IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 10 October 2024, accepted 27 October 2024, date of publication 30 October 2024, date of current version 11 November 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3487990

RESEARCH ARTICLE

KUBEAEGIS: A Unified Security Policy Management Framework for Containerized Environments

BOM KIM AND SEUNGSOO LEE
Department of Computer Science and Engineering, Incheon National University, Incheon 22012, Republic of Korea

Corresponding author: Seungsoo Lee (seungsoo@inu.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) funded by Korean Government (MSIT) under Grant 2022R1C1C1006093.

ABSTRACT Recently, containers have become the standard for cloud-native service delivery, ensuring scalability and reliability. However, they are also prime targets for various security attacks that exploit vulnerabilities. In particular, deploying security policies in dynamic cloud-native environments presents significant challenges, such as misconfigurations arising from the heterogeneity of different security policies. Despite numerous attempts to address these challenges, existing solutions often lack a unified framework for consistently managing and enforcing heterogeneous security policies across network, system, and cluster layers. Current approaches typically focus on isolated aspects of security rather than providing a comprehensive policy management solution. This fragmentation leads to inconsistencies, inefficiencies, and potential security gaps. To address these challenges, in this paper, we propose KUBEAEGIS, an advanced and unified policy management framework designed to manage the integration, verification, and enforcement of heterogeneous security policies at the network, system, and cluster levels. Our framework enables centralized management of security policies, simplifying the integration of new security tools through an adapter-based approach and API recommendation mechanisms. We also incorporate a pre-validation process to detect potential misconfigurations before policy enforcement and to enable real-time tracking of policies applied to containers. Our evaluation demonstrates the effectiveness of KUBEAEGIS in integrating and managing network, system, and cluster security policies in real cloud-native environments, providing extensive coverage and achieving a minimal translation delay of approximately 17ms.

INDEX TERMS Container security, network security policy, policy management.

I. INTRODUCTION

With cloud adoption rates reaching approximately 94% among enterprises today, the significance of cloud-native technologies is growing [1], [2], [3]. Notably, containers have become the standard for delivering cloud-native services due to their scalability, reliability, and observability. However, the rising popularity of containerized applications has made them attractive targets for security attacks that exploit misconfigurations and vulnerabilities. According to Check Point's Cloud Security Report, 61% of organizations reported breaches in 2023, a significant increase from 24% in 2022 [4].

The data breach at Capital One [5] was a major incident that resulted from an attack that exploited a server-side request forgery (SSRF) [6] vulnerability in Amazon Web Services (AWS) Cloud [7]. This vulnerability allowed attackers to access sensitive data by sending requests to server resources using untrusted input from a web application. Another case, Tesla Cryptojacking [8], involved a misconfiguration in a cloud, enabling attackers to infiltrate the system and mine cryptocurrency. The attackers accessed Tesla's cloud infrastructure, used resources without authorization, and significantly increased resource costs. In a separate incident involving Microsoft's Azure [9], an attack exploited the Super FabriXss vulnerability [10], allowing access to sensitive data by bypassing the cloud authentication system. This attack

The associate editor coordinating the review of this manuscript and approving it for publication was Ali Kashif Bashir.

leveraged a Cross Site Scripting (XSS) [11] vulnerability to bypass Azure's authentication procedures and escalate privileges.

To prevent such malicious behaviour, container orchestrators (e.g., Kubernetes [12]) are playing a central role in management of security policies for large numbers of containerized applications. In a cloud-native environment, robust security policies are essential for network, system, and cluster management security to protect applications and data. These policies help suppress malicious container behavior and prevent data leakage and unauthorized access. However, in dynamic cloud-native environments where containers are created and destroyed within seconds and dependencies between microservices are continuously evolving, deploying these policies poses significant challenges to administrators [13].

Two major challenges in security policy management are *automation* and *misconfiguration*. First, cloud-native environments are dynamically changing and consist of numerous container-based microservices, significantly increasing the complexity of managing security policies. In particular, frequent cluster configuration changes and complex interactions between services make it nearly impossible to manually manage hundreds of security policies. For example, in a large-scale microservice architecture, setting detailed network policies for each service requires excessive time and effort if done manually. Second, such manual management can easily lead to misconfiguration of security policies. Important security policies may be omitted due to minor user errors such as typos or missing settings. These errors can compromise the security of the entire system and provide opportunities for attackers to exploit and infiltrate the system. Additionally, misconfigurations make it difficult to maintain consistency in security policies. New containers are frequently created or deleted, necessitating immediate adjustments to security policies. However, detecting and correcting these changes manually is challenging.

