

# KubeFence: Security Hardening of the Kubernetes Attack Surface

Carmine Cesarano, Roberto Natella  
 Università degli Studi di Napoli Federico II, Italy  
 {carmine.cesarano2, roberto.natella}@unina.it

**Abstract**—Kubernetes (K8s) is widely used to orchestrate containerized applications, including critical services in domains such as finance, healthcare, and government. However, its extensive and feature-rich API interface exposes a broad attack surface, making K8s vulnerable to exploits of software vulnerabilities and misconfigurations. Even if K8s adopts role-based access control (RBAC) to manage access to K8s APIs, this approach lacks the granularity needed to protect specification attributes within API requests. This paper proposes a novel solution, *KubeFence*, which implements finer-grain API filtering tailored to specific client workloads. *KubeFence* analyzes Kubernetes Operators from trusted repositories and leverages their configuration files to restrict unnecessary features of the K8s API, to mitigate misconfigurations and vulnerabilities exploitable through the K8s API. The experimental results show that *KubeFence* can significantly reduce the attack surface and prevent attacks compared to RBAC.

**Index Terms**—K8s, Attack Surface, API Filtering

## I. INTRODUCTION

Kubernetes (K8s) [1] has become the mainstream platform for orchestrating containerized applications, enabling scalable deployment and management across distributed environments. Its adoption spans various critical domains, including finance, healthcare, and government services [2], [3], where security is crucial. As K8s becomes integral to these sensitive areas, ensuring the security of its components, particularly the K8s API, is of the highest importance.

K8s is a project with a huge codebase and a large, complex interface toward clients [4]. This interface provides convenience of use and feature-richness, but it also represents an “attack surface” that exposes the system to security attacks [5]. If these features are provided by vulnerable code, they can be exploited by malicious users to pursue attacks. In the case of K8s, attackers leverage vulnerabilities to run unauthorized workloads, such as cryptojacking and botnets [6]; moreover, attackers can violate the isolation between tenants in the infrastructures, such as disrupting applications and stealing or damaging their data [7].

To avoid such security risks, regulatory frameworks and security agencies are recommending the adoption of secure design practices [8]–[10]. In particular, software systems should adopt the “principle of least privilege”, by minimizing the attack surface to only provide access to the strict minimum of features and resources [11], [12]. In the case that a user behaves maliciously, they should be prevented from accessing unnecessary features that could be exploitable [7].

In practice, this approach is quite challenging to apply in complex systems such as K8s. K8s provides a REST API to manage resources, such as *Pods*, *Services*, *Deployments*, and several others. It adopts *role-based access control* (RBAC) [13] to manage API calls that access these resources. However, K8s

builds more complex abstractions on top of the REST API. When a resource is configured through the REST API, the API takes in input complex data structures (as YAML payload of a request) to describe the “desired state” of the resource (*specification*). These data structures can contain attributes to use advanced features, such as access to resources on the host machine and other special permissions. These features represent a risky attack surface that can be exploited in the case of software vulnerabilities in K8s.

We argue that RBAC does not suffice to secure K8s, since it only provides access control on resources as a whole, but lacks control over specific features in the specification of these resources. For example, while RBAC can allow users to manage *Pod* resources and disable access to *Deployment* resources, it cannot restrict the values of fields within the specification of a *Pod*. These features can be triggered by a malicious client in order to exploit vulnerabilities in K8s.

In this paper, we analyze the extent of the K8s attack surface and how to harden it. We initially present an analysis of CVE vulnerability records [14] for the K8s project, in which we found that CVEs are only exploitable through specific features of the K8s API interface. Then, we present a solution (*KubeFence*) to provide finer-grain control of the K8s attack surface to harden K8s against exploits and misconfigurations. Our solution analyzes applications that use K8s from trusted repositories (*Kubernetes Operators* [15]), and generates security policies that restrict API access to only the resources explicitly required by the application.

In our evaluation, we defined a catalog of malicious K8s specifications for testing *KubeFence*, based on known CVEs and common misconfigurations. We applied *KubeFence* on several popular applications. Our solution was able to mitigate all tested CVE exploits and misconfigurations. Moreover, we found that *KubeFence* achieved an average of 35% reduction in the attack surface compared to RBAC, while introducing an acceptable performance impact for non-realtime workloads. In particular, we measured an average overhead of 50 ms for cluster management operations (~21%). Since *KubeFence* only applies to cluster management, once containers have been deployed, it does not affect container execution.

The main contributions of this work are:

- We analyze K8s CVEs and found that vulnerable code is only exercised when specific fields are used in API requests.
- We design *KubeFence* to generate and enforce security policies for the K8s API interface.<sup>1</sup>
- We present a catalog of attacks against the K8s API interface, based on CVEs and on common misconfigurations.

<sup>1</sup>Available as open-source at: <https://github.com/dessertlab/kubefence/>.

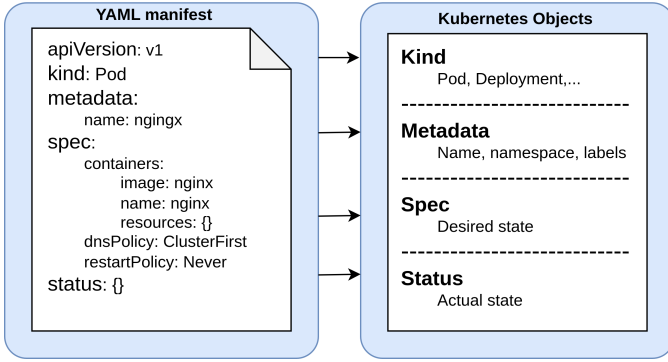


Fig. 1. An example of YAML Manifest to configure a Kubernetes Object.

- We evaluate the effectiveness of *KubeFence* at reducing the attack surface, at mitigating CVEs and misconfigurations, and at achieving efficiency with minimal runtime overhead.

## II. BACKGROUND

### A. Kubernetes

Kubernetes (K8s) is an open-source platform for automating the deployment, scaling, and management of containerized applications. At its core, Kubernetes abstracts underlying physical resources into logical entities called *Kubernetes resources*, such as Pods (for workloads), Services (networking), Volumes (storage), ConfigMaps (configuration), and Secrets (security). These resources are managed within a cluster, consisting of worker nodes orchestrated by a control plane. The K8s API Server is the central management component, responsible for handling operations within the cluster. It exposes a RESTful API that allows users to interact with the cluster by sending HTTP requests to create, modify, and delete resources. The API Server supports HTTP verbs like *get*, *post*, *put*, and *delete* to manipulate Kubernetes resources. Each resource is represented as a *Kubernetes Object*, typically defined using a declarative manifest file in YAML or JSON format. The manifest specifies the desired state of a resource, which the K8s control plane works to reconcile with the current state. Kubernetes Objects generally contain two primary nested properties, which are *spec* and *status*, to describe the desired and current state, respectively. This structure is typical for objects like Pods, Deployments, and Volumes, as shown in Figure 1. An example of a desired state is to bring up a given number of container replicas. Objects like Secrets and ConfigMaps, omit these fields, focusing on storing sensitive data or configuration in the *data* field. The K8s API exposes a set of endpoints for each resource. When a user defines a resource in a manifest, specifies all configurable fields, and applies this configuration to the cluster, an HTTP request is triggered to the relevant API endpoint, including all the resource configurations in the payload. This workflow allows users to configure resources declaratively by specifying their intent, while the API server translates these configurations into actual cluster state changes. However, these exposed API endpoints and configurable fields contribute to a significant attack surface, as discussed in Section III.

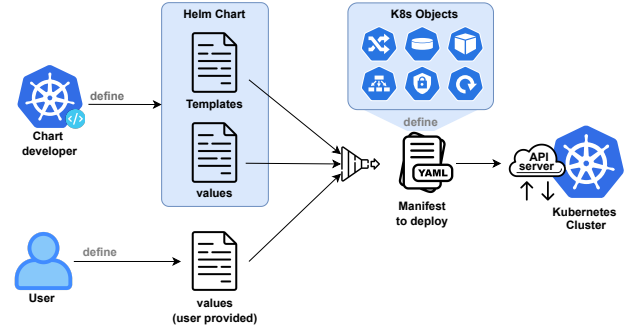


Fig. 2. Helm Template Processing.

### B. Kubernetes RBAC

To mitigate security risks, K8s provides an RBAC access control mechanism [13]. RBAC defines which users, groups, and service accounts can perform specific actions on resources, such as viewing, creating, modifying, and deleting them. RBAC policies are defined using YAML manifests through four kinds of Kubernetes Objects: *Role*, *ClusterRole*, *RoleBinding*, and *ClusterRoleBinding*. A Role or ClusterRole object contains rules that specify a set of permissions, while RoleBinding and ClusterRoleBinding objects grant the permissions defined in a role to a user or set of users. This is particularly relevant in multi-user K8s clusters, where developers should be restricted to working with designated objects, preventing them from accessing others.

While RBAC provides a structured approach to access control, it has limitations in terms of granularity. RBAC policies do not inspect the contents of K8s resource specifications in input to the API. This means that even if the user is restricted to only access specific K8s resources, they can still abuse or exploit all of the features available for that resource. Therefore, a more granular enforcement mechanism capable of inspecting and filtering API requests at a deeper level to block potential misconfigurations and exploits, as discussed in Section III.

