold value between 0.65 and 0.78 reduces the average response to a minimum of 27 ms.

Fig. 12b 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. 6b). 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. 12a shows the lowest response time, and Fig. 12b 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.

More experiments on offline learning time and rate of correct predictions at runtime can be found in [43].

Chapter 5

Extensions

5.1 Predictive Model Learning

5.1.1 Introduction

Predictive model learning is a critical step in the ProSPEC methodology (as presented in Section 4.1). Previously, we only explored Bayesian network for this model. In this section, we further investigate two additional learning models (i.e., LSTM and n-gram) and compare their performance with the model obtained from Section 4.1.1 to find the best choice for this purpose.

5.1.2 Background and Motivation

In this section, we give a background about Kubernetes events and two different approaches to event prediction, namely, n-grams and Long Short-Term Memory (LSTM) network. Predictive models are introduced in Section 2.1.

Kubernetes Events. Kubernetes events are record of actions performed at the Kubernetes cluster level. Generally, a Kubernetes event is composed of a verb (such as create, delete, update, get, etc.) and a resource (such as a user account, a deployment, a configuration, a

service, etc.). A list of all existing events can be built using the Kubernetes API documentation [45] that refers all resources and their associated verbs. For instance, Kubernetes hosts are referred as Nodes in the API, and actions can be performed on them using the verbs Create, Patch, Replace, Delete, Read, List and Watch. By knowing the verb and the resource used in a request made to the Kubernetes API, one can deduct the corresponding event `verb_resource`.

n-grams. n-gram is an NLP technique to predict the occurrence of a word based on the previous $n-1$ words [41]. In other words, n-gram models are used to compute the likelihood of an item x_i to immediately follow a sequence of $n-1$ items $x_{i-1}, x_{i-2}, ..., x_{i-(n-1)}$. 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, in this thesis, we explore n-gram as a potential predictive model learner.

LSTM. Long Short Term Memory (LSTM) is a deep learning based technique that is widely applied in various applications (including security solutions) [37]. While traditional neural networks assumed that inputs are independent from each other (e.g., a cat/dog image binary classifier might independently be presented pictures of cats and dogs), recurrent neural networks (RNN) are to keep "memory" of previously inferred data on which depend the output (e.g., in a text corpus, the word "day" is more likely to be observed than "people" following the sequence of words "have", "a", "good", even though both the words "day" and "people" have approximately the same frequency in English language). Particularly, LSTM networks feature a long-term memory cell in order to deal with the vanishing gradient problem. Praised for their ability to recognize non-contiguous patterns (i.e., predicting future events based on relevant past events by ignoring unrelated events that occur in between), LSTMs are well-used for NLP applications and more generally to make predictions based on time series data. Therefore, we explore LSTM as a potential predictive model learner.

```
patch_pod,delete_pod
delete_pod,create_service,patch_service
patch_service
create_service
delete_pod
create_service,patch_service
create_pod,patch_pod,delete_pod,create_service,patch_service
delete_pod
create_service,patch_service
create_service,patch_service
patch_pod,create_service,patch_service
patch_service
create_service,patch_service
create_pod,delete_pod,create_service
delete_pod,create_service,patch_service
delete_pod,create_service,patch_service
delete_pod,create_service,patch_service
patch_service

```

Figure 13: Sample data extracted from our dataset

5.1.3 Approach

In this section, we describe the approach that is employed to prepare the Kubernetes logs for learning and train those models using n-gram and LSTM.

Data Preparation. The dataset consists of Kubernetes event logs that are used for ProSPEC's Bayesian model learning in Section 4.1.1. As no Kubernetes dataset exists, 10,000 sequences of Kubernetes events were generated synthetically based on a dataset collected at a smaller scale. Each sequences contains between 1 and 5 Kubernetes events. The dataset generation is detailed in [43]. Fig. 13 depicts a sample of dataset used for training and testing our different models. For each model, 80% of the dataset is used for training and the remaining 20% are used for testing the model properties.

N-grams Training. We choose to implement both a 2-gram and a 3-gram, i.e. the former will use very latest historical event to predict the next one, while the latter will use the latest two 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 keeps the absolute count of how many times they encountered each event after different inputs, then compute their relative probability of appearance.

We give here an example of 2-gram training. Given the training sequence (`create_pod`, `delete_pod`, `create_pod`, `delete_pod`, `create_service`), we first extract four input samples: `create_pod` $\rightarrow$ `delete_pod`, `delete_pod` $\rightarrow$ `create_pod`, `create_pod` $\rightarrow$ `delete_pod`, and `delete_pod` $\rightarrow$ `create_service`. The 2-gram's count table is represented in Table 3. The relative probabilities as calculated after the end of training are also given in the right part of the table.

Additionally, we give an example of 3-gram training. Given the same training sequence, we first extract three input samples: (`create_pod`, `delete_pod`) $\rightarrow$ `create_pod`, (`delete_pod`, `create_pod`) $\rightarrow$ `delete_pod` and (`create_pod`, `delete_pod`) $\rightarrow$ `create_service`. The 3-gram's count table is represented in Table 4. The relative probabilities as calculated after the end of training are also given in the right part of the table.

Input	Count			Probability of prediction		
	cre_pod	del_pod	cre_service	cre_pod	del_pod	cre_service
cre_pod	0	2	0	0	1.0	0
del_pod	1	0	1	0.5	0	0.5
cre_service	0	0	0	0	0	0

Table 3: An example of the count and prediction tables of a 2-gram after training

Input	Count			Probability of prediction		
	cre_pod	del_pod	cre_service	cre_pod	del_pod	cre_service
cre_pod, del_pod	1	0	1	0.5	0	0.5
cre_pod, cre_service	0	0	0	0	0	0
del_pod, cre_pod	0	1	0	0	1.0	0
del_pod, cre_service	0	0	0	0	0	0
cre_service, cre_pod	0	0	0	0	0	0
cre_service, del_pod	0	0	0	0	0	0

Table 4: An example of the count and prediction tables of a 3-gram after training

LSTM Training. Similarly to n-gram, LSTM can take as input sub-sequences of different sizes. Given the relatively small length of sequences in our dataset, we choose to experiment with two different window sizes of 1 event and 2 events. Results are presented in Section 5.1.4. The data is prepared in the same way that for n-grams, i.e., we extract sub-sequences of 1 (or 2) events, while the next event is being used as ground truth to correct the model. We consider a classic LSTM cell architecture as shown in Fig. 14, where x_t is the input information (e.g., a representation of the current event in the sequence), h_t is the output information (e.g., a representation of the next event in the sequence) and C_t is the state of the cell. U^f, U^i, U^g, U^o and W^f, W^i, W^g, W^o are the weights of the LSTM cells.

The equations describing the behaviour of the LSTM cell are defined as follows [32]:

- $f_t = \sigma(x_t U^f + h_{t-1} W^f)$ describing what information to keep and what information to forget in the cell state
- $i_t = \sigma(x_t U^i + h_{t-1} W^i)$ describing what information to update in the cell state;
- $\widehat{C}_t = tanh(x_t U^g + h_{t-1} W^g)$ describing what new information to update the cell state with;
- $C_t = f_t C_{t-1} + i_t \widehat{C}_t$ describing the new state of the cell after forgetting and updating the information;
- $o_t = \sigma(x_t U^o + h_{t-1} W^o)$ describing what information to output; and
- $h_t = o_t * tanh(C_t)$ describing the output of the LSTM cell as a filtered version of the cell state C_t.

Multiple cells that are put in a row are able to learn from historical data. During training, data is inferred in the network and a representation of the validity of the model (i.e., the loss function) is calculated as the distance between the LSTM predictions and the ground truth. After a certain amount of data is passed through the network (i.e., a batch), weights

from different layers are updated in order to minimize the loss. The accuracy of the model can then be tested quickly to get an idea of the progress of the training. After multiple iterations (i.e., epochs), the model is considered as trained. We then infer the testing dataset to measure the final accuracy.