To address these challenges, several work [14], [15], [16], [17], [18] that focus on automatically generating network and system security policies have been proposed. However, these systems still operate dependently on specific security tools, which are not integrated. Even with automatic policy generation, these systems use different policy languages and configuration methods, requiring significant time to learn and manage the security policy configurations for each tool. Additionally, there may be inconsistencies between the user's security requirements and the automatically generated policies. The generated policies may not accurately reflect the security requirements or may not fully understand the current cluster situation, resulting in unnecessary restrictions that could hinder cloud operations. Conversely, a configuration error that grants excessive authority could allow malicious access, leading to a security incident. Lastly, the automated systems typically regard the generated policies as individual instances and do not track and manage the information about which containers each policy is applied to. Consequently, users are not notified when a container to which a security policy was applied is changed or deleted, rendering the policy invalid.

A. OUR APPROACH

In this paper, we propose KubeAegis, a unified policy management framework designed to manage the entire process of integration, verification, and enforcement of heterogeneous security policies at the network, system, and cluster levels. This allows administrators to apply the security policies centrally without having to manage each tool separately. Additionally, we design an adapter-based approach to facilitate the easy integration of new security tools into our system. To this end, we provide an API pool that can convert our security policies into actual security tools. Furthermore, we recommend appropriate conversion APIs based on a SRoBERTa model [19], [20] during the new adapter generation stage, simplifying the expansion of new security policy enforcement tools. This minimizes the need to modify the integrated management system itself, allowing for flexible accommodation of various security requirements. Before enforcing the security policies, we detect potential misconfigurations through a pre-validation process, preventing the deployment of incorrect policies into the cloud environment. Lastly, during the enforcement of the integrated security policies, we track in real time which containers each policy is applied to and immediately detect when containers are changed or deleted, ensuring the consistency of the security requirements.

Our evaluation demonstrates how our system effectively integrates three types of security policies (i.e., network, system, and cluster) in real cloud-native environments. It also shows that our system incurs a minimal translation delay of approximately 17ms in security policy translation.

B. CONTRIBUTION

Our paper contributions are as follows:

- We introduce a novel unified policy management framework, KubeAegis, for containerized environments that manages network, system, and cluster-level security policies through a single, centralized interface.
- We devise a flexible plug-in adapter design with high accuracy API recommendation mechanisms leveraging the SRoBERTa model. This design simplifies the adapter generation process, thereby facilitating the expansion of new security policy enforcement tools.
- We propose a fine-grained policy validation methodology that checks the validity of policies before they are applied to detect potential misconfigurations in advance and preemptively block incorrect policy configurations.
- We evaluate our system within the context of a real-world microservices application. This evaluation effectively demonstrates its capability to successfully integrate various security tools and validate security policies with minimal overhead.

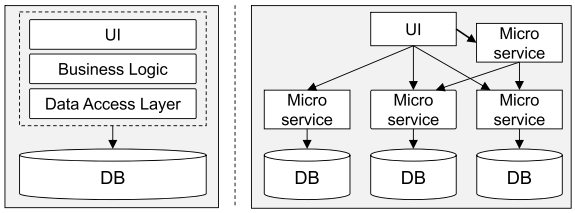

FIGURE 1. Comparison between monolithic and microservice architectures.

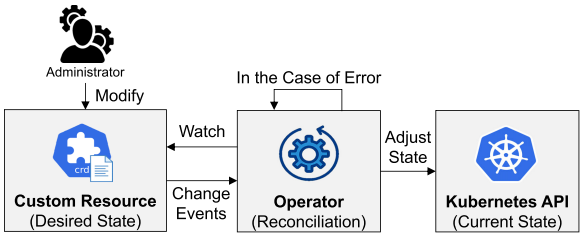

FIGURE 2. Kubernetes operator workflow: Aligning current state with desired state.

