Towards A Capability Model of Kubernetes Runtime Security Enforcement Mechanisms

Towards A Capability Model of Kubernetes Runtime Security Enforcement Mechanisms

Francesco Settanni[1][0009−0007−2342−6281],
Giuseppe Lisena[1][0009−0007−7489−0850], and
Cataldo Basile[1][0000−0002−8016−1490]

Politecnico di Torino, Torino, Italy
{name.surname}@polito.it

Abstract. The shift toward cloud-native and microservice-based architectures has made Kubernetes the de facto platform for managing containerized applications. However, its limited native support for security features has led to the proliferation of diverse enforcement mechanisms, such as Cilium, Calico, Tetragon, and KubeArmor. These tools vary in capabilities and configuration, complicating the establishment of an effective security posture. This work proposes a conceptual model that abstracts runtime security enforcement across these tools, enabling intent-based security policy design and automation. We present a model-driven approach to bridge high-level security requirements with low-level enforcement configurations. Our approach facilitates cloud portability, simplifies policy refinement, and enhances security consistency for heterogeneous environments. Validation across real-world microservice architectures and security policy catalogs demonstrates its practicality and effectiveness.

Keywords: Kubernetes · security enforcement · cloud security · network policy · Cilium · Calico · Tetragon · KubeArmor

1 Introduction

The approach to security in modern network environments and infrastructures is a continuously evolving discipline. With the advent of cloud-native architectures and increasingly distributed hybrid and multi-cloud environments, new paradigms have been devised to counteract emerging attack methodologies and risks inherent to these architectures. Pursuing zero-trust capabilities in such environments has driven significant innovation in technologies and approaches for enforcing security, particularly since traditional perimeter-based defenses are ill-suited to their highly distributed nature. This shift has necessitated relocating security policy enforcement closer to the core operational units. Trust zones are no longer bastion-based but revolve around the smallest software units in which microservices are deployed, namely, the pods [11].

Kubernetes (K8s) has emerged as one of the most widely used microservice orchestrators. It effectively manages the dynamic nature of microservice-based cloud architectures, ensuring agility and scalability in complex deployments.

While K8s is valued for its ability to manage software resources effectively, it also introduces significant challenges for developers and engineers. Many aspects of K8s environments remain the responsibility of end users, necessitating the design and deployment of extension mechanisms on top of vanilla installations. Although these mechanisms provide useful features, they often introduce additional complexities exacerbated by the myriad of overlapping solutions, each with its own unique characteristics.

This is particularly evident in runtime security enforcement, with the market not yet mature and the adoption of effective solutions stalling. K8s provides limited native support for security enforcement at runtime, especially regarding network security. Specifically, it declares the APIs for basic network restriction functionalities but leaves their implementation to the underlying network infrastructure provider. As a result, a diverse ecosystem of tools, such as CNI (Container Network Interface) plugins, operators, and other extension mechanisms, has emerged. These tools work in conjunction with K8s orchestrator to enforce security policies effectively.

Despite the availability of such tools, integrating them with K8s often presents challenges. These tools must be tailored to the underlying Kubernetes infrastructure or cloud provider and validated against the specific features required by the deployed services. This is particularly crucial in telco scenarios, where fine-grained control over the network stack is often necessary [13].

These tools often provide varying functionalities and levels of configurability for the aspects they cover, such as network-based isolation. These differences arise from the enforcement mechanisms, which may not support comprehensive or fine-grained control mechanisms or require deep expertise to configure effectively. As a result, security intents may be implemented inconsistently, making it difficult to maintain a coherent and unified security posture across diverse environments.

The persistent complexity of cloud-native systems emerges as a primary challenge across all stages of adoption, underscoring the growing intricacy of managing increasingly integral and distributed architectures [5]. As organizations expand their reliance on cloud infrastructures, regulatory compliance becomes progressively demanding. This is due not only to the inherently dynamic nature of these environments but also to a shift in security responsibility, from predominantly technical controls to broader organizational governance. Compliance efforts are increasingly assessed against established frameworks, with particular emphasis on zero-trust principles, which are both widely recommended and, in some cases, mandated by contemporary standards [11]. Furthermore, the exposure to security incidents in cloud contexts is increasingly important. This vulnerability is largely attributed to the transient and rapidly changing nature of cloud components, which can hinder the enforcement of uniform security policies and create exploitable inconsistencies [5].

The goal of our research is to provide an actionable model of the security capabilities of the enforcement mechanisms available in K8s through the available tools and CNI plugins. This model must also abstract the configuration details

exposed by the enforcement mechanisms. Such modelling would facilitate the adoption of the automatic enforcement of high-level security intents in cloud environments, as it would provide transparent access to security functionality agnostic of the enforcement mechanisms available.

Towards this goal, this paper presents the systematization of concepts related to runtime security enforcement in K8s-orchestrated environments. We introduce a conceptual framework that formally models the commonalities and differences among widely used and emerging security tools. The contributions of our research are:

1. an analysis of the runtime security enforcement capabilities in K8s and a *formal model* capturing the findings. To the best of our knowledge, this is the first attempt to systematize runtime security enforcement in K8s;
2. an *analyzer*, that, from an input K8s cluster, provides an abstract representation of the security requirements implemented. This analyzer also reports the full list of tools available in the K8s ecosystem among the ones we modelled, that are able to enforce them;
3. a *translator* that generates the low-level configuration for a target security tool from an abstract representation of the desired security requirements.

We then demonstrate how this conceptualization enables the automation of security management tasks, particularly in policy translation and intent-based security management. First, our approach facilitates service migration between cloud providers, even when each relies on a different subset of security mechanisms to enforce security policies. Second, we propose a method for generating security configurations for microservice architectures based on high-level security requirements, leveraging cloud-native enforcement mechanisms.