### C. K8s Operators and Helm templates

Kubernetes relies on *controllers* to continuously monitor and reconcile the desired and current states of cluster resources. Built-in controllers are preconfigured to handle standard Kubernetes resources and provide basic features, such as autoscaling and self-healing. However, for more complex operations that extend beyond the capabilities of built-in controllers, users must adopt custom controllers. These ad-hoc controllers, known as *Kubernetes Operators* [15], are K8s API clients that automate advanced lifecycle management tasks for stateful and specialized applications. Operators continuously monitor and adjust the application state in a control loop. For instance, if the user specifies that an application should maintain three replicas, the Operator constantly checks the cluster state. If it detects that one replica has failed, it automatically triggers a new deployment to restore the desired count. This capability allows operators to handle both Day-1 (installation, configuration) and Day-2 (updates, scaling, monitoring) operations, reducing manual intervention.

Operators can be implemented in various ways, using the Go language, Ansible, and Helm [16]. Among these, Helm-based

operators are by far the most common. The widespread adoption of Helm is evident in catalogs such as Artifact Hub [17] and OperatorHub [18], which lists hundreds of Helm-based Operators already distributed for production use, spanning applications such as databases, monitoring tools, and CI/CD pipelines.

Helm is the de facto package management solution for K8s [19], to simplify the process of handling complex K8s resources to package, configure, and deploy applications. Helm packages, known as *charts*, provide templates files, that are created by chart developers. These templates provide definitions of K8s resources to run an application. These templates consist of fixed parts and of placeholders for configurable values, as shown in Figure 2. Defaults for the configurable values are typically included in a separate *values* file, in order to provide an initial configuration for the K8s resources. Users of the K8s operator can customize and override to meet specific workload requirements. This flexibility allows users to deploy the same application across different environments with minimal changes [20].

When deploying an application, Helm processes the template by combining them with values to generate complete K8s manifests. Beyond simple value assignment, Helm templates support advanced conditional logic through directives such as *if-else* or *range*. These directives enhance template flexibility, enabling developers to include or exclude fields, iterate over collections, or conditionally populate fields based on the provided values (e.g., enabling optional configurations, as shown in Figure 3). These manifests are then submitted to the K8s API Server to create or update resources. In practice, Helm templates constrain the inputs that are sent to the K8s API Server, since the user does not change the fixed parts of the templates. We leverage this insight to harden the attack surface of the K8s API Server, as discussed in Sec. V.

### III. MOTIVATION

The flexibility and extensibility of K8s, while providing significant advantages for deployment and scaling, also introduce substantial risks. The attack surface exposed by the K8s APIs is particularly concerning, as *malicious API requests* can abuse features and exploit vulnerabilities of K8s. This section discusses these security threats and motivates the need for finer-grain security controls to mitigate them.

```
apiVersion: v1
kind: Secret
metadata:
  name: {{ template "mlflow.fullname" . }}-env-secret
labels:
  dict: Dict
  app: {{ template "mlflow.name" . }}
  chart: {{ template "mlflow.chart" . }}
  release: {{ .Release.Name }}
  heritage: {{ .Release.Service }}
type: Opaque
data:
  dict: Dict
  {{- if .Values.backendStore.postgres.Enabled }}
  PGUSER: {{ .Values.backendStore.postgres.user }}
  PGPASSWORD: {{ .Values.backendStore.postgres.password }}
  {{- end }}
```

Fig. 3. Helm Template for a Secret resource.

```
apiVersion: v1
kind: Pod
spec:
  initContainers:
    - name: busybox
      image: "busybox"
      command: ["ln", "-s", "/", "/mnt/data/symlink-door"]
      volumeMounts:
        - name: test-vol
          mountPath: /test
  containers:
    - name: my-container
      image: "nginx"
      volumeMounts:
        - mountPath: /test
          name: my-volume
          subPath: symlink-door
  volumes:
    - name: my-volume
      emptyDir: {}
```

Fig. 4. Malicious K8s API request triggering CVE-2017-1002101.

#### A. Misconfigurations of a K8s cluster

K8s is not inherently secure by default. Proper configuration is crucial to maintaining a secure environment, but the complexity of this task often leads administrators to prioritize ease of deployment over rigorous security practices. This can result in security misconfigurations that inadvertently weaken the security of the cluster [21]. For example, running containers with elevated privileges or misapplying resource limits can expose critical resources and escalate privileges. Empirical studies [22] have shown that common configuration errors, such as overly permissive network policies or default access settings, can leave clusters vulnerable to exploitation.

Malicious users can leverage these misconfigurations to abuse the cluster. For example, if the cluster runs containers with high privileges, and the user omits the *runAsNonRoot* specification attribute, the user can escalate privileges. Another example is to leave service accounts enabled (e.g., *defaultServiceAccount*), which provides the user with permission to access the K8s API in every namespace. Finally, some functionalities require careful configuration to avoid introducing weaknesses. For example, inadequate TLS/SSL settings can expose communication channels to interception.

These issues often do not stem from flaws in K8s itself, but rather from how the system is configured by system administrators. When misconfigured, a K8s cluster can quickly become an attractive target for attackers. It is also important to note that RBAC does not provide control over potential abuses of such features.

#### B. Software Vulnerabilities in the K8s codebase

Beyond misconfigurations, K8s itself is susceptible to vulnerabilities in its codebase. Numerous CVEs have been found in the recent past, and more are likely to occur in the future due to the complexity of the K8s project. Some of these CVEs can be directly exploited through the K8s API, by injecting malicious input values in object specifications. These exploits can cause disruption of cluster operations, privilege escalation, and unauthorized access to sensitive data.

For example, in K8s clusters prior to version 1.9.4, the vulnerable *subPath* feature can be exploited by attackers to access sensitive directories on the host filesystem (CVE-2017-1002101).

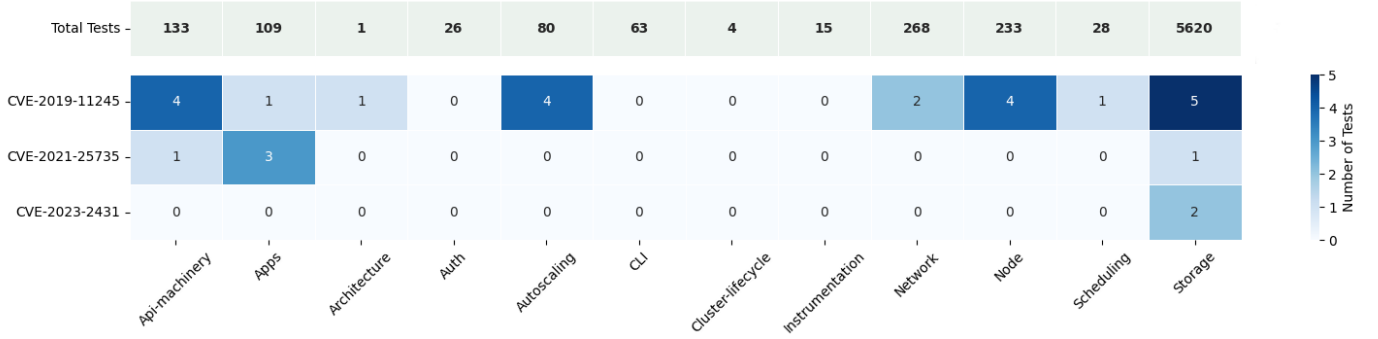


Fig. 5. Number of e2e tests in each category that interact with vulnerable files associated with a CVE.

As illustrated in Figure 4, this vulnerability can be exploited by sending a maliciously crafted API request to deploy a K8s Pod. Specifically, an init container creates a symbolic link to the root directory of the host, and the main container then mounts this symlink as a `subPath`, granting access to the host filesystem.

These vulnerabilities demonstrate that restricting user permissions at a high level, as does RBAC, is insufficient. Limiting access to certain resources (e.g., by denying access to other resources except pods) does not prevent attackers from manipulating specific configuration parameters of unrestricted resources to exploit underlying vulnerabilities. The limitations of RBAC are intrinsically due to its conceptual model, which is coarsely defined around “roles” and “resources”, since it is designed for manual definition and review by administrators. Even if a finer-grained RBAC existed, it would introduce excessive complexity for this use case. Thus, an automated and more detailed filtering of parameters within API requests is necessary to reduce the attack surface.

### C. Attack Surface across Workloads

We have seen that the K8s API is exposed to attacks against K8 misconfigurations and vulnerabilities. We hypothesize that, in practice, many of these vulnerabilities and misconfigurations can be triggered by exploiting specific features of the K8s API that are not required by many users. If this hypothesis holds, it would be possible to prevent attacks by blocking unnecessary features in a workload-specific manner, thus reducing the attack surface of K8s.

To test this hypothesis, we analyzed past vulnerabilities in K8s, and which features can trigger them from the API. We first analyzed the official K8s Vulnerability Database [14], covering all entries from July 2016 to December 2023. This effort yielded a total of 49 CVEs with CVSS scores ranging from 2.6 (low severity) to 9.8 (high criticality). By examining the source code files modified by the patches for these CVEs, we were able to map each vulnerability to the corresponding affected K8s components. These components span a wide range of functionalities, including admissions controllers, kubelet, API server, etcd, kubectrl, scheduler, networking, storage, the legacy cloud provides support and security features. We provide the full mapping in the project repository.