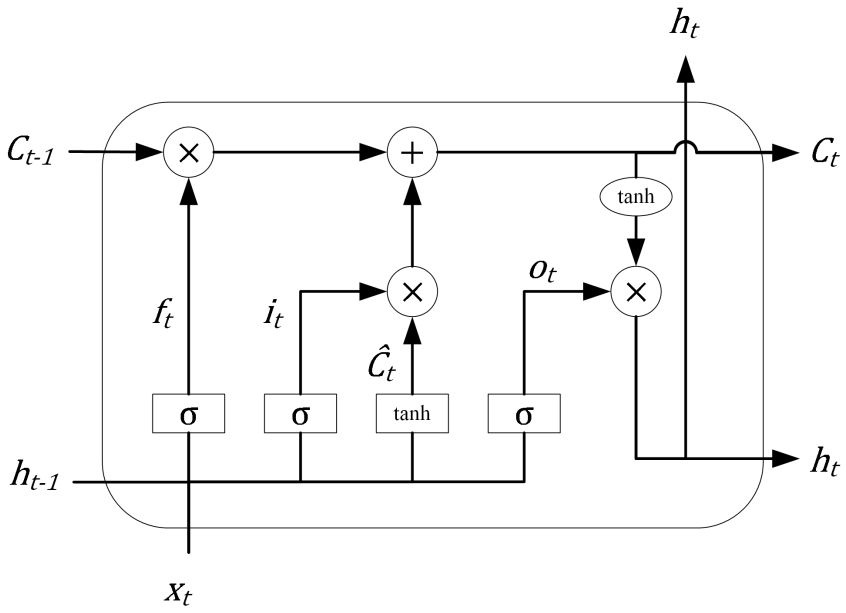


Figure 14: A classic LSTM cell as used for the LSTM model

5.1.4 Implementation and Results

In this section, we detail the implementation of those two predictive models and report experiment results. In order to compare the results, we test these models in similar fashion as in ProSPEC [43]. 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). The accuracy is defined as the count of correct prediction over the count of total prediction. Note that, in the context of ProSPEC, as Bayesian learning does not only learn immediate transition but more generally transitions happening in the future, a prediction is considered correct if the predicted event is present anywhere in the sequence

after the current event. The same accuracy calculation method has been employed for the three models in order to ensure the consistency of the results. In the following, we first describe the implementation of n-gram, then discuss an implementation of LSTM network, and finally present results.

N-gram. The n-gram models are implemented in Python using a dictionary data structure. The keys of the dictionary are the input sub-sequences represented as a single string 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 probability). Events in the dataset are first tokenized (i.e., we map an integer value with each event) using the *nltk* (Natural Language ToolKit) Python library [8].

LSTM. The LSTM models are implemented in Python using the Keras framework [11]. Our model architecture, described in Fig. 15, consists of three layers sequentially organized:

- an LSTM layer of depth 256, with recurring dropout with rate 0.2;
- a second LSTM layer of depth 128, with recurring dropout with rate 0.2; and
- a dense layer with a softmax activation.

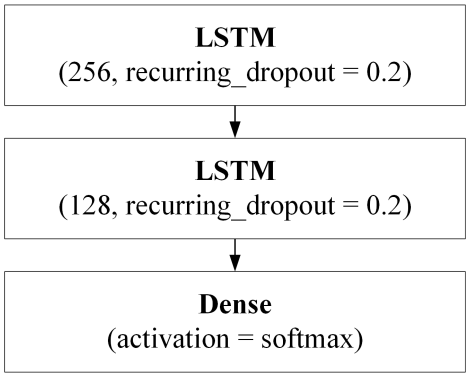


Figure 15: Architecture of our LSTM network

We choose an architecture with relatively small depth (two LSTM units) as we are dealing with relatively short sequences of events (maximum five). Also, as the vocabulary is composed of only five different events, we choose a relatively small width of LSTM (256 for the first cell and 128 for the second). Experiments not presented here have shown that, in our case, increasing both the width and the depth of the network did not help reach better accuracy. 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 overfitting. It is to be noted that the Dropout layers are only enabled during the training phase and are disabled during the testing phase. 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 (for binary classification, a sigmoid function would have been used). 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.

Results. The results of these experiments are reported in Table 5. Overall, both LSTMs and n-grams 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 (24 s for the LSTM with a window size of event compared to 4.3 s for the Bayesian network and 10 ms for the corresponding n-gram). They both additionally experience a larger inference time of around 60 ms while other models typically take less than one millisecond. On the other hand, the n-gram approach with $n = 3$ (i.e., a window size of 2 events) proves particularly efficient both from the accuracy and the timing point of view. However, results should be extended to a larger vocabulary size (i.e., more different types of event) to complete the comparison.

Approach	Size of Window	Accuracy	Offline Learning Time	Runtime Inference Time
Bayesian Network (ProSPEC)	*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

Table 5: Comparison of different predictive model learning approaches

5.2 Rule Proactivization Priority

5.2.1 Introduction

In Section 4.1, we propose and detail an approach to proactively prepare security policy for verification in containerized environment, such as Kubernetes. Security policies can manually be proactivized by security administrators using ProSPEC in a small and well-known environment, i.e., an environment where the watchlist content can easily be identified and where rules are precisely known in advance by the security administrator. However, overwhelming manual effort can be required in large-scaled deployments as many aspects of security are managed by security policies (e.g., Role-Based Access Control, network routing policy, authentications, intrusion detection, etc.). In that case, knowing where to focus efforts in order to reduce the overall response time of security enforcement can be a bottleneck. In this section, we propose to extend ProSPEC capabilities by introducing a methodology to benchmark and rank security policies rules according to different factors. This way, we offer an adaptable solution to help the security administrator gradually and efficiently improve the overall policy enforcement performance.

5.2.2 Background and Motivation

OPA/Gatekeeper [64] is a policy enforcement solution for Kubernetes. At its core, Open Policy Agent (OPA) is running in a container as a verification engine. Fig. 16 depicts the architecture of OPA/Gatekeeper as used in a Kubernetes cluster. OPA by itself can be used as a library directly embedded in a software or as a service directly on the host (e.g., Linux daemon). In the case of Gatekeeper, OPA is deployed as a library in a container alongside the necessary interface to make it queryable from the Kubernetes API through a webhook. Policies can be deployed directly in the Kubernetes cluster scope using two Custom Resource Definitions (CRD): a Templates and a Constraint. Policies are defined by first specifying a Template containing the Policy-as-Code (PaC) and some parameters, then by fixing the parameters using the Constraint CRD. Gatekeeper periodically queries the Kubernetes cluster to look for new Template and Constraints, which it internally transforms into regular OPA policies.

Fig. 17 depicts an example of policy file for OPA written in the Rego language.

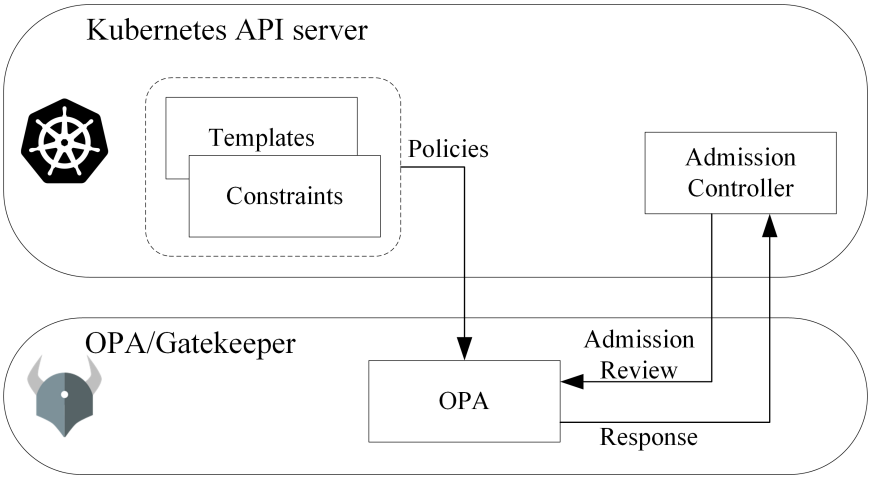


Figure 16: OPA/Gatekeeper deployment in the Kubernetes environment [72]

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 the profiling, where it measures the time needed to load

```
package kubernetes.validating.label

import future.keywords.contains
import future.keywords.if

deny contains msg if {
  value := input.request.object.metadata.labels.costcenter
  not startswith(value, "cccode-")
  msg := sprintf("Costcenter code must start with `cccode-`; found `%v`", [value])
}

```

Figure 17: Sample policy file for OPA/Gatekeeper written in Rego, a declarative language [71]

and compile the policy, and evaluate the query. It also displays the number of times each line of code of the policy is evaluated, the number of times it is re-evaluated and the total time spent on each line of code. `opa bench` will do benchmarking, i.e., it also provides the same metrics but also extensively run the evaluation a great number of times to provide statistical data. Both these commands allow the user to understand why a policy takes time to evaluate and point out exactly where in the policy code the evaluation spends most time, thus giving hints for optimization.