The rest of the paper is organized as follows. Section 2 provides background on K8s security enforcement and related research. Section 3 summarizes the analysis we conducted on K8s security mechanisms and our conceptualization approach. Section 4 and 5 present the model architecture, the components, and workflow our PoC implements. Section 6 explains the validation we performed to assess the effectiveness of our proposal and the performance penalties it adds. Section 7 draws conclusions and provides hints for future works.

2 Background and related works

In cloud-native environments, K8s is the de facto orchestrator to deploy and manage the life cycle of microservices. Several managed cloud solutions build upon vanilla K8s, particularly those offered by major cloud providers, by extending its functionalities or supplying the underlying network plumbing to the K8s cluster. K8s, in fact, on its own, lacks basic networking functionalities, which are instead implemented via CNI plugins, hereafter referred to simply as plugins. Plugins are one of several extension mechanisms offering additional or advanced functionalities. Plugins, in particular, provide pod connectivity, injecting network interfaces into the pods' namespaces and configuring them. The K8s network

architecture follows a novel paradigm to guarantee interconnection among pods and services across namespaces and nodes [8].

CNI plugins are also modular, in that they can be used in conjunction, resulting in hybrid solutions where each plugin provides specific functionalities. A well-known example was Canal, which combined two plugins: Calico for enabling network policies and Flannel for basic connectivity. This possibility has, however, been deprecated in the last versions of Flannel. At the same time, a community project[1] with goals set among several K8s SIG working groups (Special Interest Groups) has introduced a basic network policy implementation based on netfilter, albeit still in development and not appropriate for production use. We anticipate that this implementation will target scenarios where the plugin in use lacks network policy enforcement capabilities, such as Flannel.

Performance comparison of plugins has been proposed by Qi et al. [10], attributing the observed differences to network stack interactions and proposing other useful technical considerations to select one plugin over another.

Additional ways to extend the API-enabled infrastructure provided by K8s include *controllers*, *operators*, and *service meshes*. Each extends the platform's capabilities to address specific needs beyond standard features, such as pod connectivity across nodes and channel protection.

These mechanisms introduce new resources implemented according to the Custom Resource Definition (CRD) specification, which enables the extension of the K8s declarative API. Operators continuously reconcile the current state with the desired state of these resources by taking appropriate actions [2, 15]. Operators and controllers are mainly for resource management and orchestration without a clear focus on security. They have been hence excluded from our model.

Service meshes provide functionality by attaching an additional software layer to pods, abstracting certain tasks away from the core application logic. This is typically achieved by deploying a proxy container alongside each pod exposing a service, using the sidecar pattern. The proxy extends the service's functionality without requiring modifications to the application itself.

Service meshes primarily focus on application-level features such as authentication and authorization, service discovery, traffic management (e.g., load balancing), and observability across services. When it comes to security functionalities, service meshes present severe limitations from an infrastructure layer perspective, as they are designed to manage application services rather than individual pod entities. In fact, service meshes typically operate on top of the network infrastructure provided by the cluster's CNI plugin. As a result, they follow and are constrained by the network rules enforced by that plugin. Even if Layer 7 authorization policies are configured through the service mesh, they will have no effect if the underlying CNI network policies are not aligned.

For these reasons, service meshes have not been explicitly included in our model, they are only considered when they are used by the tools in the scope of our analysis as a way to implement layer 7 features (as it will be explained in Section 3.

[1] https://github.com/kubernetes-sigs/kube-network-policies

This shift in networking design inherently introduces new security risks and needs. Some derive from microservice architectures' highly distributed nature, and others stem from K8s one. Several studies have explored the security implications, challenges, and potential mitigations in K8s-orchestrated environments. A primary focus has been on the security of the orchestrator infrastructure, as it serves as the central component overseeing the life cycle of managed resources. More recently, the security of the services deployed has also been investigated [4, 17]. While the security issues plaguing microservice architecture are investigated, laying the groundwork for the definition of appropriate threat models and their mitigative measures, the ways to enact them are only covered iteratively. Nam et al. [9] propose a network stack specifically designed to address fundamental issues in cloud-native network architectures. Their work, along with other related studies, builds upon a substantial body of research in traditional and software-defined networking. They often focus on proposing frameworks for optimized performance from a networking standpoint or target specific categories of security issues and attacks, with mitigative approaches suited specifically for those. Other research has focused on comparing prominent open-source security tools based on their technical implementation [16], but still, they are mainly concerned with performance evaluation, particularly on their observability and monitoring functionality, rather than assessing the actual features exposed by the tools and their capabilities. Our work instead focuses on systematizing the latter.

The growing adoption of eBPF in the last few years, initially for observability and later for enforcement, highlights the increasing interest in leveraging kernel-level mechanisms to provide native security enforcement within K8s environments. Alongside eBPF, Linux Security Modules (LSM), far predating K8s, originally designed to integrate Mandatory Access Control (MAC) capabilities within the Linux kernel, have seen similar interest. Despite the strengths of these solutions, they are often challenging to configure, given their low-level nature.

Many solutions have been proposed recently to leverage the ePBF capabilities available in recent versions of the Linux kernel. Notable examples include KubeArmor and Tetragon, which are standalone security tools providing observability and runtime enforcement functionalities such as file access control, process execution permissions, and network-based access control and isolation. They gained traction within the cloud-native community as they coupled several novel capabilities with an ease of use, serving as an extension mechanism for K8s. Similarly, plugins have expanded their scope, introducing network security functionalities. Cilium and Calico have notably expanded upon the standard K8s network policy specification by increasing rule granularity and enabling control over additional resources.

3 K8s runtime security enforcement analysis

Enabling automated management of security within K8s environments requires the ability to accurately describe the features that each security mechanism (SM) can provide and the precision in enforcing them into the operational environ-

ment. Moreover, it requires expressing authorization rules to allow their correct configuration.