Then, we adopted a set of *workloads* to exercise the features exposed by the K8s API. We chose K8s end-to-end (e2e) tests for this purpose [23]. e2e tests were selected because they perform realistic interactions with the K8s API, by deploying resources and managing configurations, as seen in production

environments. Unlike simple resource operations (e.g., create, delete), these workloads involve complex API requests that selectively trigger different K8s features. For example, an e2e test that manages CustomResourceDefinitions (CRDs) [24] uses service names, ports, and conversion strategies, which exercise specific parts of the K8s codebase. This makes e2e tests ideal as workloads to analyze the relationship between vulnerabilities and features of the K8s API. If a vulnerability can only be triggered through a specific feature, we expect to see that only a small minority of tests cover the vulnerable code.

We selected all available e2e tests across 12 different categories (e.g., networking, storage, scheduling, autoscaling, etc.), excluding tests designed for *Windows* environments because our testbed is built on Linux, and tests in the *disruptive* category, as their focus is on resilience and fault tolerance rather than functional testing. In total, we selected 6,580 e2e tests. It is important to note that the distribution of e2e tests is not uniform across K8s components. This depends on the richness of configuration parameters available for some components (e.g., storage), where the higher number of tests reflects a greater variety of associated workloads. Therefore, we chose not to sample the tests, and to include the full test suites. We need to consider this imbalance when interpreting the results. Before execution, we instrumented the codebase to collect code coverage data, allowing us to track which lines of source code were accessed by each test [25]. We cross-reference these data with the vulnerable files identified earlier.

Figure 5 illustrates the total number of e2e tests grouped by category (columns of the matrix), where each test category interacts with different parts of the K8s API and requires different K8s resources. In addition, the figure shows, as a heatmap, the number of tests that cover vulnerable code linked to 3 CVEs (rows of the matrix). We found that vulnerable code is covered only by a very small minority of workloads. For example, in the case of CVE-2023-2431, only two workloads from the storage e2e tests cover the vulnerable code. The vulnerable code for the other 46 CVEs is not covered by any of the tests, and not shown in the figure for the sake of brevity. In total, only 29 out of 6,580 tests (i.e., less than 0.5%) exercised a vulnerable part of the codebase. Even if the distribution of e2e test across categories is skewed towards storage, which accounts for the majority of tests, the skew does not undermine this conclusion. We find that, even when excluding the largest category, vulnerable code is covered by only 21 out of 960 tests (i.e., around 2%).



In conclusion, we investigate the overlap between the features commonly used by workloads and the ones exploitable by attacks. Since the overlap is small in practice, disabling unnecessary features when executing a particular workload can thwart many attacks. This preliminary analysis shows that by enforcing API access controls tailored to specific workloads, we can significantly limit exposure to (potentially vulnerable) components that are not necessary, thereby reducing the risk of exploitation. This workload-specific filtering approach can thus minimize the K8s attack surface, addressing a critical gap in the existing coarse-grained RBAC model.

#### D. Threat Model

Our threat model assumes attackers who have gained control over the cluster and can execute commands against the K8s API. These attackers include compromised users with stolen credentials, over-privileged users, and other types of insider threats [26]. Such attackers may attempt to escalate privileges to gain full control over K8s resources (e.g., Pods, Deployments, Services), access the underlying hosts in the cluster, and disrupt provided services. To achieve these objectives, they can misuse the API to deploy malicious resources or reconfigure existing ones with harmful *specifications*, in order to leverage cluster misconfigurations or exploit vulnerabilities in the K8s codebase. An example of attack based on this threat model was described in Section III-B.

Other security threats, such as physical attacks on infrastructure (e.g., compromising the physical machine hosting the etcd database) and supply chain attacks (e.g., targeting container images or CI/CD pipelines) are considered out of scope for this work. In addition, we do not consider volumetric denial-of-service attacks, such as API flooding with high-volume request patterns. Our threat model still considers “non-volumetric” DoS that may be caused by malicious API requests from CVE exploits, which can disrupt workloads and cause service unavailability.

To sum up, we discussed how the existing K8s security model based on RBAC is insufficient to prevent abuses of cluster misconfigurations and exploitation of vulnerabilities. The analysis of K8s vulnerabilities showed that they can be exploited only through specific features of the K8s API. Therefore, we design *KubeFence* for fine-grain filtering of K8s API requests to reduce the K8s attack surface.

#### IV. CHALLENGES

Our *KubeFence* solution is based on the fundamental idea of leveraging K8s Operators to obtain strict security policies. K8s operators are becoming more and more popular, and represent a paradigm shift to manage clusters. With operators, developers implicitly encode choices on which features they use. However, the current security model of K8s does not leverage this as an opportunity to restrict the large attack surface. There are technical challenges in doing so.

*KubeFence* generates security policies from Operator configurations, commonly defined using Helm Charts. These charts bring flexibility through templating, which includes conditionals, loops, data types, and user-defined overrides (Sec. II-A). The challenge lies in ensuring that security policies generalize across all valid configurations that can be derived from charts. To address this, *KubeFence* systematically explores the configuration space of an

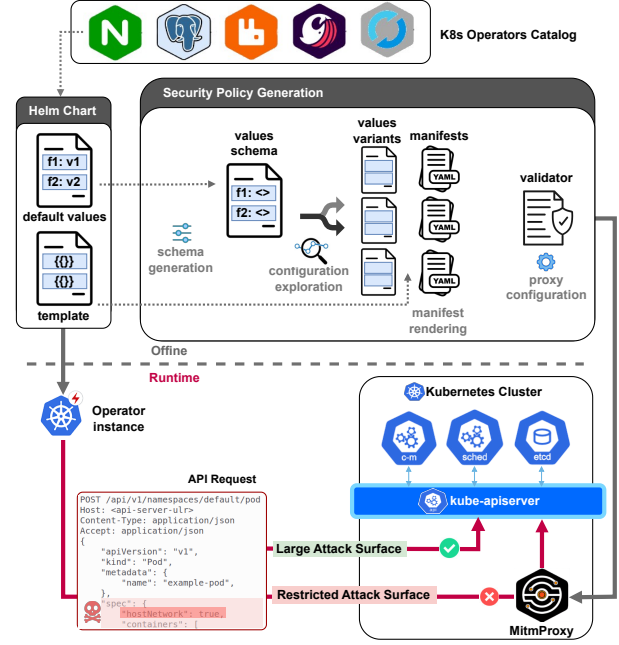


Fig. 6. KubeFence overview.

Operator, by identifying valid variants of the configuration, while restricting them to specific attributes and values where possible.

In addition, K8s API requests contain deeply nested objects with flexible, optional fields, making precise validation challenging. Traditional RBAC only checks K8s resources and actions, whereas fine-grained enforcement requires inspecting the full request structure. A flat-object approach would overlook dependencies between nested fields, enabling attackers to bypass restrictions. To address this, *KubeFence* employs a tree-based validation mechanism that mirrors the hierarchical structure of Kubernetes API requests.

Finally, the architecture of *KubeFence* should fit between clients and the K8s cluster, by ensuring that API requests cannot bypass security validation, with minimal resource and performance overheads.

#### V. KUBEFENCE DESIGN

*KubeFence* is a proxy-based enforcement mechanism designed to automatically generate and enforce fine-grained API security policies, tailored to specific K8s workloads. A *security policy*, in our context, defines allowable resource specifications, which restricts the attack surface by filtering API requests that include unnecessary attributes. These policies are represented as *validators*, a machine-readable format used by our proxy to check API requests. The proposed approach is tailored for Helm-based K8s operators, which are widely used to manage complex Kubernetes configurations and require careful security analysis [19].

Unlike static security policy checkers that only define allowed resource specifications, *KubeFence* enforces these policies dynamically at runtime. By intercepting and validating every API request during cluster operations, *KubeFence* ensures that only authorized

configurations are applied, effectively preventing unauthorized API interactions that could bypass static security measures.

*KubeFence* seamlessly integrates into the existing Kubernetes ecosystem with minimal disruption to the workflow of administrators and developers. Its operation involves three primary steps, as depicted in Figure 6: (1) analyzing K8s workloads and their Helm charts to identify required K8s objects and enumerate their potential configurations; (2) generating security policies based on this analysis and configuring an API proxy to enforce them; (3) intercepting incoming API requests, validating them against the defined policies, and blocking any that deviate from the allowed configurations. By automating policy generation and enforcement, *KubeFence* reduces the manual effort required to secure Kubernetes deployments while enhancing fine-grained protection against insider threats.

#### A. Generation of Security Policies

Writing security policies is a complex and error-prone process, especially for complex systems such as K8s. Inaccurate or incomplete policy definitions can leave clusters vulnerable to overly permissive access control.

To address this challenge, *KubeFence* automates the generation of fine-grained security policies by analyzing Helm charts (default values and templates) as input. The goal of *KubeFence* is to produce a consolidated policy in the form of a *validator*, that is, a reference schema for the validation of incoming API requests. *KubeFence* ensures that the K8s object specification in an API request, i.e., a manifest (see also Fig. 2), complies with the fields and values of the schema. The schema defines all allowable configurations for each K8s resource defined by the Helm chart. An API request that uses an attribute not included in the schema can be blocked, since it is unnecessary according to the Helm chart. Similarly, if the schema assigns a specification attribute with a fixed value, or a value taken from a small set, API requests that use any value outside this defined range can be blocked.

However, there are several aspects that need to be handled for accurate security policies. (1) Conditional logics in Helm charts dynamically vary the structure of the specification based on user-defined values (as in Figure 3). This variability means that many potential configurations generated by these conditions need to be accounted for in the policy. (2) Moreover, while Helm charts often provide default values that fix the structure and content of manifests, users can still override these defaults with custom values (as illustrated in Figure 2). Thus, policies should not rely solely on the templates of the Helm charts but should account for such overrides from the user of the K8s operator. (3) Finally, the K8s specification includes critical attributes that are recommended by security best practices (such as `runAsNonRoot`) but that may be omitted in the Helm charts. Policies must ensure these critical attributes are enforced in API requests, regardless of their presence in the Helm charts.