Motivation. OPA itself provides the end user with tools to evaluate policy performance. However, multiple issues are faced when it comes to OPA/Gatekeeper:

- First, even though profiling and benchmarking are available in OPA, these commands are not directly exposed to the Kubernetes users. More precisely, to evaluate the performance of policies performance used in OPA/Gatekeeper, one must (i) translate its OPA/Gatekeeper policies (i.e., Templates and Constraints) into a complete Rego policy file and (ii) access the OPA/Gatekeeper container command line (e.g., using either `kubectl exec` or `docker exec` commands) to use the `opa eval` or `opa bench` commands.

  Note that one can also choose to install OPA on his host machine and perform the

evaluation there, however this can greatly impact the evaluation results as performance on a host and performance inside a container on that host can greatly differ.

- To the best of our knowledge, no options are available to practically evaluate an entire fleet of policies at once. To do so, one must either compile all policies in one single file and deal with the overwhelming evaluation report, or evaluate its policies one by one, an operation that can be time-consuming in a large-scaled cluster.

- Knowing the performance profile of policies is one thing but in order to optimize its policy enforcement tool overall response time, one might be interested in knowing the frequency of usage of each policies (e.g., a policy which profiling indicates a large overhead might be considered negligible compared to a policy with smaller overhead if the former is used much less frequently than the latter). Although OPA does expose metrics regarding overall policy performance [63], these are not precise enough to determine which particular policies are queried the most and their cumulative overhead.

- Finally, there is no automated options to compare policies performance between them, i.e., one must manually go through multiple policies evaluation reports to prioritize among them which one can be improved using proactivization.

To illustrate further our motivation, we measure the execution time for each line of code of a sample OPA policy [62]. Results are displayed in Table 6. Bold values represent that 84.6% of the total execution time comes from only 15.4% of the total lines of code. Some lines of code (e.g., #2, #4, #5, etc.) are not present because they are not executed at runtime (e.g., they only serve for compilation). It can be observed that for this policy, the two most expensive lines of code in terms of execution time (i.e., #24 and #23) together account for almost 85% of the total execution time while accounting for only 15.4% of all lines of code. As a result, we observe that by solely focusing on the very most time consuming and

Line of Code	Execution Time (ns)	% of Total Execution Time	Cumulative % of Total Execution Time	Cumulative & of Total Lines of Code
#24	2239736	77.4	77.4	7.7
#23	208192	7.2	**84.6**	**15.4**
#22	95771	3.3	87.9	23.1
#1	94510	3.3	91.2	30.8
#25	81162	2.8	94	38.5
#3	72793	2.5	96.5	46.2
#6	34616	1.2	97.7	53.8
#16	26690	0.9	98.6	61.6
#17	21366	0.7	99.3	69.2
#11	7359	0.3	99.6	76.9
#9	4359	0.2	99.8	84.6
#12	3919	0.1	99.9	92.3
#15	3315	0.1	100	100

Table 6: Distribution of execution time for each line of code of an OPA policy

resource consuming policies, one can largely impact the overall performance of the system.

5.2.3 Approach

This section presents our approach to gather new and existing security policies, evaluate and rank them from the most expensive to the less expensive and present the end user with results.

Overview. Fig. 18 shows an overview of our approach using four modules and a `policy registry`. Our solution takes input from OPA and returns the list of the most expensive policies in terms of resource consumption and response time. First, the `policy watcher` module constantly monitors the presence of new policies (or update of previously known one) in the Kubernetes cluster. It stores information about the policies in the `policy registry`. The `profiler` module does profiling of resource consumption and response time of those policies.. To evaluate the policies performance profile, it starts by taking as input a query from historical data (e.g., the latest query ran against a

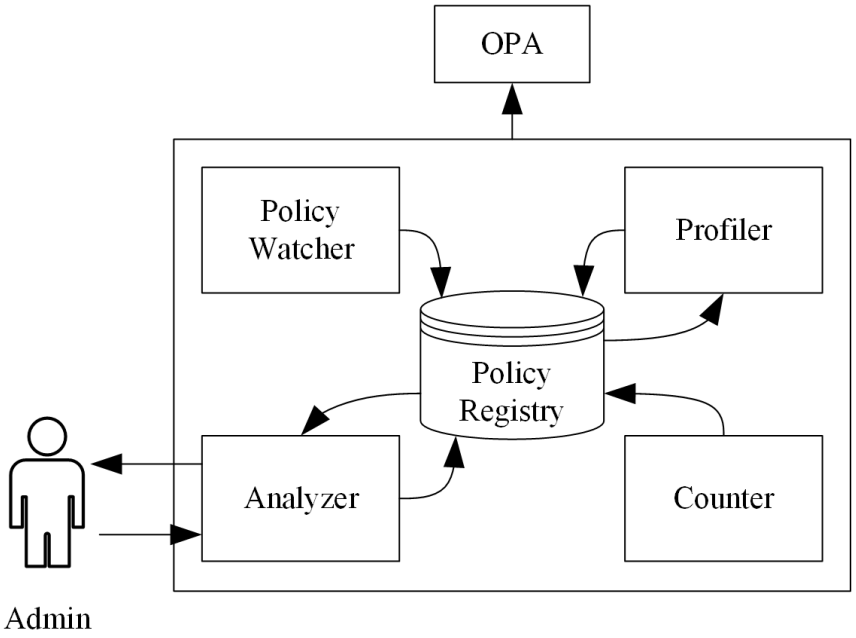


Figure 18: Overview of our approach to gather new and existing security policies, evaluate and rank them from the most expensive to the less expensive and present the end user with results

particular policy) then it performs fuzzing on that input data in order to obtain a more diverse performance profile. The `profiler` then uses OPA to do profiling and stores the performance results in the `policy registry`. Meanwhile, the `counter` module continuously tracks the usage frequency of each policy, in order to estimate their global time and computation consumption (done by the `analyzer`). These numbers are also continuously updated in the `policy registry`. Finally, the `analyzer` module queries the `policy registry` and ranks the policies by performance (from worst to best) according to different parameters specified by the admin. It also provides hints about what specific line of code can be improved (e.g., by using proactive computation [43]). 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 and stores the name of the policy. Even though Rego files can contain multiple policies per file, we consider here each policy independently as they are compiled by the OPA engine. In the context of Kubernetes and OPA/-Gatekeeper, we consider that one Constraint represent one independent policy (i.e., multiple Constraints associated with one same Template will be stored as multiple policies).

- *Count* is a value tracking the number of time a *Policy* has been used.

- *First_observed* contains the timestamp at which the policy was added to the database. This value is necessary in order to compute the frequency of usage of policies. It 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 age of the policy (i.e., the amount of seconds since its first deployment in the cluster) is computed at runtime by the `analyzer` module when ranking the policies.

- *Response_ time* is a value storing the response time value as reported by the `profiler` module.

- *Computation_resource* is a value storing the amount of resource consumed reported by the `profiler` module. It is measured as the average percentage of resource used over the total amount of resource available for CPU and memory as reported by htop [39] (e.g., if a policy benchmark reports a CPU usage of 14% and a memory usage of 26%, the average value of 20% is considered).

- *Profile_report* contains a complete profiling evaluation report as provided by the `profiler` module. 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 data used to verify against the policy, in `json` [69]

format. Practically, this usually correspond to the input data used in the first query observed for the policy.

Policy Watcher. The `policy watcher` module continuously scans for policies deployed in the Kubernetes cluster by querying the latter. For efficiency reasons, the `policy watcher` only periodically queries the Kubernetes cluster. It keeps track of the existing policies present in the database. On one hand, if a completely new policy is added, the `policy watcher` simply adds a new *(policy_name, first_observed)* entry to the database. It initializes the *count* to 0 and leave other fields empty. On the other hand, if an already existing policy has been updated, it resets its *count* to 0, updates the *first_observed* timestamp and deletes the existing *(response_time, computation_resource, profile_report, sample_data)* fields. This is because updated policies can have very different performance profiles and use different input data, thus they should be re-evaluated.