The abstract model we have built conceptualizes the capabilities and behavior of the SMs based on a thorough analysis of the current solutions and their limitations, features, and granularity.

We analyzed and modeled five SMs, selected based on their feature sets and adoption: *Calico* (v3.28.1), *Cilium* (1.16.5), *KubeArmor* (v1.4.9), *Tetragon* (v1.3.0), and the vanilla K8s network policies model (v1.25.0). The latter is not a standalone SM, but rather represents the default policy specification interface exposed by K8s. It offers a more limited subset of features compared to the other SMs, and requires a dedicated enforcement backend, e.g., a SM, to implement the rules defined using its specification language. Most plugin SMs support the standard K8s network policy API alongside their proprietary extensions, to maintain compatibility. This is true both for Calico and Cilium. Despite its limitations, it remains the official mechanism for specifying network policies in K8s and defines a baseline security model that has been adopted by several other SM.

As previously noted, Flannel, despite its popularity due to being bundled with many lightweight K8s clusters, does not support network policies and was therefore excluded from the analysis. However, complementary enforcement solutions, such as the one previously mentioned, currently being developed within the scope of various K8s SIGs, adhere to the vanilla K8s specification, thereby highlighting its continued relevance as the canonical policy definition model in K8s.

Cilium is built around eBPF. Using it, both the `CiliumNetworkPolicy` and vanilla K8s `NetworkPolicy` rules are enforced by injecting eBPF bytecode directly into the kernel. Layer 7 network policies, however, are enforced via the Envoy proxy, a node-local instance through which traffic is proxied to enforce Layer 7 policies.

Calico operates using either the standard Linux data plane, i.e., by translating network policies into *iptables* rules, or with eBPF, injecting eBPF bytecode similarly to Cilium. Calico is offered in three distinct versions: Open-Source, Cloud, and Enterprise. Our analysis considered only the free version, but feature support, particularly for advanced features, varies between versions. In particular, Layer 7 policies differ in availability and enforcement approach, which will be briefly discussed later in this section.

Tetragon and KubeArmor can enforce restrictions on processes, file access, and network operations by regulating the system-level behavior within pods. KubeArmor relies on Linux Security Modules (LSMs), such as AppArmor and BPF-LSM. The latter is a recently introduced kernel capability that integrates eBPF with LSM hooks. Tetragon, on the other hand, relies entirely on eBPF with a custom implementation for kernel-level enforcement.

Our analysis categorized the analyzed features according to the three main operational defensive techniques in the *Isolate* tactic of the MITRE D3FEND framework [6] (v1.0.0): *Network Isolation*, *Access Policy Administration*, and *Execution Isolation*. Fig. 1 summarizes the main analysis findings. These concepts are the basic elements of the model presented in Section 4.

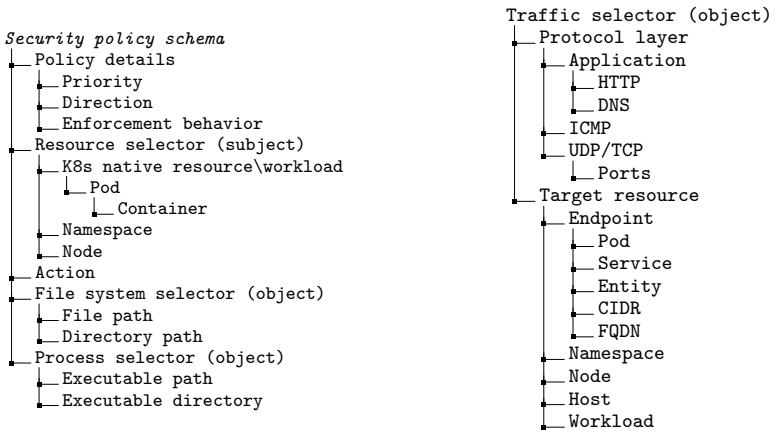

Fig. 1: Summary of the main policy concepts

3.1 Network isolation capabilities

The network-based capabilities are the richest and most complex provided by the SMs we analyzed. They are composed of different elements. The *resource selector* selects the entity, typically pods, that is targeted by the policy, that is, the entity for which network restrictions are defined. Then, the *object selector* identifies the traffic targeted by the policy action with a list of rules selecting the other end of the communication channel, the type, and details of the selected traffic. The *resource selector* identifies the entity to be protected, typically pods. The *object selector* then identifies the traffic targeted by the policy action through a set of rules that specify the other end of the communication channel, the traffic type, and other details of the selected traffic. This element is where most of the specification's granularity resides.

Then, the *action element* specifies what happens on the pinpointed objects of the identified resource, such as denying or logging the communication. Other traffic concepts are used to configure SMs, such as the communication direction and fundamental policy enforcement behaviors, such as policy resolution strategies and allow-deny listing behavior.

The most widely used plugins are currently also the most feature-rich SMs. Network policies enforced by plugins typically use an allow-list approach to implement restrictions. By default, each pod can communicate with any other pod without restriction. As soon as a network policy selects a pod, only the selected traffic, in that specific direction, will be allowed. This is typically referred to as the "default-deny" behavior. Vanilla K8s network policies, together with plugin network policy specifications, adhere to it.

KubeArmor and Tetragon also provide network restrictions, albeit with significantly looser granularity and reduced functionality. The former enforces network policies at the container level by configuring the behavior of system calls for network operations (e.g., `bind()`, `listen()`, `accept()`, and `connect()`). For Cilium, we considered only the `CiliumNetworkPolicy` resource and left the

`ClusterWide` to future work. The same applies to Calico's managed resources capabilities. Tetragon has similar functionalities but doesn't offer a high-level interface, requiring individual rules for the system calls involved in the target network operation. Moreover we haven't considered cross-cluster policies, that require the Cluster Mesh functionality, both in Cilium and Calico, the latter limited to the enterprise version. The other two SMs do not support multi-cluster functionalities.