These aspects make policy generation more nuanced. *KubeFence* manages them by exploring the configuration space represented in the Helm charts, to ensure that policies cover all legitimate API requests, while guarding against potential API misuses. The policy generation process is divided into four phases, detailed below.

**Generation of values schema.** This phase analyzes values of K8s resource specifications in the Helm charts, in order to

# Default Values File	# Values Schema
<code>image:</code>	<code>image:</code>
<code>  registry: docker.io</code>	<code>  registry: docker.io</code>
<code>  repository: bitnami/mlflow</code>	<code>  repository: bitnami/mlflow</code>
<code>pullSecrets:</code>	<code>pullSecrets: [list]</code>
- name: secret-1	
- name: secret-2	
<code>tracking:</code>	<code>tracking</code>
enabled: true	enabled: bool
replicaCount: 1	replicaCount: int
host: "0.0.0.0"	host: IP
containerSecurityContext:	containerSecurityContext:
runAsNonRoot: true	runAsNonRoot: true
<code># postgresql.arch</code>	
<code># standalone or repl</code>	
<code>postgresql:</code>	<code>postgresql:</code>
arch: standalone	arch: standalone, repl

Fig. 7. Schema generation from the *default Values* file used in the MLflow.

identify the domain of every field. The value schema will serve as a basis for exploring the configuration space in the next phase.

*KubeFence* performs a transformation of the default values to produce a structured *values schema*. This transformation aims to: (1) Replace static values with placeholders representing data types or valid ranges, such as `bool`, `string`, `int`, `IP`, `[list]`, and `{dict}`, using regex-based substitution. (2) Replace enumerative fields replaced with lists including all valid options, extracted from annotations in the values file. (3) Lock predefined safe constants to fields critical to security, according to best practices for K8s resource specifications. For example, `securityContext.runAsNonRoot` can be locked to `true` [27]. Similarly, fields like `registry` and `image name` can be restricted to trusted values to mitigate risks like typosquatting attacks [28]. Thus, security-sensitive fields are locked with safe constants rather than placeholders, and any missing critical field is explicitly added. Figure 7 demonstrates an example of this process applied to the MLflow Operator.

**Exploration of the configuration space.** The values schema produced in the previous phase provides a generalized representation of possible configurations. However, it is still not ready for rendering with the Helm template (i.e., processing conditionals, loops, and placeholders in the template). The rendering process requires that only one value is indicated from the configuration space of enumerative fields in the schema. To address this, *KubeFence* performs the rendering multiple times, by exploring different combinations of values in enumerative fields. Each combination leads to a variant of the schema (*values variant*).

At each iteration, *KubeFence* replaces enumerative placeholders in the schema with one of their valid values, while preserving placeholders for non-enumerative fields and constant fields. To avoid combinatorial explosion, *KubeFence* only explores a subset of configurations, such that each valid value for an enumerative field is covered in at least one generated variant. This process guarantees that the union of all variants covers all potential valid values from API requests, which should be allowed in the system by *KubeFence*. More specifically, at each iteration *i*, the process generates a new values variant by replacing each enumerative field with its *i*-th value. If an enumerative list has fewer options than the current iteration index, its last value is reused. The process iterates up to the length of the longest enumerative list. In the example of

the schema in Figure 7 (right), this process generates two values variants, one for each option in the `arch` enumerative field.

**Rendering of manifests.** Once the *values variants* are generated by the previous phase, they need to be translated into Kubernetes manifests. These manifests are concrete representations of resource specifications, by resolving conditionals and loops and using actual values. These manifests will be the basis for generating the final security policies.

Each values variant is combined with the Helm template, using the `helm template` command. This command processes the template and values file to render a manifest. By the end of this phase, *KubeFence* produces a set of Kubernetes manifests, capturing all permissible configurations for the resources required by the K8s operator.

**Generation of validators.** The final step is to consolidate the generated manifests into a single *validator*, a YAML schema that defines all allowable configurations for K8s resources. This validator supports the enforcement of fine-grained security policies, by serving as a reference for validating incoming API requests.

Manifests are grouped based on the resource type (*kind*) (e.g., *Pod*, *Service*, *Deployment*) to ensure the resulting validator is organized and easily navigable. Each group represents the allowed configurations for a specific Kubernetes resource type. Special placeholders (e.g., `string`, `int`, `hostIP`) from the manifests are retained to represent data types, enabling flexibility in configuration validation. Enumerative fields from multiple manifests are consolidated into arrays containing all valid values. Duplicate values are eliminated, while conflicting values are resolved by including all possible options in the array. Fields with constant, security-critical values (e.g., `securityContext.runAsNonRoot: true`) are directly carried over from the manifests without modification. This ensures that security best practices are enforced consistently across all configurations. Figure 8 shows a validator generated merging two manifests.

The proposed approach automates the generation of fine-grained security policies for specific K8s workloads, by generating a YAML policy validator that captures all their allowable configurations, based on the systematic analysis of their Helm charts.

```
# Manifest sample 1
containers:
- name: nginx
  image: nginx:latest
  imagePullPolicy: IfNotPresent
  ports:
  - name: string
    container: IP
# Manifest sample2
containers:
- name: nginx
  image: nginx:latest
  imagePullPolicy: Always
```

```
# Generated Validator
containers:
- name: string
  image: string
  imagePullPolicy:
  - IfNotPresent
  - Always
  ports:
  - name: string
    container: IP
```

Fig. 8. Policy Validator generated from two manifests.

## B. Enforcement of Security Policies

To enforce security policies and apply the generated validators, *KubeFence* employs Mitmproxy [29]. Mitmproxy is chosen

for its capabilities in intercepting, inspecting, and modifying HTTP and HTTPS traffic, with support for SSL/TLS certificates. Mitmproxy is deployed as a Pod on each K8s control-plane node, positioned between the K8s API server and clients (e.g., *kubectl*, *CI/CD pipelines*, or *Operators*). This ensures that all incoming API requests are intercepted and validated against the policy validator before reaching the API server.

In order to maintain a secure enforcement mechanism, the requests to the API server must not bypass the proxy (according to the *Complete Mediation* design principle [11], [12]). To this aim, the API server is restricted to accepting only certificate-based trusted connections, allowing only the proxy with a valid certificate to connect. Furthermore, since client-to-API server connections are encrypted, clients must trust the CA certificate of the proxy to enable traffic interception and decryption for inspection. Thus, proper certificate management is crucial for secure operations.

The core validation mechanism is implemented as a Python-based Mitmproxy plugin, which loads the YAML policy validator. When Mitmproxy intercepts an HTTPS request, the plugin parses the request body to extract the Kubernetes object, including the resource type (*kind*) and its specification fields, for validation. This validation process ensures that only authorized and correctly configured API requests are forwarded to the API server. The validation process operates as a hierarchical comparison, akin to a tree overlap, between the incoming manifest and the policy validator, iterating across the requested K8s resources. The plugin extracts the *kind* field from the request to identify the resource type and verifies its presence in the validator. Then, it ensures that only fields explicitly defined in the validator are present in the manifest and validates that each field's data type matches the expected type specified in the validator. Enumerative fields and security-sensitive fields are validated against their strict list of allowed values in the validator. If the request complies with the validator rules, it is forwarded to the K8s API server unchanged. Otherwise, the plugin blocks the request, returning an HTTP error response to the client. Violations are logged with details of the offending field and the reason for denial, enabling auditing and forensic analysis.

## VI. EXPERIMENTAL ANALYSIS

This section evaluates *KubeFence* across three dimensions. First, we quantify the attack surface exposed by the Kubernetes API and demonstrate how *KubeFence* can reduce unnecessary exposure by restricting access to unused endpoints and fields. Second, using a catalog of misconfiguration-based attacks and CVE exploits targeting specific API fields, we assess the effectiveness of *KubeFence* in mitigating these threats compared to native Kubernetes RBAC. Finally, we analyze the runtime overhead introduced by *KubeFence* for API request validation.

### A. Experimental Setup

We set up a K8s test environment replicating real-world deployment scenarios. The testbed includes a cluster using Kubernetes version 1.28.6, configured with a control-plane node and a worker node deployed on two distinct Ubuntu Linux 22.04.4 virtual machines. Both nodes are allocated 8 vCPUs and 16 GB of RAM, hosted on a machine with an *Intel Xeon E5-1620* 3.70 GHz CPU. *KubeFence* is deployed on the control-plane node, besides the API Server, using a container with Mitmproxy version 10.2.2.