The remainder of this paper is organized as follows: Section II-A provides background on cloud-native environments and security policy management challenges. Section III reviews related work and its limitations. Section IV outlines the design and key requirements of KubeAegis. Section V delves into KubeAegis's core features and operational principles. Section VI presents the implementation and performance evaluation of KubeAegis. Section VII discusses system limitations and future work. Finally, Section VIII concludes the paper with a summary of key findings and implications.

II. BACKGROUND AND CHALLENGES

A. BACKGROUND

1) CLOUD-NATIVE APPLICATION

Cloud-native applications are designed to maximize the agility and scalability that are the hallmark features of the cloud computing model. These applications are primarily based on containers, which provide numerous benefits for practitioners in software development and deployment, including agility, portability, reproducibility, modularity, and flexibility [21], [22], [23], [24]. By incorporating containers, these applications enhance the operational efficacy of large-scale systems, particularly in a distributed microservice architecture (MSA). Additionally, cloud-native applications are built to effectively utilize the elasticity and distributed processing capabilities of the cloud during the creation, deployment, and operation stages, which fundamentally distinguishes them from traditional monolithic architectures.

As shown in Figure 1(A), in the traditional monolithic architecture, the user interface, business logic, and data access layers are tightly coupled within a single application, primarily built on physical infrastructure or virtual machines. While this structure simplifies initial development and deployment, it reveals maintenance difficulties and scalability limitations as the application complexity and scale increase. In particular, the inefficiency of redeploying the entire application for a single feature update becomes problematic. On the other hand, as depicted in Figure 1(B), the microservice architecture (MSA) emerged to overcome these limitations [25]. The MSA divides an application into small, independently deployable services, allowing each service to be developed, deployed, and scaled independently. Each microservice is responsible for a specific business function, can have its own database, and runs in an isolated containerized environment. This architecture reduces the coupling between services, improves system elasticity, and enables automated deployment and management through orchestration tools such as Kubernetes [12].

2) KUBERNETES OPERATOR

Kubernetes [12] includes a controller [26], one of its core components that enables automated management and operation. The controller continuously monitors the status of specific resources in the cluster, ensuring that the desired state matches the current state. Although the controller automatically reacts to changes in the cluster's status, additional logic is required to manage custom applications or services in detail. To address this need, an operator [27] from the Kubernetes, also known as a custom controller, was proposed, as illustrated in Figure 2. The operator allows custom applications to be managed in the cluster as if they were native Kubernetes resources. It provides the necessary custom resources and controllers to automate complex management tasks and apply application-specific logic. Leveraging the extensibility of the Kubernetes API and the concept of custom resources, the operator can automatically handle the installation, upgrade, backup, and recovery of applications.

The operation of the operator involves several steps, from user intervention to the adjustment of the state through the Kubernetes API, as shown in Figure 2. First, the user modifies a custom resource (CR) [28] that represents the desired state of the application. Then, the operator continuously observes state changes in the cluster by observing these changes or events through the 'Watch' function and initiates actions in response. When an event occurs, such as the creation, update, or deletion of a CR, the operator investigates the current state based on the API information through the 'Reconcile' loop and executes tasks to achieve the desired state. Thus, the operator is essential for managing the cluster and maintaining a stable MSA by automating these actions. This automation reduces the burden on administrators by eliminating the need for direct management of all tasks and significantly improves operational efficiency.

3) SECURITY POLICY

Security policies can elaborately control the operation of container-based services in the Kubernetes cluster and

TABLE 1. Examples of security policy enforcement systems for networks, systems, and clusters.

Category	Security Policy Enforcement System
Network	Kubernetes [12], Cilium [29], Calico [30]
System	KubeArmor [31], Falco [32], Tetragon [33]
Cluster	Kyverno [34], Kubewarden [35]

block malicious behavior, as shown in Figure 3. These policies can be divided into three categories based on their characteristics: network, system, and cluster management. Table 1 shows representative security tools that provide security policy enforcement for each category. First, network policies define communication rules between pods (i.e., groups of containers) or network endpoints within the cluster to efficiently manage traffic and enhance isolation and security between services. For example, a network policy (`net-pol` in Figure 3) allows the `frontend` pod to send packets to port 5959 in the `backend` pod. While the network policies provided by Kubernetes [12] offer basic IP-based and label-based traffic management, Cilium [36] and Calico [37] leverage extended Berkeley Packet Filters (eBPF [38]) to manage traffic between services in more detail (e.g., HTTP, DNS) and with high performance.