The following paragraphs summarize some of the main findings concerning the network isolation capabilities we modeled.

Priority. Cilium network policies have only two priority levels, whereas deny policies take precedence over allow ones. In Calico, the priority can be set freely with an integer instead. The resulting behavior in the network for a specific traffic *event* is derived from the union of all deny and allow policies, with their respective precedence order. The vanilla K8s specification instead lacks an explicit priority field, thus, their effect is determined by the union of all policies selecting a specific pod.

Selector mechanisms. In K8s, given the ephemeral nature of pods, the resource and traffic selection in policies typically use labels instead of IP addresses to cope with IP address changes. Hereafter, when we refer to pod selection, we mean the case in which a policy targets a pod, i.e., when the policy includes that specific pod in its subject or resource selector field. Entities external to the cluster can be referenced by traditional identifiers such as IP addresses or Fully Qualified Domain Names (FQDNs). Both analyzed plugins support this, while KubeArmor does not support specifying the destination endpoint. Vanilla K8s policies only support K8s-native `LabelSelector` for traffic endpoint selection. However, Calico support for DNS-based policy through FQDN selectors for the destination endpoint is limited to the enterprise versions. Whereas Calico has a single selector supporting multiple entities, Cilium uses multiple sub-selectors, each used for different types of entities. Both plugins also support non-K8s-native entities, typically relevant in managed environments. For example, Cilium's entity selector allows the specification of network infrastructure components such as the K8s API server, health endpoints, and external networks. A current limitation in our mapping concerns application-layer capabilities, which are currently limited to HTTP and DNS. However, Cilium supports additional protocols via the Envoy proxy, such as gRPC and Kafka (the latter currently in beta). Furthermore, Cilium provides a framework to extend its capabilities by supporting additional Layer 7 protocols, including custom ones, through Envoy, by allowing the definition and implementation of the appropriate custom parsing logic. This feature remains in beta at the time of writing, but significant community contributions are expected to expand support for more Layer 7 protocols in the future.

Protocol support. The main distinction at the network level lies between plugins and standalone tools. Plugins provide higher-level protocol support at Layer 7 (L7), whereas tools like KubeArmor and Tetragon do not. Specifically, these tools do not natively support the definition of application-layer policies, such as those targeting specific HTTP methods, paths, or headers, as they operate at

lower levels of the network stack (i.e., L3/L4 and system call level). For example, KubeArmor supports rules for TCP, UDP, and ICMP traffic. Both Cilium and Calico support L3/L4 network policies. However, in terms of L7 protocols, Cilium is more advanced, as previously noted. It supports multiple application-layer protocols and allows for extensibility. Cilium also supports L7 DNS policies, which define rules for DNS queries themselves. These are distinct rules from the ones using fully qualified domain names (FQDNs) merely as selectors in L3/L4 policies. Vanilla K8s policies support TCP, UDP, and SCTP. Calico's support for L7 policies merits further discussion. Our analysis reveals that the technical implementation differs between the open-source version and other editions. In the open-source version, L7 policy enforcement requires deploying the Istio service mesh within the cluster. In this configuration, each pod is paired with an Envoy sidecar proxy, through which traffic is tunneled and policy enforcement is enabled. In contrast, other Calico versions appear to use an internal implementation for L7 policy support, similar to Cilium. In the first case, the policy capabilities are tied to those offered by Istio, whereas in the latter, a proprietary specification is provided. We hypothesize that these architectural differences may also result in significant performance variations. However, to the best of our research, no detailed performance comparisons have been presented in the literature.

Granularity of rules. With KubeArmor, fine-grained rules at the process level can be enforced, enabling network restrictions targeting singular executables specified in the policy, which is a unique feature among the SMs we analyzed. But, on the other hand, its functionalities are particularly limited regarding the traffic selection as it does not allow specifying ports but only protocol specification. Similarly, Tetragon allows defining low-level rules targeting even specific system calls being used. We found these functionalities only in these standalone tools, as plugins typically expose higher-level APIs, hence with a higher-level interface.

Enforcement behavior and supported actions. An important feature recently introduced in both Cilium and Calico is the use of deny rules. These rules allow for expressing policies with a deny-listing behavior rather than the default allow-listing approach. However, there are differences between the two plugins. In Calico, the action field within each rule specifies whether traffic should be allowed or denied, regardless of the policy's direction. In contrast, Cilium uses direction-specific fields, `ingressDeny` and `egressDeny`, to specify dedicated deny rules to block traffic explicitly. Additionally, deny rules in Cilium do not support L7 rules, and also the use of FQDN entity selectors in the traffic selector of L3/L4 policy rules. Importantly, we note that plugins follow an allow-list model that triggers default-deny behavior as soon as a policy selects a pod, even when using deny policies. This means that once a deny rule selects a pod in a given direction and specifies that certain traffic is forbidden, all other traffic is also denied and dropped by default, unless it matches at least one allow rule. This behavior may be undesirable and can be avoided, for example, by accompanying deny rules with a global allow rule for each pod targeted by such policies. This approach

effectively enables a deny-listing model for restricting network access. Vanilla K8s network policies, by contrast, do not support deny rules at all. In Cilium, alternatively, the allow-list behavior can be disabled via the *enableDefaultDeny* setting, which allows disabling it per direction. In that case, deny policies can be used without automatically dropping non-targeted traffic, eliminating the need to add a global allow rule alongside every deny rule within a pod policy. However, at the time of writing, *enableDefaultDeny* does not apply to L7 policies. Calico supports the `Log` action, disabling enforcement to monitor the selected traffic. This option is not supported at the same level in other SMs. For example, in Cilium, there's a global `Audit` option with similar behavior, but that can be enabled per cluster and directly on the CNI daemon. Once activated, the enforcement of the applied policies is turned off. KubeArmor supports both deny and allow listing with `Block` and `Allow`, and an audit mode with the `Audit` action.