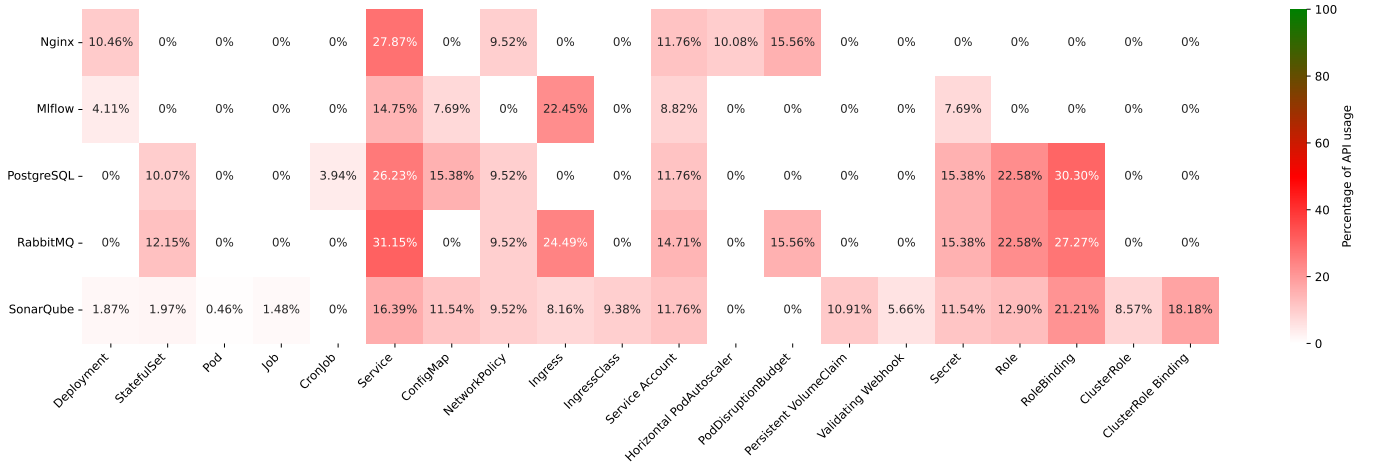


Fig. 9. Percentage of API usage across workloads and endpoints.

This experimental analysis focuses on K8s Operators as a use case to demonstrate the feasibility and effectiveness of fine-grained workload-aware API enforcement. Since Operators are highly configurable, have extensive interactions with the K8s API [20], and are typically deployed using Helm charts, they are a compatible target for validating *KubeFence*. We select five Operators available on the Artifact Hub catalog [17], representing diverse categories of workloads commonly deployed in K8s clusters, including databases (*PostgreSQL* [30]), networking services (*Nginx* [31]), AI/ML applications (*MLflow* [32]), data streaming (*RabbitMQ* [33]), and security (*SonarQube* [34]).

#### B. Quantifying K8s Attack Surface Exposure and Reduction

The K8s API serves as the primary interface to a cluster, which the endpoints for users to query and modify resources. As a result, it represents a critical part of the attack surface for a cluster. We hypothesize that many workloads utilize only a small subset of this API, leaving a significant portion unnecessarily exposed and susceptible to exploitation. Reducing the accessibility of unused or unnecessary API endpoints and fields is a key objective of *KubeFence*. In this experiment, we quantify the attack surface exposed by the Kubernetes API and evaluate how effectively *KubeFence* can reduce it by limiting access to non-essential endpoints and fields.

To quantify the attack surface, we conducted a static analysis of the K8s API. This process involved counting the total number of endpoints exposed by the K8s API Server, and cataloging the configurable fields available for each resource type. Next, we analyzed the selected Kubernetes Operators to understand how real workloads interact with the API endpoints. By examining the validators generated through *KubeFence*, we identified the space of endpoints and fields that can potentially be used by each workload. This analysis provided a detailed understanding of K8s API utilization across different workloads and highlighted workload-specific behavior. The results are summarized in Figure 9, showing the percentage of fields utilized by each workload for each endpoint, relative to the total available fields.

Our findings revealed significant under-utilization of the K8s API by Operators, with numerous fields and endpoints remaining unused in practice. For instance, we observed that certain

resources, such as *Pod* and *Job*, are entirely unused (i.e., 0% usage), by a substantial number of workloads. Other resources, such as *Service* and *ServiceAccount*, are actively used by all workloads, even if many of their fields are left unused. It is still worth blocking these unused fields since they contribute to the attack surface, potentially serving as entry points for exploitation, despite offering no functional value to the workloads. However, these resources cannot be completely disabled, due to the frequent use of some of their fields. This leaves some residual risk of vulnerabilities in these APIs, as discussed in Section VIII.

To evaluate the attack surface reduction achievable by *KubeFence* against RBAC, we analyzed the percentage of fields restrictable by each approach. RBAC restricts access to fields only when the entire resource type (API endpoint) is unused in the heatmap, meaning it lacks the granularity to filter individual fields within an allowed resource. In contrast, *KubeFence* can enforce restrictions on any unused field, even within partially-used endpoints. This makes *KubeFence* a strict superset of RBAC’s enforcement, covering all fields RBAC could restrict while also providing additional reductions in the attack surface.

For the complete set of considered endpoints, we summed up the total configurable fields across all resources. For each workload, we computed the percentage of fields restrictable by RBAC and *KubeFence* as a measure of the attack surface reduction potentially achievable by the two techniques. Table I summarizes the results of this analysis. *KubeFence* consistently achieves a higher reduction in attack surface across all workloads, with improvements of 20.57%, 19.31%, 36.98%,

TABLE I  
ATTACK SURFACE REDUCTION ACHIEVABLE BY KUBEFENCE VS RBAC

Workload	Restrictable Fields		Attack Surface Reduction	
	RBAC	KubeFence	RBAC	KubeFence
Nginx	3747 / 4882	4751 / 4882	76.75 %	97.32 %
MLflow	3883 / 4882	4826 / 4882	79.54 %	98.85 %
PostgreSQL	2906 / 4882	4711 / 4882	59.52 %	96.50 %
RabbitMQ	3676 / 4882	4708 / 4882	75.30 %	96.44 %
SonarQube	1012 / 4882	4772 / 4882	20.73 %	97.75 %



TABLE II  
CATALOG OF K8S MALICIOUS SPECIFICATIONS

ID	Exploit/Misconfiguration	Targeted API Field	Ref.
E1	Activation of hostNetwork (CVE-2020-15257)	hostNetwork	[35]
E2	Abusing LoadBalancer or ExternalIPs (CVE-2020-8554)	externalIPs	[36]
E3	Command injection via volume (CVE-2023-3676) and volumeMounts	containers.volumeMounts.subPath containers.volumes.subPath	[37]
E4	Mount subPath on a file o emptyDir (CVE-2017-1002101)	containers.volumeMounts.subPath	[38]
E5	Absent Resource Limit (CVE-2019-11253)	containers.resources.limits	[39]
E6	Symlink exchange allow host filesystem access (CVE-2021-25741)	container.command	[40]
E7	Bypass of Seccomp Profile (CVE-2023-2431)	containers.securityContext.seccompProfile.localhostProfile	[41]
E8	Privileged Containers (CVE-2021-21334)	containers.securityContext.privileged	[42]
M1	Activation of hostIPC	hostIPC	[21]
M2	Activation of hostPID	hostPID	[21]
M3	Use Readonly Filesystem	containers.securityContext.readOnlyRootFilesystem	[21]
M4	Running Containers as Root	containers.securityContext.runAsNonRoot containers.securityContext.runAsRootAllowed	[21]
M5	Allow Dangerous Capabilities to Containers	containers.securityContext.capabilities.add	[21]
M6	Escalated Privileges for Child Container Processes	containers.securityContext.allowPrivilegeEscalation	[21]
M7	Custom SELinux user or role	containers.securityContext.seLinuxOptions.user containers.securityContext.seLinuxOptions.role	[21]

21.14%, and 77.02% across the five workloads, averaging 35% compared to RBAC. These results highlight that RBAC achieves lower attack surface reduction for workloads requiring multiple endpoints, as it cannot blacklist partially-used resources. In contrast, *KubeFence* can restrict unused fields within utilized endpoints, providing finer-grained protection.

### C. Catalog of Malicious Specifications

As described in Section III, malicious API requests pose critical security risks, by exploiting specific fields to achieve privilege escalation, unauthorized access to critical resources, and misconfigurations that may lead to degradation of cluster availability or reliability.

To evaluate *KubeFence* against these threats, we built a catalog of 15 malicious specifications, comprising 7 misconfigurations and 8 malicious fields used by CVE exploits. These malicious specifications inject malicious values in Kubernetes manifests that can expose vulnerabilities or enable unsafe configurations, making them a practical subset for testing the effectiveness of *KubeFence* in mitigating attacks. Table II summarizes this catalog, identifying the targeted fields and providing references to their sources.

This catalog was developed by analyzing prior research [22], security blogs [43]–[46], CVE disclosure [47], and Kubernetes security guidelines [21], [27]. Examples include enabling the `hostNetwork` field for host network sharing (CVE-2020-15257), exploiting `subPath` for host directory access (CVE-2017-1002101), and bypassing security profiles (CVE-2023-2431). We focus on CVEs from Section III-C that are exposed to malicious specifications from the K8s API interface, as these align with our threat model and the scope of API-level enforcement. We exclude CVEs that are unable to reproduce due to strict environmental prerequisites, which fall outside our experimental setup. For instance, Kubernetes clusters are affected by CVE-2023-5528 only if they use an in-tree storage plugin for Windows nodes.

### D. KubeFence Effectiveness against RBAC

Misconfigurations and CVE exploits pose significant security risks in K8s (see Section III). The native Kubernetes RBAC

```
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
        - name: nginx
          image: testImage
          securityContext:
            runAsNonRoot: false
```

Fig. 10. Example of a malicious YAML manifest

mechanism provides access control at the resource- and verb-level, but lacks the granularity to restrict individual fields within the resource specification. This experiment measures the effectiveness of *KubeFence* compared to RBAC, by evaluating its ability to *mitigate CVEs and misconfigurations*. To quantify this, we generate workload-specific policies for the five selected operators (listed in Section VI-A), and test whether *KubeFence* can block API-based misconfigurations and CVE exploits.