For system policies, containers set security rules to block unauthorized access to system resources (e.g., specific files and system calls). For instance, a system policy (`sys-pol` in Figure 3) can prevent any processes in the backend pod from accessing the `/etc/passwd` file. Among various system policies in Kubernetes, KubeArmor Policy [39] can block malicious activities at runtime by preventing access to critical files by malicious processes using eBPF and linux security module (LSM [40]) or by setting rules that block certain critical system calls, thereby strengthening the overall security posture. Similarly, Tetragon [33] and Falco's Policy [32] support security policies that monitor the system calls of specific containers to detect suspicious activities. These systems can capture abnormal behaviors on the system in real time and generate alerts, enabling early detection and response to security incidents.

Lastly, cluster policies are used to define and enforce security rules for containers within a cluster. A cluster policy can mandate that a specific label (`type=online`) must be attached when a pod is created as shown in `cluster-pol` in Figure 3. If a pod is created without this label, the policy automatically detects the discrepancy and either adds the appropriate label or blocks the resource creation. Kyverno Policy [41], which provides such functionality, can automatically verify policy compliance by applying various rules to the lifecycle events of Kubernetes resources, modifying or blocking noncompliant resources. For example, a Kyverno policy can restrict the use of container images to specific image registries. The allowed registries are set through namespace annotations, and for the entire cluster, they are set through `ConfigMap`. A rule can then be

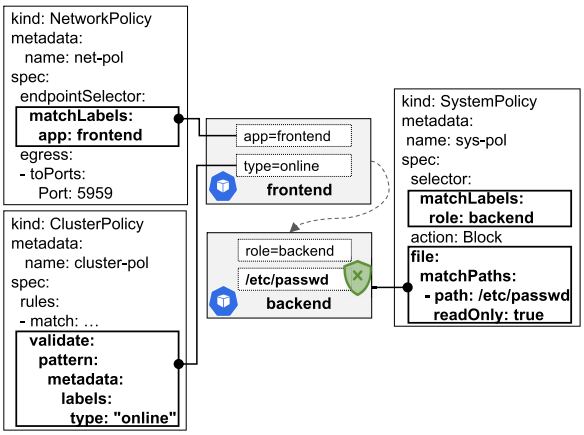

FIGURE 3. Examples of three different types of policy enforcement scenarios.

established to block the use of images from unauthorized registries when creating a pod. This approach enables detailed registry control in a multi-tenant environment.

B. CHALLENGES

Establishing thorough security policies is essential for a secure cloud-native environment. Nevertheless, as the variety of security services expands, numerous challenges arise in policy enforcement. This section investigates four key challenges involved in managing these security policies.

1) C1: COMPLEXITY IN HETEROGENEOUS POLICY MANAGEMENT

In the contemporary cloud-native environment, security systems, whether operating independently or collaboratively, exhibit diverse and complex security requirements. Each system demands its own security measures, posing challenges in the centralized configuration and maintenance of appropriate security policies across all services. Different security tools utilize distinct policy schemas, requiring administrators to learn each policy structure, which makes it difficult to uphold a consistent overall security status due to the disparity among policy languages.

2) C2: SCALABILITY ISSUES IN ADOPTING NEW SYSTEMS

As shown in Table 1, there are various types of systems that can enforce security policies for each category. In the case of network security policies, besides Kubernetes' own security policies, service-specific network security policies can be enforced by various CNIs such as Cilium and Calico. Therefore, when providing a centralized security policy system as suggested in C1, the modification of the management system is inevitable whenever a new security system is introduced. However, it is not realistic to manually modify and distribute the main code each time, such as analyzing the security policy scheme of the newly introduced security system and creating a corresponding security policy. Furthermore, this can act as a major obstacle to the rapid

application of security policies when the service is rapidly expanded, significantly reducing operational efficiency.