3.2 File access restriction capabilities

These capabilities are currently only supported by two of the SMs we analyzed, Tetragon and KubeArmor. The policy they support enables defining path-specific file access restrictions and pinpointing the protection of individual sensitive files, such as configuration files, certificates, or credentials. Directory-based controls enable broader protection across entire directory trees.

These restrictions can be applied at the pod level, ensuring that all processes run within their context comply with the policy or with higher precision for specific executables run within the pod. Both SMs support `Allow`, `Block`, and `Audit` actions. KubeArmor has native support for those actions.

With Tetragon, it is necessary to craft a low-level policy for handling syscalls instead. In particular, this can be done either by altering the return value of a function to prevent its execution or by sending signals, such as `SIGKILL`, to terminate processes that meet specific criteria. Depending on the desired behavior, a policy can use an override or signal action, which can also be combined. In our case, we mapped the generic `Deny` action as a combination of `SIGKILL` and `Override` of the return value with -1. The generic `Allow` and `Log` actions are mapped, respectively, to Tetragon's `NoPost` and `Post`.

3.3 Process execution restriction capabilities

The process execution restriction capability modeling and its translation behavior follow the same pattern as the file restrictions, since they operate on file system elements. KubeArmor process restriction can be enforced by specifying the `Allow`, `Block`, or `Audit` action.

Tetragon process restriction follows the same pattern as the file access restriction capabilities in that the limits are enforced by manipulating the system calls used for the operation. Hence, we do not repeat the detailed explanation here. The only difference between process and file access restriction capabilities lies in the system calls targeted by the policies.

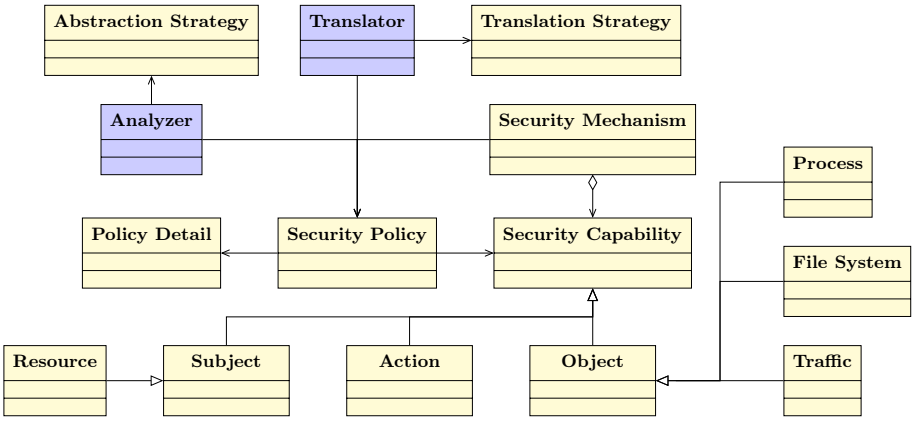

Fig. 2: Model architecture

4 The model

The model categorizes the concepts to specify policies that drive the behavior of K8s SMs, as shown in Fig. 2. This abstract model systematizes the security features that can be enforced through the SMs we've analyzed and the rules they rely on to configure them.

A *Security Policy* refers to one or many *Security Capabilities* and additional *Enforcement Details* that better determine the way a policy will be evaluated and enforced [3].

The model design is based on a hierarchical scheme with high-level concepts representing the features provided by SMs, regardless of the configuration details required to implement them properly. Within each high-level concept, low-level characteristics are represented by subclassing the higher-level entity, further refining the feature with increasing levels of granularity. Security policies are statements expressed according to the *subject-action-object* authorization policy paradigm [14]. Hence, each security capability is categorized according to the *Subject*, *Object*, and *Action* classes. The subjects determine the K8s resources on which the policy applies, and objects clarify the entities on which the authorization action needs to be enforced. The objects identified by our analysis have been determined via subclassing using the *Traffic*, *File System* and *Process* classes. The two tasks, further described in Section 5, that the model allows to perform are represented in this model. Indeed, the model includes the *Analyzer* (*AZ*) and *Translator* (*TR*), and their configuration details, i.e., the *Abstraction Strategy* and the *Translation Strategy*.

5 Implementation

We implemented a proof-of-concept for exploiting the security enforcement capabilities our model exposes with the *AZ* and the *TR*. Both tools have been developed as standalone Python modules.

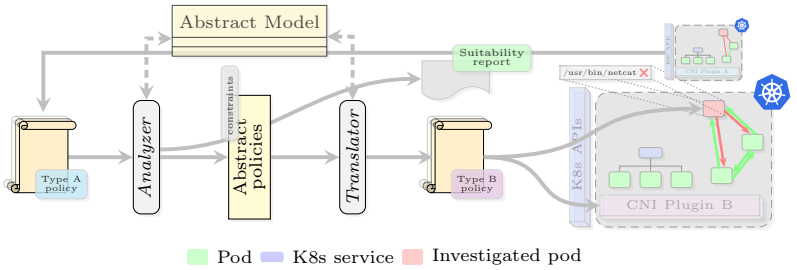

Fig. 3: Functional architecture and workflow

The functional architecture of our tools is presented in Fig. 3. The *AZ* gathers all network policies applied within a K8s cluster using the K8s APIs. The collected information is represented as an instance of our Abstract Model. It includes the concrete values of the policy and the target SM.

The *TR* gets as input abstract policies. The administrator can manually write these policies, be the output of the *AZ*, or other tools. These functionalities allow our approach to be used seamlessly for intent-based security management workflows. The generic features the abstract policies describe are converted into SM-specific semantically-equivalent constructs.

5.1 Analyzer

The *Analyzer (AZ)* component first parses the YAML content of each policy found in the target K8s system and verifies that the policy is syntactically and semantically correct. This step requires checking the presence of given keys and values in the data structure and their positioning.