Profiler. The `profiler` module takes care of performance profiles for each policy. First, it periodically looks for policies in the `policy registry` that do not have a performance profile yet. To do so, the `profiler` simply looks for the presence of a value in the policy's *profile_report* fields. Then, if no profile yet exists (e.g., the policy was recently added or updated), it verifies if *sample_data* are available to run the profiling with. In that case, it uses the *policy_name* and *sample_data* values to establish a policy profile using `opa eval` command. The profiling process is described in Algorithm 3. The average query response time, the average computation resources as well as the average complete profile report are saved in the `policy registry`.

Counter. The `counter` module is responsible for incrementing the *count* field of a policy upon its usage. As soon as a query is run against a policy, the counter receives a signal from OPA containing the policy name as well as the data from the query. If the policy is queried for the first time (i.e., the `count` value was equal to 0), the `counter` module also adds the `json` data received from the query to the *sample_data* field. In any case, it increments

Algorithm 3 Profiling process

Input: PolicyName, SampleInputData
Output: ResponseTime, ComputationResource, CompleteProfileReport
procedure PROFILE(PolicyName, SampleInputData)
    for Policy in PolicyRegistry **do**
        if SampleData is not empty and CompleteProfileReport is empty **then**
            Get the corresponding $policy\_file$;
            for i from i=0 to i=100 **do**
                Fuzz the SampleInputData fields;
                Run `opa eval --data` $policy\_file$ `--data SampleInputData --profile --metrics --count=10`
                Save output;
            end for
            Average profiling results;
            Return ResponseTime, ComputationResource, CompleteProfileReport to PolicyRegistry;
        end if
    end for
end procedure

the *count* field by 1 for that policy.

Analyzer. The `analyzer` module role is to return the end-user with a ranking of policies for proactivization. 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 is of a parameter $\alpha \in [0, 1]$ expressing the percentage of response time to be considered into the ranking. A second parameter $\beta \in [0, 1] = 1 - \alpha$ representing the percentage of computational resource to be considered into the ranking is implicitly calculated.

To compute the consumption score of a policy, we use the following formula:

$$Score = \frac{Count * (\alpha * ResponseTime + (1 - \alpha) * ComputationResource)}{(NowTime - FirstObserved)} \quad (1)$$

Note that we do not simply consider the number of time a policy is used as even a

computationally lightweight policy can have more cumulative response time that a recently enforced but inefficient policy. The latter should be prioritized for proactivization as it represents more computational effort and response time in the long term. Instead, we consider the frequency at which policies are used, i.e., the count of their usage over the amount of time they have been enforced.

Fig. 19 details an example using our approach. In step (1), the admin creates a new policy named "Block CVE-2020-8554" using Kubernetes CRD Template and Constraint. By querying the cluster, the `policy watcher` module detects this new policy in step (2) and adds it the `policy registry` in step (3) (*timestamp 1669329373*). In step (4), the new policy entry is initialized with a *count* of 0 and a timestamp corresponding to the current time. Once the new policy has been used one, the `profiler` fuzzes the sample input and uses the `opa eval` command to build a performance profile in step (5). It stores the results in the `policy registry` in step (6).

After a while, in step (7) (*timestamp 1669334850*), the admin decides to optimize its security policies performance, and privileges response time over resource usage by specifying a parameter $\alpha = 0.7$ to the `analyzer` module. The latter calculates the score of each policies in step (8). 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 usage is 26%. The `analyzer` module thus calculates a score of 0.147 (note that this score has been scaled by a factor 10,000 for readability). It does the same for the policy "Label existence" and calculates a score of 38.718. In step (9), the admin knows that the policy "Label existence" should preferably be optimized first, as it has a higher score than "Ingress conflicts".

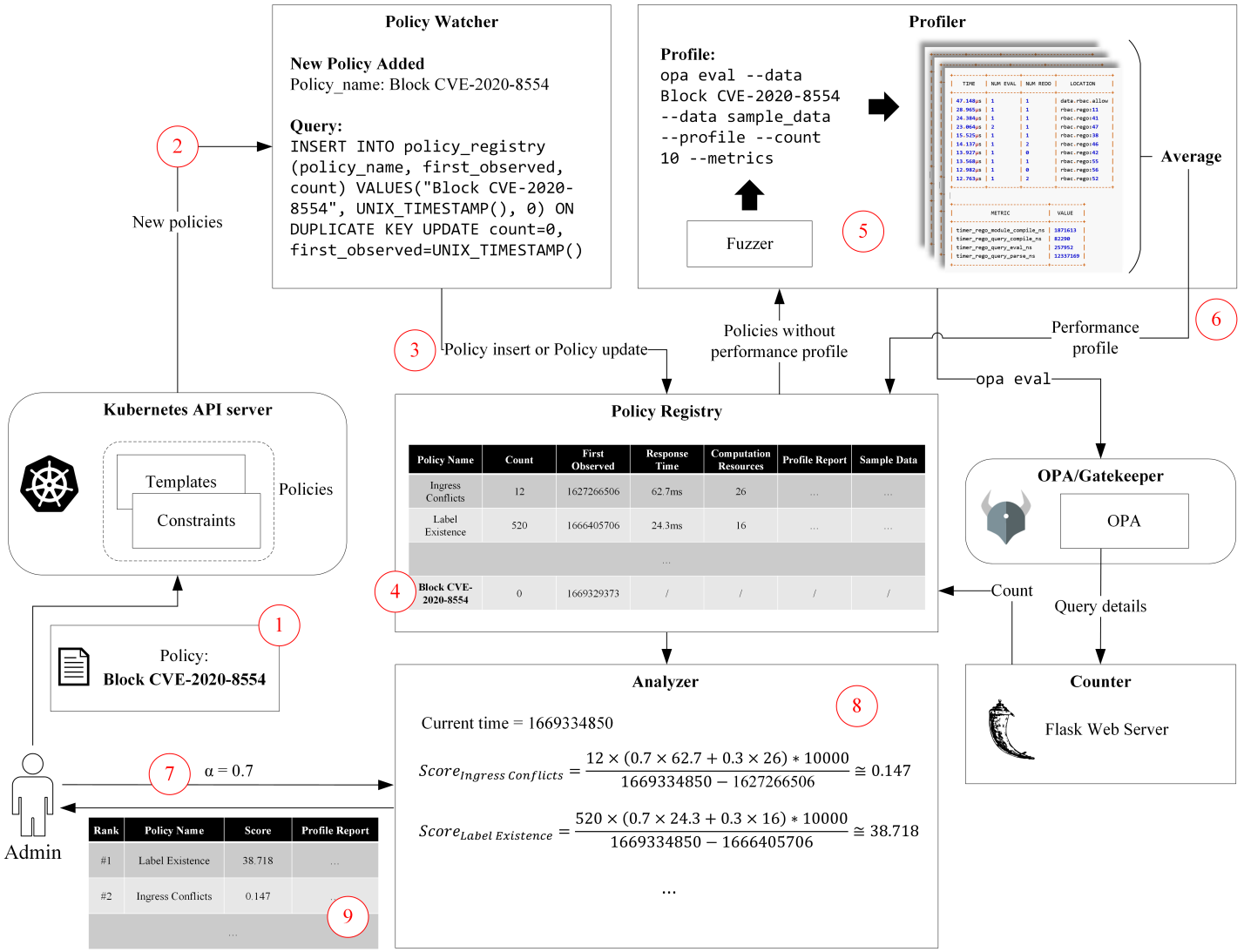


Figure 19: An example of our approach

5.2.4 Preliminary Results

In this section, we present preliminary results on how much response time and resource usage could be potentially gained by simply focusing on optimizing the right policies. To do so, we extend our study to five more policies and observe the distribution of response delay among the rules composing them.

For our study, we take five different policies from the Rego Playground [71] and obtain their performance profile using the `opa eval` command. These five policies concern RBAC, label existence on resources, label format on resources, container image safety and ingress conflicts, respectively. We do not limit the amount of output generated in order to get profiling for each single line of code executed. While results presented as motivation in Table 6 focus on the distribution of execution time among one single policy, these preliminary results instead focus on the distribution of performance among multiple policies.