To test the enforcement mechanisms, we generated malicious API requests using our catalog of malicious specifications (Table II). We inject the malicious fields in the catalog in resource types that support that malicious fields. For instance, the `spec.externalIPs` field is specific to *Service* resources. Other fields apply to relevant K8s resources, such as to *Pod* and higher-level abstractions like *Deployment*, *ReplicaSet*, *StatefulSet*, and *DaemonSet*.

Legitimate resource configurations were retrieved from Operator manifests, and malicious fields were injected into this configuration to create 15 distinct malicious manifests for each operator. For example, Figure 10 shows how the misconfigured `runAsNonRoot` field is injected into a *Deployment* resource from the Nginx Operator.

These malicious manifests were then submitted to the K8s API while the respective workload-specific RBAC or *KubeFence* policy was in place. This process simulates realistic attack scenarios where a malicious client attempts to exploit CVEs or misconfigurations through the K8s API interface. The effectiveness of each enforcement mechanism was measured by recording whether

each CVE exploit or misconfiguration attempt was mitigated.

**Native K8s RBAC setup.** We evaluated RBAC by configuring the K8s cluster with audit logging enabled, to capture API requests during the execution of an attack-free workload. Audit logs keep track of accesses to *API endpoint*, the *resource type* (e.g., Pods, Services), *verb* (e.g., get, create, update, delete), and the resource specification in API requests. Then, the audit logs were processed with the `audit2rbac` tool [48], which infers the minimum permissions required for a workload. This process generated five distinct YAML files, representing a tailored RBAC policy for each operator, based on the observed API interactions. Malicious manifests were applied to the cluster with RBAC policies in place. For each attack, we recorded the success or failure of the API request.

**KubeFence setup.** Then, we evaluated *KubeFence* by generating fine-grained security policies tailored for each workload, by analyzing the configurations required by the workloads. These policies were enforced using our proxy placed between the clients and the K8s API Server.

The same malicious manifests used for testing RBAC were applied against the *KubeFence* proxy. Each interaction of the Operators with the API server was intercepted by the proxy, which validated the API request against the workload-specific policy (i.e., the validator). Moreover, our logs report the denied actions, and the malicious fields that triggered by filtering.

**Experimental Results.** Table III reports the mitigated CVEs and misconfigurations by RBAC and *KubeFence*, respectively. While RBAC did not block any of the attacks, *KubeFence* successfully blocked all of them.

The results highlight that RBAC policies, even when tailored to workloads using tools like *audit2rbac*, lack the granularity to enforce restrictions on individual fields within resource specifications. Figure 11 illustrates an audit entry recorded during the deployment of the MLflow Operator, logging the creation of a *Deployment* resource. The generated RBAC policy effectively defined access at the resource level, specifying resource *kind*, *namespace*, *API group*, and allowed *verbs*. However, it omitted critical parameter-level details, such as *spec* fields “available” in the audit logs. This omission is not a limitation of *audit2rbac*, but rather an inherent limitation of RBAC policies, which do not allow specification at this level of detail. As a result, RBAC failed to prevent attacks that exploit features not needed by the operators, such as enabling *hostNetwork* or disabling *runAsNonRoot*.

By contrast, *KubeFence* successfully enforced fine-grained controls, blocking all attacks to misconfigurations and CVEs. For instance, it denied requests abusing the *subPath* field, as this parameter was not part of the configuration space defined in the Helm charts of the Operators. Furthermore, legitimate workload actions were unaffected, demonstrating the precision and reliability of *KubeFence* in blocking unauthorized API request parameters without disrupting normal operations.

### E. KubeFence Overhead

This section evaluates the runtime overhead introduced by *KubeFence* compared to the native K8s RBAC. The focus is on the online phase, where API requests are inspected and forwarded to the API server, as this directly impact cluster operations. The offline phase of *KubeFence*, which involves

TABLE III  
MITIGATED CVEs AND MISCONFIGURATIONS BY RBAC AND KUBEFENCE.

Workload	CVEs		Misconfigurations	
	RBAC	KubeFence	RBAC	KubeFence
PostgreSQL	0	8	0	7
Nginx	0	8	0	7
MLflow	0	8	0	7
RabbitMQ	0	8	0	7
SonarQube	0	8	0	7

learning security policies, is excluded from this analysis as it does not affect runtime performance.

We measured the round-trip-time (RTT) latency for processing the full set of API requests generated during the deployment of the five selected operators. These requests are issued by the `kubectl apply` command from the client, and include all interactions with the API server to configure the resources defined by the Operator. The RTT latency was defined as the total elapsed time from issuing the *apply* command until the client received a response, indicating the API server had finished processing the requests. This experiment was conducted under two scenarios: first with native RBAC, and then with *KubeFence*. All requests were benign, as this experiment focused on normal operations rather than attack scenarios. To ensure statistical significance, we repeated the process 10 times per workload and computed the average latency and standard deviation for both RBAC and *KubeFence*. In addition, using the same workload, we evaluate the impact of *KubeFence* on system resources. To this end, we measured the CPU and memory usage of the proxy container, reporting the average and standard deviation over 10 repetitions.

Table IV presents the average latencies and standard deviations for each workload, as well as the increase in latency introduced by *KubeFence* over native RBAC.

The results show that the additional latency introduced by *KubeFence* remains minimal, with absolute increases ranging from 0.0266 s to 0.0846 s. Even in the worst case, where the overhead reaches 26%, the total RTT latency remains well below 0.5 seconds, which is negligible for cluster management tasks such as workload deployment [49], [50]. Furthermore, this overhead impacts only operations initiated by external actors interacting with the K8s control plane, such as managing deployments and querying resource modifications, while internal API interactions by K8s components remain unaffected by *KubeFence*. Moreover, the application themselves (e.g., web resources served by Nginx) are unaffected. This minimal overhead is a worthwhile trade-off considering the enhanced attack mitigation capabilities and the attack surface reduction provided by *KubeFence*.

TABLE IV  
RBAC VS KUBEFENCE AVERAGE REQUEST LATENCY

Operators	RBAC RTT (ms)	KubeFence RTT (ms)	Increase (ms, %)
MLflow	211.0±39.2	237.6±37.5	+26.6 (12.61%)
Nginx	168.4±25.7	210.4±26.7	+42.0 (24.94%)
PostgreSQL	178.1±16.1	213.6±13.0	+35.5 (19.93%)
RabbitMQ	242.9±16.6	307.6±23.4	+64.7 (26.64%)
SonarQube	385.9±14.0	470.5±35.0	+84.6 (21.92%)



Fig. 11. RBAC policy generated (on the right) from an audited *create deployment* operation (on the left).

In addition, the impact of *KubeFence* on system resources was minimal. CPU usage increased only by 1.21% ( $\pm 0.04$ ), and memory consumption increased by 85.54 MiB ( $\pm 0.25$ ), which is a negligible overhead considering the security benefits.

The experimental results demonstrate that *KubeFence* effectively enhances Kubernetes security. It achieves significant attack surface reduction by restricting unused API fields and endpoints, addressing gaps in native RBAC that cannot enforce such fine-grained control. Moreover, *KubeFence* successfully blocks all tested misconfigurations and CVE exploits by precisely validating API requests against workload-specific policies, ensuring robust protection against real-world threats. Despite introducing a small latency overhead during API request validation, the impact remains reasonable for operations initiated by external actors and does not affect internal K8s operations. These findings establish *KubeFence* as a practical and efficient solution for securing K8s environments.

## VII. RELATED WORK

### A. Static Analysis

Kubernetes is widely adopted for its scalability and automation capabilities. However, its extensive configurability introduces significant risks of misconfiguration. Static analysis tools and methodologies have been developed to address these issues, focusing on pre-deployment detection in Kubernetes manifests and Helm Charts.

Empirical studies [22], [51], [52] highlight the prevalence of Kubernetes misconfiguration and their security implications. Research analyzing Helm charts and Operators [19], [53] identifies insecure defaults and outdated dependencies that elevate security risks. Broader efforts [21], [54] emphasize adherence to best practices, including RBAC or network segmentation, but these approaches rely on manual intervention and lack runtime adaptability.

Static analysis tools, such as KubeLinter [55], Polaris [56], Checkov [57], KICS [58], and SLI-Kube [22] identify misconfigurations using predefined rules. Graph-based methods such as KGSecConfig [59] and [60], [61] automate secure cluster configuration or provide security assessment of configurations. Generative approaches like GenKubeSec [62] and [63], [64] leverage large language models to identify and remediate misconfigurations. Despite their utility, these tools operate pre-deployment, leaving systems exposed to runtime threats, such as malicious API requests targeting unused or overly permissive fields.

RBAC misconfiguration detection tools, such as EPSCan [65], focus on identifying excessive permissions but lack mechanisms

to enforce fine-grained control at runtime. These approaches are inadequate for addressing threats where unused API fields can be exploited.

In contrast, *KubeFence* bridges this gap by dynamically enforcing workload-specific API policies at runtime, reducing the attack surface through fine-grained control of resource specifications.

### B. Container Security

Container-centric security solutions primarily address runtime behaviors of containers, targeting process execution and interactions within the host environment.