Then, policies are compared against a predefined set of security concepts with rule-matching. This step transforms enforcement-specific configurations into high-level, SM-agnostic logical rule statements and constitutes the Abstraction Strategy details. It decomposes policy attributes according to their SM-specific patterns, normalizes identifiers, and applies a set of transformation functions that remove direct dependencies on SM-specific constructs while preserving semantics, breaking it down into the key components (i.e., `subjects`, `actions`, `object`, and `enforcement constraints`).

The *AZ* returns two outputs: a *suitability report*, which includes the list of SMs able to enforce that input policy and the list of unsupported or missing features or concepts, and an abstract policy, representing the high-level description of the input policy, conveying the generic features and its related security requirements, stripped of their original SM language syntax and semantic specifics. The suitability report also includes the features in the input policy that are not yet represented in the model, as is the case with Cilium's `authorization` feature.

5.2 Translator

The *Translator (TR)* component receives abstract policies as input, e.g., the data structure produced by the AZ, which contains generic features as defined by the abstract model, the concrete values of the policy, and the target SM. The translation engine then processes these features by routing their data to the appropriate converter classes, reported in the Translation Strategy class details. These classes inherit the hierarchical structure of the abstract model, as they are bound to each generic feature and provide the necessary conversion functions for each composing element.

For example, the `Traffic selector` will invoke the function of the `Pod` entity, following the chain from `Endpoint selector`. These functions handle the conversion from generic features to SM-specific constructs while ensuring semantic equivalence across different implementations of the same isolation capability. This guarantees that the converted policy maintains the same enforcement behavior as the original.

For instance, the *TR* component manages the differences in enforcement behavior arising from the different implementations of deny policies in Calico and Cilium. A simple abstract deny-policy for specific traffic will be translated in Cilium to a policy with the default-deny behavior disabled and a deny direction selector. At the same time, in Calico, it will be converted into an allow policy that grants full access, combined with a deny policy that is assigned a higher priority. In the current PoC, the generated policies are stored in the file system, but future work will integrate them with the K8s APIs.

6 Validation and Discussion

We evaluated the effectiveness of our framework in three use cases. The first two are based on widely used microservice architectures, selected after reviewing several publicly available solutions. Aderaldo et al. [1] proposed a benchmark for such architectures. However, many of these have since been abandoned or are no longer actively maintained. Ultimately, we based our validation on the Google Online-Boutique demo, one of the most up-to-date and comprehensive solutions used for evaluation of enterprise cloud solutions, and Sock Shop, which serve as the first and second use case scenarios, respectively.

These architectures are frequently used in research as experimental environments and have been employed in prior studies on policy-based access control and network isolation in K8s environments [7]. Regardless of the specific services exposed, i.e., online shops, they exemplify complex, distributed, real-world microservice applications. They consist of multiple interconnected services that necessitate network-level segmentation, and spanning a variety of frameworks and programming languages, reflecting heterogeneous conditions typical of cloud-native environments across IT domains. Online-Boutique has also been adopted for validating internal components of the iTrust6G EC-funded project's framework. In both scenarios, segmentation is applied at the pod level using the vanilla K8s specification, following an allow-listing approach.

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  namespace: default
  ...
spec:
  description: ...
  endpointSelector:
    matchLabels: app: example-service
  ingress:
    - {}
  egress:
    - toEndpoints:
        - matchLabels: app: example-service
      toPorts:
        - ports:
            - port: "80"
    - toEndpoints:
        - matchLabels:
            io.kubernetes.pod.namespace:
                kube-system
            k8s-app: kube-dns
      toPorts:
        - ports:
            - port: "53" protocol: UDP
          rules: dns:
              - matchPattern: "*.google.com"
```

```
apiVersion:
    projectcalico.org/v3
kind: NetworkPolicy
metadata:
  namespace: default
  ...
spec:
  description: ...
  selector: app ==
      'example-service'
  egress:
  - action: Allow
    destination:
      selector: app ==
          'example-service'
      ports:
      - 80
  - action: Allow
    protocol: UDP
    destination:
      selector: k8s-app ==
          'kube-dns'
      namespaceSelector:
        pod.namespace ==
            'kube-system'
      ports:
        - 53
```

```
{"details": {"namespace": "default"},
"enforcement_behavior": "default-deny",
"resource_selector": {"type": "pod", "filter_labels":[{"app": [
    ↪ "example-service"]}]},
"rules": [{"direction": "Egress", "action": "allow", "traffic_selector": {
    "type": "pod", "proto": ["UDP"], "port": ["53"],
    "filter": {"type": "labels", "filters": [{"pod.namespace": [
        ↪ "kube-system"]}]},
    "payload_rule": {"proto": "DNS", "filter": {"header": {}, "payload": {
        ↪ "pattern": "*.google.com"}}}}
},{"direction": "Egress", "action": "allow", "traffic_selector": {
    "type": "pod", "proto": [], "port": ["80"],
    "filter": {"type": "labels", "filters": [{"pod.namespace": [
        ↪ "kube-system"]}]}
}}]}
```