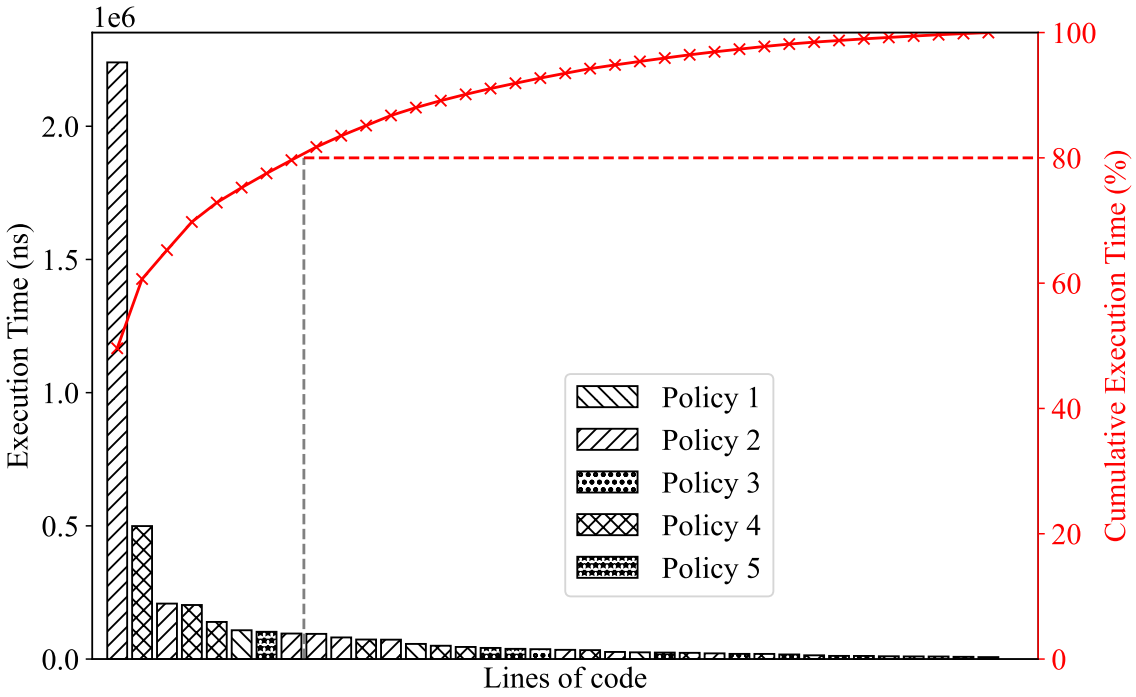


Figure 20: Distribution of execution time on five policies

Results are presented in Fig. 20. The cumulative execution time of these policies loosely follows a power law. More precisely, we observe that 20% of the policies' lines of code are responsible for 80% of the execution time, also known as Pareto's principle (more precisely, it follows a Pareto's distribution with Pareto's index $\alpha \approx 1.16$). We can conclude that, for this particular example, making only 20% of the policies rules proactive would lead in 80% savings in total execution time. To confirm this, more experiments should be ran on a large scale of policies in a real-world environment in our future work.

5.2.5 System Design and Integration

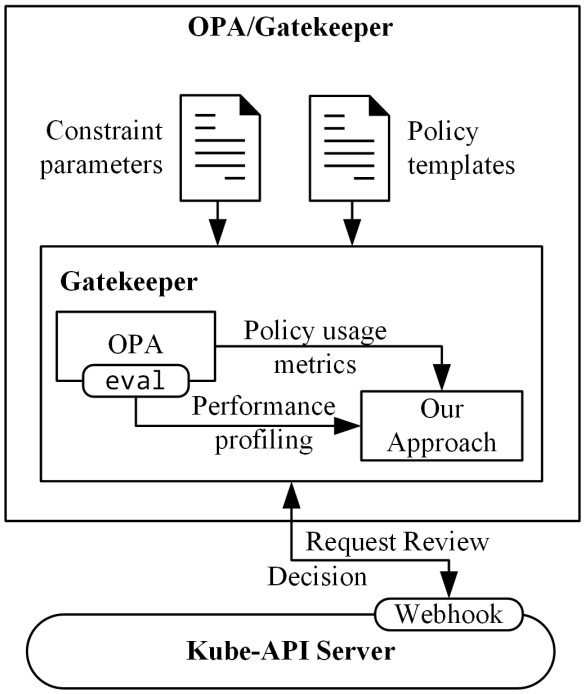


Figure 21: Integration of our solution in the OPA/Gatekeeper environment

This section details the design of our system and its integration with existing solution OPA/Gatekeeper. Our solution is deployed as a sidecar container in the same Pod as OPA/-Gatekeeper. We use SQLite as a lightweight database for the `policy registry`. 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 `counter` module is an HTTP server implemented with Flask [35] that receives information about a policy and the corresponding verified query each time a policy is used. To do so, the OPA source code of the `Rego.Eval()` function (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. Additionally, it leverages Linux's `htop` tool [39] to estimate CPU and memory usage 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 policies ranking and performance profiles as text. All database queries are done using Python and the `sqlite3` library. Fuzzing is done using open-source solution such as libFuzzer [50] (`pyfuzzer` in Python). Fig. 21 shows our integration alongside the OPA/Gatekeeper environment.

Chapter 6

Deployments and Implementations

Many technical aspects of this thesis were documented and automated for the sake of reproducibility and reusability. This chapter presents a background about the most important works realized during our research, details their topology and provides guidelines for using them. Finally, we highlight that the presented deployments have been utilized in several other research works conducted by the members of our group.

6.1 Kubernetes Cluster

The central thread of this thesis remains containers and their deployment in clouds. However, as cloud environments are often composed of multiple nodes, the required manual efforts to deploy, configure and maintain the environment are multiplied. Hence, the ability to deploy Kubernetes testbeds reliably and quickly was a key to progress and improve the research efficiency of anyone working on Kubernetes in our group. This section presents *quick-kubernetes*, a framework to automate the deployment of a Kubernetes cluster over virtual machines (VMs).

6.1.1 Background

This section provides backgrounds on technologies used in this deployment as follows.

- A container is a lightweight bundle of an application and its dependencies. By opposition to VMs which virtualize the physical hardware and the operating system, containers directly run at the operating system level, increasing portability and reducing starting time. Containers on the operating system are managed by a Container Runtime such as Docker, CRI-O or containerd.

- Kubernetes [44] is a container orchestrator, its role is to manage containers spread on multiple hosts (nodes). Particularly, Kubernetes ensures the maintenance and the availability of the applications and services deployed as container by communicating with multiple nodes and their local Container Runtime.

- VirtualBox [87] is a Type-2 hypervisor capable of managing multiple VMs. Particularly, VirtualBox provides virtualized physical layers (CPU, memory, network) on which VM images can be executed.

- Vagrant [85] is an open-source software to build and automate virtualized environments, such as VMs. Particularly, Vagrant offers the option to leverage Ansible to automate the software provisioning and configuration of those VMs.

- Ansible [3] is a provisioning and configuration tool that allows the automation of different steps of the deployment process, such as software installation and management of configuration files. Particularly, given a list of software to install and commands to execute, Ansible can connect to remote hosts and automatically perform all those operations while ensuring their correct completion and handling potential errors.

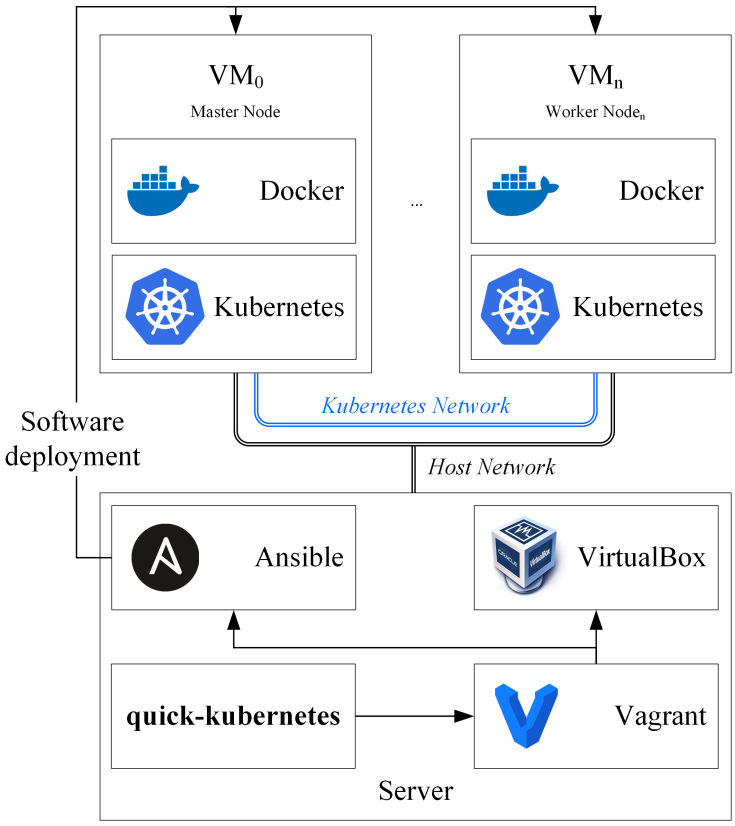


Figure 22: Quick-kubernetes automates the deployment of a complete Kubernetes cluster