3) C3: RISKS OF MISCONFIGURATION

Basically, security systems in the cloud operate independently and have different configuration requirements. If not managed consistently, policy enforcement errors or misconfigurations can occur. Specifically, incorrect security policies may be set for resources that the service, to which the actual security policy applies, does not have. Moreover, the rapid pace of change in the cloud environment introduces additional difficulties in detecting and correcting incorrectly configured and applied policies. Additionally, conflicts with the status of the cluster can arise when updating existing policies or enforcing new ones. Security vulnerabilities caused by such mistakes or human errors can be exploited by attackers, potentially leading to serious security incidents such as data leaks.

4) C4: DIFFICULTIES IN POLICY STATUS TRACKING

In a cloud-native environment where numerous containers are created and destroyed rapidly, the lifecycle of a security policy is closely linked to the lifecycle of a service. As the status of a service continuously changes, the corresponding security policy must also be updated swiftly if needed. However, manually updating the security policies to accurately reflect the dynamically changing service status is a cumbersome task. Additionally, due to the interconnected nature of multiple services and containers in a cloud-native environment, predicting how an updated policy applied to one service will affect another is challenging. The lack of such connection information between the security policies and the resources could incur unintended operations in the cloud-native environments.

III. RELATED WORK

A. CLOUD-NATIVE SECURITY

In recent years, various studies have been conducted to strengthen security in cloud-native environments, focusing on resource optimization [42], [43], [44], network security [45], [46], [47], and system protection [17], [48], [49]. Wang et al. [42] proposed RMiner, a hybrid subsystem that systematically manages invisible shadow resources in environments with corrupted shared states to improve resource utilization. Additionally, Wang et al. [43] introduced DeepScaling, which enhances resource management by predicting CPU utilization and supporting automatic scaling in large-capacity processing microservices. For serverless functions, Li et al. [50] proposed a scheduling technique to maintain performance while reducing resource costs. While these studies contribute significantly to specific aspects of cloud-native security, they often operate in isolation, focusing on individual layers such as resource management or network traffic monitoring. This siloed approach fails to address the need for a comprehensive, multi-layered security strategy. Furthermore, these solutions frequently require integration with specific security tools, leading to compatibility issues and policy fragmentation across different security domains. In the realm of network security, You et al. [45] developed Helios, a hardware-based network security solution using SmartNICs, and Le et al. [46] proposed a mechanism to prevent malicious user attacks by minimizing unnecessary system call exposure in Kubernetes environments. Lim et al. [47] suggested a method to effectively capture container logs by providing a security audit function at the container level. These systems undoubtedly enhance network and system-level security. However, they do not address the critical issue of policy consistency across different security domains. This limitation creates gaps in cross-layer security management, potentially leaving vulnerabilities at the intersections of various security layers. Lastly, in the case of the system security, Song et al. [48] proposed NIMOS, a framework that strengthens container security by analyzing system call sequences to prevent various risks in container environments. Van't Hof and Nieh [49] introduced a security system that protects container data by securing an independent memory space. Ghavamnia et al. [17] introduced Confine, a method that prevents attacks by automatically generating system calls in Docker containers.

Unlike these existing solutions, KubeAegis, addresses the critical need for a unified framework capable of managing heterogeneous security policies across network, system, and cluster levels. It offers a centralized policy management approach that simplifies the enforcement of security policies across all layers, ensuring consistency across different security tools. KubeAegis not only integrates existing security mechanisms but also provides a flexible, adapter-based architecture that can easily incorporate new security tools and policies. This comprehensive approach fills a significant gap in current cloud-native security solutions, offering a more holistic and adaptable security posture for modern, dynamic cloud environments.

B. SECURITY POLICY IN CONTAINERIZED ENVIRONMENTS

Recently, numerous studies have proposed methods for automatic security policy generation [14], [16], [51]. Lumi [51] is a system that translates network policies entered by network operators in natural language into low-level configuration commands and distributes them across the network. This system improves the accuracy of policy translation by utilizing machine learning algorithms and operator feedback. Log2Policy [14] presents a new approach to automatically configure access control between microservices, generating fine-tuned access control rules through a graph generation algorithm based on access logs and request attribute extraction using machine learning. AutoArmor [16] automatically generates necessary access control policies by analyzing the interactions between microservices, using a request extraction mechanism based on static analysis.