```
{{"sm_compatibility": {"Calico": {"policy_compatilibity": "partially␣
    ↪ supported", "capabilities":
  [{"type": "enforcement_behavior", "id": "default-deny","unsupported": []},
    {"type": "policy_action","id": "action","unsupported": []},
    {"type": "resource_sel","capability": "entity-sel","sel":
        ↪ "labels","unsupported": []},
    {"rule": "2","type": "traffic_sel","capabilities": {"details": {"proto":
        ↪ {},"port":{},"unsupported":[]}, {"rule": 1, "unsupported": []}
    "entity_sel": {"sel_type": "labels","entity_type":
        ↪ "pod","unsupported":[]},
    "payload_rule": {"rule_type": "DNS","unsupported":["DNS"]}}}]},
"KubeArmor": {"policy_compatilibity": "unsupported","capabilities":
   [{"type": "enforcement_behavior","capability":
       ↪ "default-deny","unsupported": ["default-deny"]},
      {"type": "policy_action","capability": "action","unsupported": []},
      {"type": "resource_sel","capability": "entity-sel", "selector":
          ↪ "labels","unsupported": []}, {"rule": 1, "unsupported": [
          ↪ "port", "entity_type", "sel_type"]}
      {"rule": 2, "type": "traffic_sel","capabilities": {"details": {
          ↪ "proto": {},"port": {}, "unsupported": ["port"]},
      "entity_sel": {"sel_type": "labels","entity_type":
          ↪ "pod","unsupported":["entity_type", "sel_type"]},
      "payload_rule": {"rule_type":
          ↪ "DNS","unsupported":["rule_type"]}}}]}},{"..."}}}
```

Fig. 4: Policy translation at the top, security requirements of abstract policy and suitability report excerpts, redacted for brevity, in the middle, and at the bottom, respectively.

In these scenarios, we simulated a cloud provider wanting to migrate from different SMs, as in Fig. 3. We prepared *ad hoc* K8s clusters with the target SMs, created with Minikube and hosted on an OpenStack infrastructure with Intel Xeon 32 vCPUs, Cascade Lake, 128GB RAM.

For Boutique, the migration was from native K8s to Calico, and for Sock Shop to Cilium. The validation workflow employed the *AZ* and the *TR*. The network policies deployed in the original clusters were retrieved, selecting all `NetworkPolicy` resources, 14 for Boutique, and 13 for Sock Shop. Their correctness was verified, and an abstract representation was crafted according to the *Abstract model*, extracting all security features and then translating into the language of the SM in the target cluster. The *suitability report* by the *AZ* reported that both Cilium and Calico supported the origin network isolation prescriptions, whereas KubeArmor lacked sufficient granularity, as its system-based enforcement mode does not support specific ports at the transport layer, used in the original network policies. We report an example of translation performed by our tools in Fig. 4.

In the Cilium case, the selector mechanism was translated to the specific entity selector for the resource used in the origin policies `endpointSelector`, i.e., pods, based on the label filter. A deny-all default policy was present in both, even though Sock Shop used an early beta-version expression dating back to before the K8s `NetworkPolicy` specification was stabilized and nowadays deprecated. Hence, the *AZ* reported it as unsupported and discarded it in the translation process. Finding unsupported features or configurations may endanger the security, as for the Sock Shop cluster, given that the isolation would be missing, hence the importance of reporting the issue and the need for correct policy configurations due to the intrinsic difficulties in handling different SM formats.

Aside from this, the *TR* correctly handled the resource selector, using labels for selecting the specific pod resources and the traffic selector, which in both Boutique and Sock Shop were specified at the transport layer using ports, protocol, and the traffic direction. Once generated, we manually pushed them into the destination clusters and verified the correctness of the pods' connection restrictions as per the policy specified with the *netcat* utility. For this purpose, we attached a "sidecar" ephemeral container with the utility pre-installed to the pods investigated, as many of the pods' containers did not expose a shell. As expected, a major isolation gap was found in Sock Shop, as the missing global ingress isolation policy caused all incoming traffic to pods other than the directions filtered by the policies to be permitted.

As the third and last use case, we tested the use of our tools to implement an intent-based workflow, i.e., to refine high-level requirements into network security configurations ready to use for target K8s networks. The high-level requirements were imported, through the *AZ*, from a catalog of existing K8s system- and network-based hardening requirements for specific services or infrastructures,

For this last scenario, we considered the open-source catalogue of hardening rules publicly maintained and curated by AccuKnox[2], designed for common

[2] https://github.com/kubearmor/policy-templates

cloud-native and 5G telco infrastructures and services. The catalogue includes both system-level and network-level isolation rules. Among its recommendations are numerous controls and related rules mapped to widely adopted compliance frameworks, such as NIST SP 800–53, CIS Benchmarks, and STIG, as well as mitigations aligned with common adversary techniques identified in the MITRE ATT&CK framework. Many of these rules provide technical guidance and implementation benchmarks that play a central role in governance processes aimed at achieving a security baseline, particularly within organizations adopting structured methodologies like the Risk Management Framework (RMF).

We fed each hardening requirement into the *AZ*. Our tool showed the mechanisms supporting the concepts implied by the recommendations and identified any missing ones when applicable. In particular, it reported that the system-based recommendations for KubeArmor were only supported by one other SM, that is, Tetragon, while the network-based recommendations from Cilium were only partially supported by Calico, given the presence of certain selectors not available in it, and not by KubeArmor. Additionally, the suitability report issued when analyzing the group, highlighted the use of the `CiliumClusterWide` policy type that is not yet supported by our model, specifically the `NodeSelector`, together with Kyverno policies. Kyverno has not been covered in our research as it focuses on admission-time security enforcement, rather than on runtime enforcement, which is the focus of this research, and has been left to future work. In Fig. 4, we show an example with its abstract representation (bottom right), where the concepts and concrete constraints are reported. In particular, we note that starting from those high-level requirements, as per the original policy (left), a resulting Calico instance would be lossy since the layer 7 DNS capability, filtering DNS traffic based on their request, is not supported in Calico.