6.1.2 Topology

Fig. 22 describes the topology of the deployment created by quick-kubernetes. Quick-kubernetes is a repository containing configuration files for Vagrant and Ansible [3] as well as prerequisites and guidelines to use it. Kubernetes is deployed using Docker or containerd as Container Runtime, upon the user's choice. It creates two networks: one internal network to support Kubernetes, allowing Nodes to directly reach others, and one host network to allow the Nodes to reach Internet and to allow the user to remotely access them (using NAT). The IP addresses of the Nodes are automatically assigned on both network, and can be customized. Quick-kubernetes provisions the Nodes with the necessary software to make Kubernetes work (e.g., Docker, Kubeadm, Kubectl, Kubelet, OpenSSL, Linux headers, Calico, etc.). More programs can easily be added in the configuration files if necessary.

Quick-kubernetes can create a Kubernetes cluster composed of one master Node and 10 worker Nodes hosted on VMs, *ex nihilo* and ready-to-go in less than 30 minutes.

6.1.3 Deployment Steps

Quick-kubernetes is hosted as a Git repository. Prerequisites are Ansible, VirtualBox and Vagrant. We install Ansible, VirtualBox and Vagrant using:

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ python3 get-pip.py --user
$ python3 -m pip install ansible

$ wget -q https://www.virtualbox.org/download/oracle_vbox_2016.asc -O- |
    sudo apt-key add -
$ sudo add-apt-repository "deb [arch=amd64] http://download.virtualbox.
   org/virtualbox/debian $(lsb_release -cs) contrib"
$ sudo apt update && sudo apt install virtualbox-6.1 -y

$ sudo apt install software-properties-common
$ curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -
$ sudo apt-add-repository "deb [arch=amd64] https://apt.releases.
   hashicorp.com $(lsb_release -cs) main"
$ sudo apt update && sudo apt install vagrant
$ export VAGRANT_EXPERIMENTAL="dependency_provisioners"
```

A `Vagrantfile` contains configuration for VirtualBox VMs and Vagrant. An example of `Vagrantfile` is given in Appendix B. Particularly:

- The parameter `N` is the number of Kubernetes worker nodes desired.
- `master.vm.network "private_network", ip: "192.168.61.10", nic_type: "Am79C973"` defines the IP address of the master Node, and `node.vm.network "private_network", ip:`

`"192.168.61.#{i + 10}", nic_type: "82545EM"` defines the IP addresses of the `N` worker Nodes, respectively.

- The network interface controllers (NIC) are set to allow IP forwarding (`--nicpromisc: allow-all`) as it is needed by the MACVLAN feature of Multus (detailed in 6.2).

`master-playbook.yaml` and `node-playbook.yaml` contain Ansible instructions for the master Node and worker Nodes provisioning, respectively. Particularly:

- Base packages (e.g., `ca-certificates`, `curl`, `linux-headers`, `software-properties-common`) are installed using `apt`.
- As required by Kubernetes official installation guide, memory swap is disabled.
- Calico is deployed as Container Network Interface (CNI) plugin for Kubernetes. Multus can be additionally installed as required in 6.2).
- Worker nodes automatically join the Kubernetes cluster upon creation. This is done by exporting the `join-command` from the Master node to the host then by importing it to each worker Nodes upon their creation.

Once configured as desired, the following command are executed:

```
$ vagrant up
$ vagrant ssh-config >> ~/.ssh/config
```

Kubernetes can then be accessed using `$ ssh k8s-master` and the `kubectl` command-line interface.

6.2 5G Core on Kubernetes

There is a need for telecommunications-related scenario and applications. On one hand, there exist open-source frameworks to emulate 5G environments such as Free5GC [29] or

Open5GS [65], providing 5G core applications. On the other hand, UERANSIM [82] is a user endpoint (UE) and radio access network (RAN) simulator that aims at simulating mobile devices registered in an operator's RAN. In order to build a 5G-related deployment on Kubernetes, we inspire from these projects and from towards5Gs [80] (maintained by Orange) to provide *kubernetes-free5gc*, a ready-to-use repository to quickly deploy a Free5GC/UERANSIM implementation on top of a Kubernetes cluster. Kubernetes-free5gc is fully compatible with Kubernetes clusters created using quick-kubernetes (see Section 6.1).

6.2.1 Background

This section quickly provides background about technology used in this deployment as follows.

- Calico [10] is a Container Network Interface (CNI) plugin for Kubernetes that allows Pods to communicate together, but also to specify network traffic rules for Pods and Services.

- Multus [59] is also a CNI plugin used to provide additional network interfaces to containers in a Kubernetes cluster. Particularly, Multus can use MACVLAN to create multiple sub-networks inside Calico networks and thus connect containers through different interfaces.

- Free5GC [29] is an open-source project offering a 5G core network functions (NF) implementation as defined by 3GPP specifications. Specifically, Free5GC provides source code and binaries for the following NFs: AMF, SMF, AUSF, UDR, NRF, PCF, UDM and UPF. It also provides configuration files and many guidelines about how to deploy and use a 5G core on top of VMs. It also provides hints on using UERANSIM

to simulate UEs and a RAN, and to perform tests to validate the behaviour of the 5G core.

- UERANSIM [82] is an open source project offering a simulation of UE and RAN (i.e., mobile phones and antennas/base stations connected to a 5G network). Users have the possibility to emulate UE devices and perform actions such as registration, deregistration, access to the data network (mobile data) and mobile roaming (i.e., moving from the area covered from one base station to another).

6.2.2 Topology

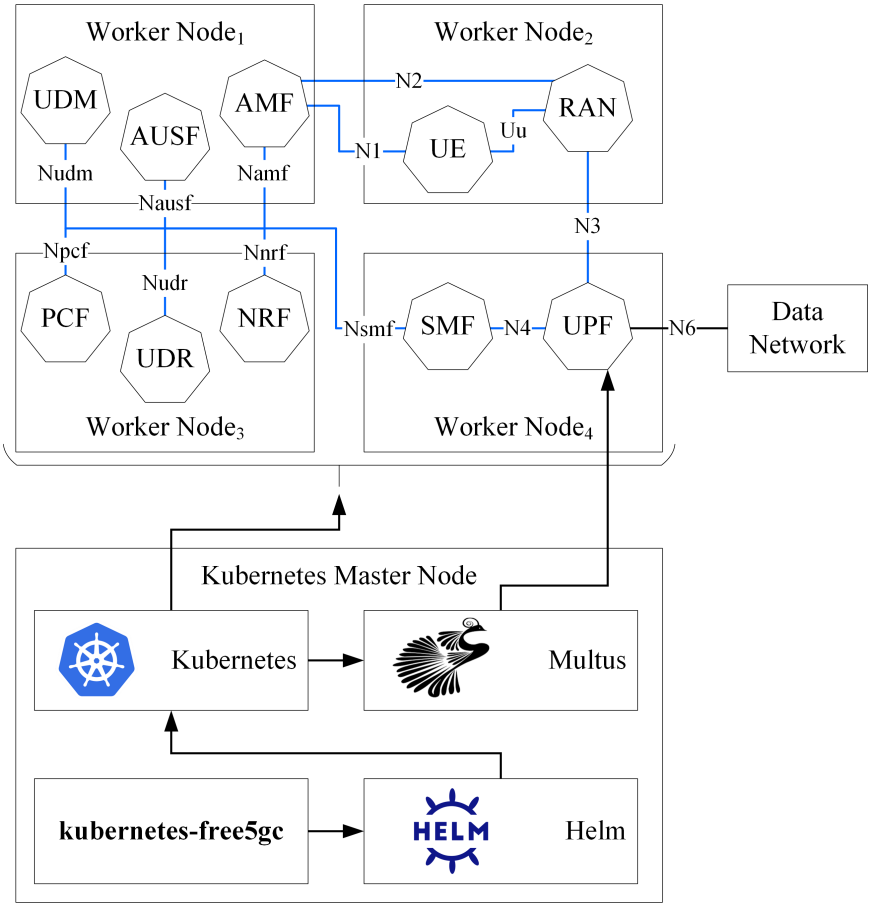


Figure 23: Kubernetes-free5gc automates the deployment of 5G core in Kubernetes

Fig. 23 details an example of topology of a 5G core and UE/RAN deployed over a Kubernetes environment composed of 4 worker nodes (i.e., such as described in Section 6.1) by kubernetes-free5gc. The 5G core NFs (namely, AMF, AUSF, NRF, PCF, SMF, UDM, UDR and UPF) are running Free5GC binaries while the RAN and the UE are from UERANSIM. We use Helm, a tool for configuring complex deployment in Kubernetes, to deploy the applications and build the network interface between them. Particularly, Multus uses MACVLAN on the Kubernetes network to create different network interfaces inside the containers. For instance, on the UPF container, the N3 and N4 interfaces should be connected to the Kubernetes network (in blue) while the N6 interface should be able to reach the Data Network (i.e., the Internet) on the Host network using NAT (see Section 6.1). Other interfaces between NFs are automatically created and IP are assigned inside the Kubernetes network. Finally, UEs can register to the 5G network and reach the data network through a tunnel, similarly to a scenario where a mobile user reaches out the Internet using a smartphone and mobile data.

6.2.3 Deployment Steps

Kubernetes-free5gc is hosted as a Git repository. Pre-requisites are Helm, GTP5G and Multus. To install them, following commands are executed:

```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/
   main/scripts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh

$ git clone https://github.com/free5gc/gtp5g
$ make clean && make
$ sudo make install
```

```
$ curl -fsSL https://raw.githubusercontent.com/k8snetworkplumbingwg/
    multus-cni/master/deployments/multus-daemonset.yml | kubectl apply -
    f -
```

As Free5GC uses a database, we create a database folder on one of the worker nodes. We then create a persistent volume and set a `nodeAffinity` to deploy the database on the same node. One should make sure to configure the network interfaces correctly, particularly:

- Interfaces N2, N3 and N4 should be on the Kubernetes network; and
- Interface N6 should be on a network reaching the Internet, e.g. the host network in our case (see Section 6.1).

Once configured, the 5G core and UERANSIM can be deployed using Helm:

```
$ helm install free5gc -n free5gc free5gc/charts/free5gc
$ helm install ueransim -n free5gc free5gc/charts/ueransim
```

After registering the UE in the 5G core, we can validate the installation by trying to reach the Internet from the UE container:

```
$ kubectl -n free5gc exec -it <ue_pod_name> -- ping -I uesimtun0 www.
    google.com
```

The UE should be able to reach the Internet through the newly created mobile data interface `uesimtun0`.

6.3 Usage in Security Research Works

Quick-kubernetes (see Section 6.1) was created to fill the need for a reliable, quick and customizable way to deploy a Kubernetes cluster as a non-negligible part of research in the department is shifting from the VMs environment (e.g., using OpenStack) to containerized

environments. Similarly, kubernetes-free5gc (see Section 6.2) was developed in order to allow experiments and research on 5G environments. As a result, these projects are currently being used by ongoing research projects about federated learning on multi-cluster 5G cores, Kubernetes cluster security evaluation, and netowkr function integrity verification of 5G core on Kubernetes.

6.4 Vulnerability Discovery

As mentioned in Section 2.2, this research lead to the discovery of CVE-2021-43979 [17]. Particularly, while experimenting with OPA/Gatekeeper, we found out that the data replication mechanism could lead to inconsistencies between resources existing in the cluster and resources known to OPA/Gatekeeper. As a result, one could bypass certain security policies enforced by OPA/Gatekeeper by concurrently performing multiple actions in the cluster. A Proof-of-Concept has been made publicly available [18]. This vulnerability has been reported to Styra, vendor of OPA/Gatekeeper; however it was not deemed relevant by the company as "Kubernetes states are only eventually consistent". Furthermore, Styra states that "the policy bypass would be detected afterwards by the audit feature of OPA/Gatekeeper". The CVE has then been placed in a "disputed" state after we informed MITRE of our discussion with OPA/Gatekeeper maintainers. Even considering that Kubernetes states, by design, reflect users' actions with a slight delay, we still believe that users should be aware of the policy bypass scenario and the behaviour of OPA/Gatekeeper data replication mechanism so they can adapt. Our Proof-of-Concept details a simple example depicting a policy bypass and teaches users about the reason behind such behaviour.

Chapter 7

Conclusion

In this thesis, we proposed 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.

We then extended our solution by exploring more predictive model learning techniques (e.g., n-gram and LSTM) and proposed an approach to analyze and rank the performance of security policies in order to facilitate their proactivization. We also contributed to improve the research capabilities in the field of container security and 5G security by developing two frameworks; one to automate the deployment of Kubernetes clusters, and the other to deploy a 5G core on Kubernetes premises. Finally, we discovered and reported an official vulnerability (CVE-2021-43979) in OPA/Gatekeeper, the *de facto* security policy enforcement solution for Kubernetes.

Limitations and Future Work. There exist a few limitations in our work as follows. First, ProSPEC does not retrain 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 this thesis, 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 our solution is integrated with Kubernetes. In the future, we plan to integrate it with other container orchestrators, such as Docker Swarm [22] and OpenShift [66].

Bibliography

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

[2] A. Ankan and A. Panda. pgmpy: Probabilistic graphical models using python. In *SCIPY*. Citeseer, 2015.

[3] Red Hat Ansible, 2022. `https://www.ansible.com/`.

[4] Aqua: Aqua Container Security Platform, 2021. `www.aquasec.com/aqua-cloud-native-security-platform/`.

[5] 'Azurescape' Kubernetes Attack Allows Cross-Container Cloud Compromise, 2021. `https://threatpost.com/azurescape-kubernetes-attack-container-cloud-compromise/169319/`.

[6] M. Bellare and B. Yee. Forward integrity for secure audit logs. Technical report, Citeseer, 1997.

[7] Benchmark results of Kubernetes CNI over 10Gbit/s network, 2019. `https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-august-2020-6e1b757b9e49`.

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

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

[10] Calico: Open source networking solution for Kubernetes, 2020. `https://docs.projectcalico.org/`.

[11] Chollet, Francois and others. Keras, 2015. `https://keras.io`.

[12] Cloud Native Computing Foundation 2020 Annual Report, 2020. `www.cncf.io/cncf-annual-report-2020/`.

[13] Cloud Native Computing Foundation 2020 Survey Report, 2020. `www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf`.

[14] Container Runtimes, 2021. https://kubernetes.io/docs/setup/production-environment/container-runtimes/#cgroup-drivers.

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

[16] CVE-2020-8554: Man in the middle in Kubernetes, 2020. https://blog.champtar.fr/K8S_MITM_LoadBalancer_ExternalIPs/.

[17] CVE-2021-43979, 2021. https://nvd.nist.gov/vuln/detail/CVE-2021-43979.

[18] CVE-2021-43979 on GitHub, 2021. https://github.com/hkerma/opa-gatekeeper-concurrency-issue.

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

[20] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 459–474, 2020.

[21] Docker authorization with OPA, 2021. www.openpolicyagent.org/docs/latest/docker-authorization/.

[22] Docker Swarm, 2021. https://docs.docker.com/engine/swarm/.

[23] DockerSlim, 2022. https://github.com/docker-slim/docker-slim.

[24] E. Dolzhenko, J. Ligatti, and S. Reddy. Modeling runtime enforcement with mandatory results automata. *International Journal of Information Security*, 14(1):47–60, 2015.

[25] Dynamic Admission Control, 2021. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/.

[26] ETSI. Report on the Enhancements of the NFV architecture towards "Cloud-native" and "PaaS", ETSI GR NFV-IFA 029. Technical report, ETSI, 2019.

[27] Falco, 2018. https://falco.org/.

[28] S. N. Foley and U. Neville. A firewall algebra for openstack. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 541–549. IEEE, 2015.

[29] Free5GC, 2022. https://www.free5gc.org/.

[30] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang. Containerleaks: Emerging security threats of information leakages in container clouds. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 237–248. IEEE, 2017.

[31] S. Geisser. *Predictive Inference*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1993.

[32] Gers, Felix A. and Schmidhuber, Jürgen and Cummins, Fred. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 10 2000.

[33] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *RAID*, 2020.

[34] S. Ghavamnia, T. Palit, and M. Polychronakis. C2C: Fine-grained Configuration-driven System Call Filtering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1257, 2022.

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

[36] R. D. Hipp. SQLite, 2020. https://www.sqlite.org/index.html.