Runtime monitoring tools, such as system call monitoring [66], detect anomalous container behavior but require extensive pre-configuration and rely on heuristic models. ProSPEC [67] predicts potential breaches through proactive policy enforcement. Tools like Kub-Sec [68] and KubeArmor [69] generate security profiles for pods, enforcing the principle of least privilege. Sysdig Falco [70] and similar tools monitor container operations based on pre-defined rules. Tools like eBPF [71] and Seccomp [72] can be used to whitelist allowable syscalls from containers. However, these approaches focus on behaviors occurring within workload containers, rather than K8s API requests. In principle, syscall whitelisting can detect some of the malicious behaviors that may occur after exploiting K8s vulnerabilities, e.g., executing privileged system calls. However, it is quite challenging to define policies for finer-grain filters on system calls, since it would need complex static/dynamic analysis of programs, or manual definitions. Moreover, some malicious behaviors can still escape system call policies, such as by abusing privileges that are available to the application.

*KubeFence* shifts the focus to API-level security, complementing container security solutions. By dynamically generating fine-grained policies tailored to workload configurations, it secures the Kubernetes control plane against threats arising from unused API fields and misconfigurations, offering comprehensive protection for the Kubernetes ecosystem.

### C. REST API Security

Securing REST APIs have been a key focus in web and application security, with numerous methodologies proposed for generating and enforcing access control policies to mitigate unauthorized access and enhance runtime security. Languages and frameworks like RestPL [73] facilitate precise and flexible policy definitions for RESTful APIs, emphasizing request-level granularity but remaining confined to static pre-deployment configurations. Similarly, Jayathilaka et al. [74] propose a framework for enforcing API security policies in cloud platforms, which ensures backward compatibility and enforces

best practices during API deployment. Atlidakis et al. [75] extend REST API security through property checking, fuzzing, and runtime monitoring, while a more recent framework by Khan et al. [76] focuses on detecting and mitigating vulnerabilities in REST APIs by integrating reverse proxy techniques to identify and prevent attacks like SQL injection and XSS in real-time.

While effective for traditional REST APIs, these solutions lack mechanisms for dynamically adapting policies on the rich configurations and nested specifications unique to Kubernetes APIs. In contrast, *KubeFence* extends these principles by generating fine-grained security policies tailored to Kubernetes workloads.

## VIII. DISCUSSION

**Extensibility beyond Helm.** The current implementation of *KubeFence* focuses on Helm-based workloads, using Helm templates to generate API security policies. While effective, this limitations its applicability to Helm deployments. However, the methodology of analyzing manifests to derive workload-specific security policies can be easily extended to other deployment mechanisms, such as Kustomize or raw YAML manifests. By adapting the parsing and policy generation processes, *KubeFence* can ensure consistent security enforcement across diverse deployment workflows.

**Scope of attack surface hardening.** The primary objective of *KubeFence* is to *reduce the Kubernetes attack surface*, denying unnecessary and risky features on a per-workload basis. Despite this, it does not claim to eliminate CVE exploitability or misconfiguration risks entirely. The solution leverages the client configuration space to infer which API endpoints and fields allow, and best practices guidelines to infer which critical field to lock to safe values. This significantly mitigates opportunities for attackers to exploit malicious configurations irrelevant to the workload.

*KubeFence* is based on the idea of blocking code not used by common workloads, which can be difficult to apply for some users. It is possible that uncommon, but legitimate workloads are blocked by such restrictive security policy. In general, false positives are a challenge for any filtering approach, as the same issue is faced by admins that manage firewalls, IDS, and similar tools. *KubeFence* mitigates false positives by tailoring policies to K8s operators, which are becoming a popular approach to manage clusters. In such use cases, features can be restricted with very high accuracy. However, in other use cases, it may be difficult to anticipate which features should be allowed. Still, some enterprises may still want to block uncommon workloads, and enable them only after more careful scrutiny.

It is also possible that *KubeFence* does not restrict interfaces that are prone to vulnerabilities, in the case that these interfaces are used by legitimate workloads, in order not to disrupt them. These interfaces represent a residual risk, which has to be handled through other complementary strategies. One approach is to adopt anomaly detection methods on API calls, which can identify misuses and exploitation attempts of the features [77]. Another strategy is to perform more thorough testing, such as fuzzing [78], to identify vulnerabilities in the residual attack surface.

*KubeFence* does not validate the functional correctness of Helm charts when they introduce unnecessary features or omit required ones. Instead, it enforces the stated resource definitions as provided. Ensuring correctness in such cases is an orthogonal problem and fall outside the scope of *KubeFence*. External

YAML validation tools (e.g., KubeLinter [55], Checkov [57]) can be used before policy generation to address this problem.

Furthermore, *KubeFence* does not address risks arising from compromises in the Kubernetes Operator catalog through supply chain attacks, where malicious Operators could inject unsafe configurations. In such cases, the responsibility for addressing these risks lies with workload developers.

**Performance Optimizations.** While the overhead introduced by *KubeFence* is negligible for most cluster management tasks, with latency increases ranging from 0.0266 to 0.0846 s, it could still impact performance-critical of real-time deployment scenarios. The current implementation relies on a proxy Pod to intercept, validate, and forward external API requests to the API server, which adds network latencies. To address this, *KubeFence* could be integrated directly into the API server, eliminating the forwarding delays and leaving only the validation cost. Although it requires modifications to the API server codebase, this approach offers a viable path to optimize enforcement efficiency for more demanding use cases.

## IX. CONCLUSION

The extensive Kubernetes API and reliance on coarse-grained RBAC leave clusters vulnerable to misconfigurations and CVE exploits. This paper introduced *KubeFence*, a proxy-based enforcement mechanism that enhances Kubernetes security by automatically generating and enforcing fine-grained API security policies tailored to workloads, effectively reducing the attack surface and mitigating insider threats.

## ACKNOWLEDGMENT

We are grateful to our shepherd François Taïani and to the anonymous reviewers for their feedback. This work was supported by the projects GENIO (CUP B69J23005770005) funded by MIMIT, and “IDA—Information Disorder Awareness” funded by the European Union-Next Generation EU within the SERICS Program through the MUR National Recovery and Resilience Plan under Grant PE00000014.

## REFERENCES

- [1] Kubernetes project, “Kubernetes,” <https://kubernetes.io/>, 2024.
- [2] Cloud Native Computing Foundation, “Annual survey,” <https://www.cncf.io/reports/cncf-annual-survey-2023/>, 2024.
- [3] Kubernetes project, “Kubernetes User Case Studies,” <https://kubernetes.io/case-studies/>, 2024.
- [4] BlackDuck Open Hub, “Kubernetes,” <https://openhub.net/p/kubernetes>, 2024.
- [5] C. Theisen, N. Munaiah, M. Al-Zyoud, J. C. Carver, A. Meneely, and L. Williams, “Attack surface definitions: A systematic literature review,” *Information and Software Technology*, vol. 104, pp. 94–103, 2018.
- [6] Benjamin Grap, and Manoj Ahuje, “CrowdStrike discovers first ever dero cryptojacking campaign targeting Kubernetes,” <https://www.crowdstrike.com/en-us/blog/crowdstrike-discovers-first-ever-dero-cryptojacking-campaign-targeting-kubernetes/>, 2024.
- [7] N. Yang, W. Shen, J. Li, X. Liu, X. Guo, and J. Ma, “Take over the whole cluster: Attacking Kubernetes via excessive permissions of third-party applications,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023, pp. 3048–3062.
- [8] US Cybersecurity and Infrastructure Security Agency, “Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software,” <https://www.cisa.gov/resources-tools/resources/secure-by-design>, CISA, Tech. Rep., 2023.
- [9] UK National Cyber Security Centre (NCSC), “Secure design principles: Guides for the design of cyber secure systems,” <https://www.ncsc.gov.uk/collection/cyber-security-design-principles>, NCSC, Tech. Rep., 2020.