Finally, we measured the performance of analysis and translation for the two migration scenarios using the `timeit` utility. As expected, the footprint was minimal, measuring just under 0,147 ms and 0,132 ms on average, with a 2 µs standard deviation for the Boutique and Sock Shop migrations, respectively. The cluster network performance varies depending on the SMs, as some of them, i.e., plugins, also provide the underlying network connectivity, hence being the main element in determining the actual network performance. However, this variation is independent of the policy refinement and was not considered in our performance analysis. The same applies to the policy loading time to the K8s APIs and enforcement time. We refer to existing research for further insights into these aspects, such as [10].

7 Conclusions

In this paper, we investigated the landscape of K8s native runtime enforcement of security policies by analyzing the most widely used and promising solutions, comparing their features and functionalities, and providing a general overview that can help identify the most effective combination of enforcement mechanisms to achieve the desired security goals. This work is important to help automate

tasks that are needed for compliance with governance standards, which are increasingly required to prove effectiveness in coping with the cybersecurity risks of emerging threats in cloud environments.

Furthermore, we introduced a model-driven approach to translating high-level abstract requirements into valid configurations for a set of SMs for K8s that resulted from the conceptualization effort. This has led to an abstract model of the capabilities exposed by various K8s security solutions. This abstraction streamlines management operations for K8s security policies by providing a higher-level representation that abstracts away the complexities from the configuration details.

We evaluated our approach using three use cases: two cloud-provider migrations involving real-world, complex microservice architectures, which required adapting security policies to different SMs, and the analysis of a public policy catalog for the security hardening of cloud-native environments, including common services and 5G architectures. Our goal is to integrate the runtime security enforcement abstraction capabilities into a cloud-native, intent-based framework developed in previous work [13], as well as into a playbook-based security orchestration platform to support the orchestration of security operations in K8s environments [12].

Acknowledgments

This work is supported by the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme under Grant Agreement No. 101139198, iTrust6G project. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or SNS-JU. Neither the European Union nor the granting authority can be held responsible for them.

References

1. Aderaldo, C.M., Mendonça, N.C., Pahl, C., Jamshidi, P.: Benchmark requirements for microservices architecture research. In: 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE). pp. 8–13 (2017). https://doi.org/10.1109/ECASE.2017.4
2. Attaoui, W., Sabir, E., Elbiaze, H., Guizani, M.: VNF and CNF Placement in 5G: Recent Advances and Future Trends. IEEE Trans. on Network and Service Management **20**(4), 4698–4733 (2023). https://doi.org/10.1109/TNSM.2023.3264005
3. Clemm, A., Ciavaglia, L., Granville, L.Z., Tantsura, J.: Intent-Based Networking - Concepts and Definitions. RFC 9315 (Oct 2022). https://doi.org/10.17487/RFC9315
4. Haindl, P., Kochberger, P., Sveggen, M.: A systematic literature review of inter-service security threats and mitigation strategies in microservice architectures. IEEE Access **12**, 90252–90286 (2024). https://doi.org/10.1109/ACCESS.2024.3406500

5. Hendrick, S., Lawson, A., Sica, J.: 2024 Cloud Native Security Report. Tech. rep., The Linux Foundation (2024). https://doi.org/10.70828/MRCE5096, https://www.linuxfoundation.org/hubfs/Research%20Reports/lfr_cloudnative_security24_101424a.pdf
6. Kaloroumakis, P.E., Smith, M.J.: Toward a knowledge graph of cybersecurity countermeasures. The MITRE Corporation **11** (2021)
7. Li, X., Chen, Y., Lin, Z., Wang, X., Chen, J.H.: Automatic policy generation for Inter-Service access control of microservices. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 3971–3988. USENIX Association (Aug 2021)
8. Minna, F., Blaise, A., Rebecchi, F., Chandrasekaran, B., Massacci, F.: Understanding the security implications of kubernetes networking. IEEE Security & Privacy **19**(5), 46–56 (2021). https://doi.org/10.1109/MSEC.2021.3094726
9. Nam, J., Lee, S., Seo, H., Porras, P., Yegneswaran, V., Shin, S.: BASTION: A security enforcement network stack for container networks. In: 2020 USENIX Annual Technical Conf. (USENIX ATC 20). pp. 81–95. USENIX Association (Jul 2020)
10. Qi, S., Kulkarni, S.G., Ramakrishnan, K.K.: Understanding container network interface plugins: Design considerations and performance. In: 2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN). pp. 1–6 (2020). https://doi.org/10.1109/LANMAN49260.2020.9153266
11. Rose, S., Borchert, O., Mitchell, S., Connelly, S.: Zero trust architecture (2020). https://doi.org/https://doi.org/10.6028/NIST.SP.800-207, https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=930420
12. Settanni, F., Regano, L., Basile, C., Lioy, A.: A model for automated cybersecurity threat remediation and sharing. In: 2023 IEEE 9th Int. Conf. on Network Softwarization (NetSoft). pp. 492–497 (2023). https://doi.org/10.1109/NetSoft57336.2023.10175486
13. Settanni, F., Zamponi, A., Basile, C.: Dynamic security provisioning for cloud-native networks: An intent-based approach. In: 2024 IEEE Int. Conf. on Cyber Security and Resilience (CSR). pp. 321–328 (2024). https://doi.org/10.1109/CSR61664.2024.10679397
14. Snyder: Formal models of capability-based protection systems. IEEE Trans. on Comput. **C-30**(3), 172–181 (1981). https://doi.org/10.1109/TC.1981.1675753
15. Vayghan, L.A., Saied, M.A., Toeroe, M., Khendek, F.: A kubernetes controller for managing the availability of elastic microservice based stateful applications. J. of Syst. and Softw. **175**, 110924 (2021). https://doi.org/https://doi.org/10.1016/j.jss.2021.110924
16. van Vugt, T.M., Malik, T.: A practical analysis of open-source security tools in microservice kubernetes environments. In: 2023 Cyber Research Conference - Ireland (Cyber-RCI). pp. 1–8 (2023). https://doi.org/10.1109/Cyber-RCI59474.2023.10671405
17. Yarygina, T., Bagge, A.H.: Overcoming security challenges in microservice architectures. In: 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE). pp. 11–20 (2018). https://doi.org/10.1109/SOSE.2018.00011