[37] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[38] S. Hosseinzadeh, S. Laurén, and V. Leppänen. Security in container-based virtualization through vTPM. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 214–219, 2016.

[39] Linux's htop, 2020. https://www.man7.org/linux/man-pages/man1/htop.1.html.

[40] Installing Kubeadm, 2021. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/.

[41] M. Jardino. Multilingual stochastic n-gram class language models. In *1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings*, volume 1, pages 161–163. IEEE, 1996.

[42] Z. Jian and L. Chen. A defense method against docker escape attack. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, pages 142–146, 2017.

[43] H. Kermabon-Bobinnec, M. Gholipourchoubeh, S. Bagheri, S. Majumdar, Y. Jarraya, M. Pourzandi, and L. Wang. ProSPEC: Proactive Security Policy Enforcement for Containers. In *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy*, pages 155–166, 2022.

[44] Kubernetes, 2021. `https://kubernetes.io`.

[45] Kubernetes API Reference, 2021. `https://v1-18.docs.kubernetes.io/docs/reference/`.

[46] Kubernetes Audit Logs, 2021. `https://kubernetes.io/docs/tasks/debug-application-cluster/audit/`.

[47] Kubernetes: Configuring a cgroup driver, 2022. `https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/configure-cgroup-driver/`.

[48] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li. SPEAKER: Split-phase execution of application containers. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 230–251. Springer, 2017.

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

[50] LLVM libFuzzer, 2022. `https://llvm.org/docs/LibFuzzer.html`.

[51] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):1–41, 2009.

[52] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *European Symposium on Research in Computer Security*, pages 87–100. Springer, 2010.

[53] Logstash, 2021. `www.elastic.co/logstash/`.

[54] F. Loukidis-Andreou, I. Giannakopoulos, K. Doka, and N. Koziris. Docker-sec: A fully automated container security enhancement mechanism. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1561–1564. IEEE, 2018.

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

[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*. Springer, 2016.

[57] 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*. Springer, 2017.

[58] 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*. Springer, 2019.

[59] Multus CNI, 2022. https://github.com/k8snetworkplumbingwg/multus-cni.

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

[61] NIST: National Institute of Standards and Technology. www.nist.gov/.

[62] OPA Example Policy, 2022. https://www.openpolicyagent.org/docs/latest/#putting-it-together.

[63] OPA/Gatekeeper Metrics, 2022. https://open-policy-agent.github.io/gatekeeper/website/docs/metrics/.

[64] Open Policy Agent/Gatekeeper, 2019. https://open-policy-agent.github.io/gatekeeper/.

[65] Open5GS, 2022. https://open5gs.org/.

[66] OpenShift, 2021. https://docs.openshift.com/.

[67] OpenStack Congress, 2015. https://wiki.openstack.org/wiki/Congress/.

[68] OWASP: The Open Web Application Security Project, 2001. https://owasp.org/.

[69] Pezoa, Felipe and Reutter, Juan L and Suarez, Fernando and Ugarte, Martín and Vrgoč, Domagoj. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.

[70] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 476–486, 2017.

[71] Rego Playground, 2022. https://play.openpolicyagent.org/.

[72] Rita Zhang, Max Smythe, Craig Hooper, Tim Hinrichs, Lachie Evenson, Torin Sandall. OPA Gatekeeper: Policy and Governance for Kubernetes, 2022. https://kubernetes.io/blog/2019/08/06/opa-gatekeeper-policy-and-governance-for-kubernetes/.

[73] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[74] 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*. IEEE, 2020.

[75] S. Sultan, I. Ahmad, and T. Dimitriou. Container security: Issues, challenges, and the road ahead. *IEEE Access*, 7:52976–52996, 2019.

[76] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger. Security namespace: making Linux security frameworks available to containers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1423–1439, 2018.

[77] Sysdig, 2018. https://sysdig.com/.

[78] 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*, 1(5):e12080, 2019.

[79] Torin Sandall. OPA: Top 5 Kubernetes Admission Control Policies, 2020. https://thenewstack.io/open-policy-agent-the-top-5-kubernetes-admission-control-policies/.

[80] Towards5GS, 2022. https://github.com/Orange-OpenSource/towards5gs-helm.

[81] I. Tsamardinos, L. E. Brown, and C. F. Aliferis. The max-min hill-climbing Bayesian network structure learning algorithm. *Machine learning*, 65(1):31–78, 2006.

[82] UERANSIM, 2022. https://github.com/aligungr/UERANSIM.

[83] K. W. Ullah, A. S. Ahmed, and J. Ylitalo. Towards Building an Automated Security Compliance Tool for the Cloud. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1587–1593, 2013.

[84] Ultra Reliable and Low Latency Communications, 2022. https://www.3gpp.org/technologies/urlcc-2022.

[85] HashiCorp Vagrant, 2022. https://www.vagrantup.com/.

[86] W. Viktorsson, C. Klein, and J. Tordsson. Security-Performance Trade-offs of Kubernetes Container Runtimes. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–4. IEEE, 2020.

[87] Oracle VM VirtualBox, 2022. `https://www.virtualbox.org/`.

[88] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *SCIPY*, 2010.

[89] S. S. Yau, A. B. Buduru, and V. Nagaraja. Protecting critical cloud infrastructures with predictive capability. In *CLOUD*. IEEE, 2015.

Appendix A

List of Abbreviations

3GPP	Third-Generation Partnership Project
5G	Fifth Generation of broadband cellular network
API	Application Programming Interface
BFS	Breadth-First Search
CLI	Command Line Interface
CNI	Container Network Interface
(v)CPU	(virtual) Central Processing Unit
CRD	Custom Resource Definition
CRI	Container Runtime Interface
CVE	Common Vulnerability and Exposure
DAG	Directed Acyclic Graph
ETSI	European Telecommunications Standards Institute
LSTM	Long Short-Term Memory
MACVLAN	MAC Virtual Local Area Network
NAT	Network Address Translation
NF	Network Function
NIC	Network Interface Controller
OPA	Open Policy Agent
RAN	Radio Access Network
RNN	Recurrent Neural Network
UE	User Equipment
VM	Virtual Machine

Appendix B

Vagrant configuration file

```
IMAGE_NAME = "bento/ubuntu-20.04"
N = 2

Vagrant.configure("2") do |config|
    config.ssh.insert_key = false

    config.vm.define "k8s-master" do |master|
        master.vm.box = IMAGE_NAME
        master.vm.network "private_network", ip: "192.168.61.10",
   nic_type: "Am79C973"
        master.vm.hostname = "k8s-master"
        master.vm.provision "ansible" do |ansible|
            ansible.playbook = "playbooks/master-base-playbook.yaml"
            ansible.extra_vars = {
                node_ip: "192.168.61.10",
            }
        end
        master.vm.provider "virtualbox" do |v|
            v.memory = 8192
            v.cpus = 4
            v.customize ["modifyvm", :id, "--nicpromisc1", "allow-all"]
            v.customize ["modifyvm", :id, "--nicpromisc2", "allow-all"]
            v.default_nic_type = "Am79C973"
        end
    end

    (1..N).each do |i|
        config.vm.define "k8s-node-#{i}" do |node|
            node.vm.box = IMAGE_NAME
            node.vm.network "private_network", ip: "192.168.61.#{i +
   10}", nic_type: "82545EM"
            node.vm.hostname = "k8s-node-#{i}"
            node.vm.provision "ansible" do |ansible|
                ansible.playbook = "playbooks/node-base-playbook.yaml"
                ansible.extra_vars = {
                    node_ip: "192.168.61.#{i + 10}",
```

```
            }
        end
        node.vm.provider "virtualbox" do |v|
            v.linked_clone = true
            v.memory = 8192
            v.cpus = 4
            v.customize ["modifyvm", :id, "--nicpromisc1", "allow-
all"]
            v.customize ["modifyvm", :id, "--nicpromisc2", "allow-
all"]
            v.default_nic_type = "Am79C973"
        end
    end
 end
end
```