- [10] European Commission, “EU Cyber Resilience Act,” <https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act>, 2024.
- [11] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.
- [12] R. E. Smith, “A contemporary look at Saltzer and Schroeder’s 1975 design principles,” *IEEE Security & Privacy*, vol. 10, no. 6, pp. 20–25, 2012.
- [13] Kubernetes, “Using RBAC Authorization,” <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>, 2024.
- [14] Kubernetes, “Official CVE Feed,” <https://kubernetes.io/docs/reference/issues-security/official-cve-feed>, 2024.
- [15] Kubernetes, “Operator Pattern,” <https://kubernetes.io/docs/concepts/extend-kubernetes/operator/>, 2024.
- [16] A. Handy, “Build Your Kubernetes Operator with the Right Tool,” <https://www.redhat.com/en/blog/build-your-kubernetes-operator-with-the-right-tool>, 2021.
- [17] Cloud Native Computing Foundation, “Artifact Hub,” <https://artifacthub.io/>, 2024.
- [18] Red Hat, “OperatorHub,” <https://operatorhub.io/>, 2024.
- [19] A. Zerouali, R. Opdebeeck, and C. De Roover, “Helm charts for Kubernetes applications: Evolution, outdatedness and security risks,” in *IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 523–533.
- [20] S. Henning, B. Wetzel, and W. Hasselbring, “Reproducible benchmarking of cloud-native applications with the kubernetes operator pattern,” 2021.
- [21] NSA, CISA, “Kubernetes Hardening Guide,” <https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/2716980/nsa-cisa-release-kubernetes-hardening-guidance/>, NSA, CISA, Tech. Rep., 2022.
- [22] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, “Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study,” *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, May 2023. [Online]. Available: <https://doi.org/10.1145/3579639>
- [23] Kubernetes project, “Kubernetes e2e tests,” <https://github.com/kubernetes/kubernetes/tree/master/test/e2e>, 2024.
- [24] —, “CRD Conversion Webhook e2e test,” [https://github.com/kubernetes/kubernetes/blob/master/test/e2e/apimachinery/crd\\_conversion\\_webhook.go](https://github.com/kubernetes/kubernetes/blob/master/test/e2e/apimachinery/crd_conversion_webhook.go), 2024.
- [25] M. Gasch, “Go 1.20 Coverage Profiling Support for Kubernetes Apps,” <https://www.mgasch.com/2023/02/go-e2e/>, 2024.
- [26] CloudSecDocs, “Kubernetes Threat Model,” [https://cloudsecdocs.com/containers/theory/threats/k8s\\_threat\\_model/#threat-actors](https://cloudsecdocs.com/containers/theory/threats/k8s_threat_model/#threat-actors), 2024.
- [27] Kubernetes project, “Pod security standards,” <https://kubernetes.io/docs/concepts/security/pod-security-standards>, 2024.
- [28] G. Liu, X. Gao, H. Wang, and K. Sun, “Exploring the uncharted space of container registry typosquatting,” in *31st USENIX Security Symposium (USENIX Security)*, 2022, pp. 35–51.
- [29] A. Cortesi, M. Hils, and T. Kriechbaumer, “mitmproxy,” <https://mitmproxy.org/>, 2024.
- [30] Artifact Hub, “PostgreSQL Operator,” <https://artifacthub.io/packages/helm/bitnami/postgresql>, 2024.
- [31] —, “Nginx Operator,” <https://artifacthub.io/packages/helm/bitnami/nginx>, 2024.
- [32] —, “MLflow Operator,” <https://artifacthub.io/packages/helm/community-charts/mlflow>, 2024.
- [33] —, “RabbitMQ Operator,” <https://artifacthub.io/packages/helm/bitnami/rabbitmq>, 2024.
- [34] —, “SonarQube Operator,” <https://artifacthub.io/packages/helm/openshift-bootstraps/sonarqube>, 2024.
- [35] NVD, “CVE-2020-15257,” <https://nvd.nist.gov/vuln/detail/cve-2020-15257>, 2024.
- [36] —, “CVE-2020-8554,” <https://nvd.nist.gov/vuln/detail/cve-2020-8554>, 2024.
- [37] —, “CVE-2023-3676,” <https://nvd.nist.gov/vuln/detail/cve-2023-3676>, 2024.
- [38] —, “CVE-2017-1002101,” <https://nvd.nist.gov/vuln/detail/cve-2017-1002101>, 2024.
- [39] —, “CVE-2019-11253,” <https://nvd.nist.gov/vuln/detail/cve-2019-11253>, 2024.
- [40] —, “CVE-2021-25741,” <https://nvd.nist.gov/vuln/detail/cve-2021-25741>, 2024.
- [41] —, “CVE-2023-2431,” <https://nvd.nist.gov/vuln/detail/cve-2023-2431>, 2024.
- [42] —, “CVE-2021-21334,” <https://nvd.nist.gov/vuln/detail/cve-2021-21334>, 2024.
- [43] A. Suda, “Don’t use net=host,” <https://medium.com/nttlabs/dont-use-host-network-namespaces-f548aeeef575>, 2020.
- [44] Kubernetes blog, “Fixing the Subpath Volume Vulnerability in Kubernetes,” <https://kubernetes.io/blog/2018/04/04/fixing-subpath-volume-vulnerability>, 2018.
- [45] Akamai, “Can’t Be Contained: Finding a Command Injection Vulnerability in Kubernetes,” <https://www.akamai.com/blog/security-research/kubernetes-critical-vulnerability-command-injection>, 2023.
- [46] Palo Alto Networks, “Protecting Against an Unfixed Kubernetes MITM Vulnerability,” <https://unit42.paloaltonetworks.com/cve-2020-8554/>, 2020.
- [47] GitHub, “GitHub Advisory Database,” <https://github.com/advisories>, 2024.
- [48] J. Liggitt, “audit2rbac,” <https://github.com/liggitt/audit2rbac>, 2024.
- [49] M. Barletta, L. De Simone, R. D. Corte, and C. Di Martino, “Failover timing analysis in orchestrating container-based critical applications,” in *19th European Dependable Computing Conference (EDCC)*, 2024, pp. 81–84.
- [50] Kubernetes, “Kubernetes scalability and performance SLIs/SLOs,” <https://github.com/kubernetes/community/blob/master/sig-scalability/slos/slos.md>, 2024.
- [51] S. I. Shamim, “Mitigating security attacks in Kubernetes manifests for security best practices violation,” in *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 1689–1690.
- [52] D. B. Bose, A. Rahman, and S. I. Shamim, “‘under-reported’ security defects in Kubernetes manifests,” in *IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, 2021, pp. 9–12.
- [53] Q. Xu, Y. Gao, and J. Wei, “An Empirical Study on Kubernetes Operator Bugs,” in *33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1746–1758.
- [54] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, “XI Commandments of Kubernetes Security: A Systematization of Knowledge related to Kubernetes Security Practices,” *IEEE Secure Development (SecDev)*, pp. 58–64, 2020.
- [55] StackRox, “KubeLint,” <https://kubelinter.io/>, 2024.
- [56] Fairwinds, “Polaris,” <https://www.fairwinds.com/polaris>, 2024.
- [57] Prisma Cloud, “Checkov,” <https://www.checkov.io/>, 2024.
- [58] Checkmarx, “KICS,” <https://kics.io>, 2024.
- [59] M. U. Haque, M. M. Kholoosi, and M. A. Babar, “KGSecConfig: A Knowledge Graph Based Approach for Secured Container Orchestrator Configuration,” in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 420–431.
- [60] A. Blaise and F. Rebecchi, “Stay at the Helm: secure Kubernetes deployments via graph generation and attack reconstruction,” in *IEEE 15th International Conference on Cloud Computing (CLOUD)*, 2022, pp. 59–69.
- [61] G. Dell’Immagine, J. Soldani, and A. Brogi, “KubeHound: Detecting Microservices’ Security Smells in Kubernetes Deployments,” *Future Internet*, vol. 15, no. 7, 2023.
- [62] E. Malul, Y. Meidan, D. Mimiran, Y. Elovici, and A. Shabtai, “GenKubeSec: LLM-Based Kubernetes Misconfiguration Detection, Localization, Reasoning, and Remediation,” *arXiv preprint arXiv:2405.19954*, 2024.
- [63] F. Minna, F. Massacci, and K. Tuma, “Analyzing and Mitigating (with LLMs) the Security Misconfigurations of Helm Charts from Artifact Hub,” *arXiv preprint arXiv:2403.09537*, 2024.
- [64] G. Lanciano, M. Stein, V. Hilt, T. Cucinotta *et al.*, “Analyzing Declarative Deployment Code with Large Language Models,” *CLOSER*, vol. 2023, pp. 289–296, 2023.
- [65] Y. Gu, X. Tan, Y. Zhang, S. Gao, and M. Yang, “EPScan: Automated Detection of Excessive RBAC Permissions in Kubernetes Applications,” in *IEEE Symposium on Security and Privacy (SP)*, 2025.
- [66] H. Kitahara, K. Gajananan, and Y. Watanabe, “Highly-scalable container integrity monitoring for large-scale Kubernetes cluster,” in *IEEE International Conference on Big Data (Big Data)*, 2020, pp. 449–454.
- [67] H. Kermabon-Bobinnec, M. Gholipourchoubeh, S. Bagheri, S. Majumdar, Y. Jarraya, M. Pourzandi, and L. Wang, “Prospec: Proactive security policy enforcement for containers,” in *12th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2022, pp. 155–166.
- [68] H. Zhu and C. Gehrmann, “Kub-Sec, an automatic Kubernetes cluster AppArmor profile generation engine,” in *14th International Conference on Communication Systems & NETWORKS (COMSNETS)*, 2022, pp. 129–137.
- [69] Cloud Native Community Group, “KubeArmor,” <https://kubearmor.io>, 2024.
- [70] Sysdig, “Falco,” <https://sysdig.com/opensource/falco/>, 2024.
- [71] eBPF.io authors, “eBPF,” <https://ebpf.io/>, 2025.
- [72] J. Edge, “A seccomp overview,” <https://lwn.net/Articles/656307/>, 2015.
- [73] Y. Luo, H. Zhou, Q. Shen, A. Ruan, and Z. Wu, “Restpl: Towards a request-oriented policy language for arbitrary restful apis,” in *IEEE International Conference on Web Services (ICWS)*, 2016, pp. 666–671.
- [74] H. Jayatilaka, C. Krintz, and R. Wolski, “Rest web service maintenance through api policy enforcement,” UC Santa Barbara tech report, Tech. Rep., 2014.

- [75] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Checking security properties of cloud service rest apis," in *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 387–397.
- [76] M. S. Khan, R. S. F. Siam, and M. A. Adnan, "A framework for checking and mitigating the security vulnerabilities of cloud service restful apis," *Service Oriented Computing and Applications*, pp. 1–22, 2024.
- [77] Kubescape, "Behavioral Cloud Application Detection & Response," <https://www.armosec.io/platform/cloud-detection-and-response/>, 2024.
- [78] P. V. Dommaraju, "Erroneous Kubernetes Object Generation using Structure-aware Fuzzing," Ph.D. dissertation, Universiteit van Amsterdam, 2024.