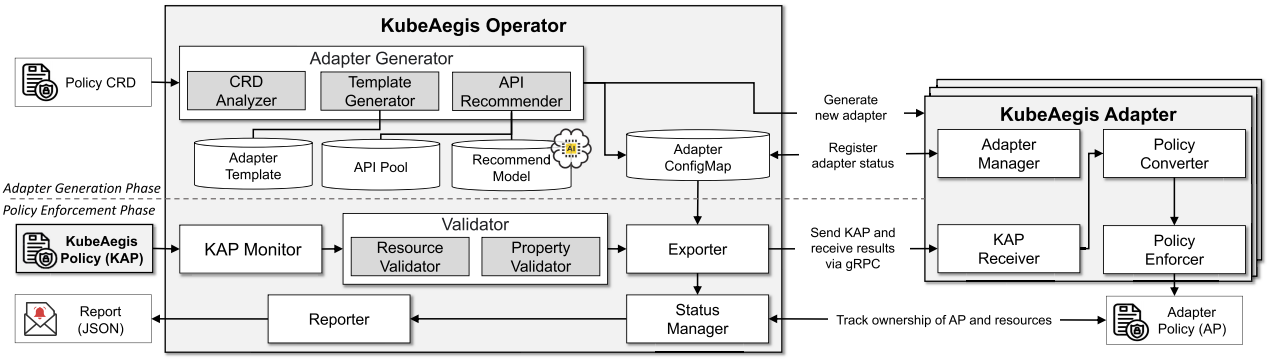

FIGURE 4. Overall architecture and workflow of KubeAegis with three key components: (*i*) KubeAegis Policy (KAP), (*ii*) KubeAegis Operator, and (*iii*) KubeAegis Adapter. Additionally, Our System Includes Two Operational Phases: Adapter Generation and Policy Enforcement.

In addition, various open-source tools have been proposed to effectively manage security policies in cloud-native environments. For example, Otterize [52] allows users to declaratively define their intended access policies and automatically converts them into network policies and Kafka policies [53]. Open Policy Agent (OPA) [54] enables policy-based access control and verification in various environments, including cloud-native environments, and is actively used to express and execute security policies in microservice architectures. Leveraging OPA, GateKeeper [55] verifies requests in a Kubernetes cluster, disallowing unauthorized requests. In the academic field, Verikube [56] has been proposed for the automatic verification of container network policies. It is compatible with Cilium only and features a novel graph structure to minimize memory usage and improve verification speed. Kano [57], [58] proposes an incremental verification with an intent-based verification language and a bit matrix model, presenting a method to automatically identify policy violations and generate a correction plan.

While these tools provide valuable solutions for specific aspects of policy management, they often require users to manually adapt policies for different security tools. Additionally, Moreover, many are specialized for a single type of policy (e.g., network policies), leading to fragmented management. KubeAegis addresses these limitations by offering a unified policy management solution that supports heterogeneous security tools and provides an integrated API recommendation mechanism, streamlining the process of policy adaptation across various tools.

IV. KubeAegis Overviews

This section outlines the design requirements to address the challenges mentioned in Section II-B, which motivate KubeAegis. It also provides a detailed description of the system architecture.

A. Design Requirements

This research stems from the hypothesis that unified management of heterogeneous security policies in cloud-native environments is more effective than separate management approaches. We anticipate that an integrated management approach will enhance policy consistency, prevent conflicts, and enable rapid responses to dynamic environmental changes. During implementation, we face challenges inherent to dynamic cloud environments, as detailed in Section II-B. To address these challenges and validate our hypothesis, we present four key design requirements: unified policy management (§IV-A1), flexible plug-in adapter design (§IV-A2), policy validation (§IV-A3), and ownership tracking for policies (§IV-A4). These requirements are designed to overcome the identified challenges and realize the potential benefits of integrated security policy management in cloud-native environments.

1) R1: Unified Policy Management

The framework should provide a unified policy management system to centrally meet the requirements of multiple security tools and policies. Specifically, it should allow security policies at the system, network, and cluster levels to be managed through a single, unified interface, ensuring consistent policy application and management. Additionally, the unified policy system should support efficient processing of tasks such as creating, modifying, and deleting security policies.

2) R2: Flexible Plug-in Adapter Design

The framework should have a flexible plug-in adapter design that can integrate with various security tools, allowing users to easily incorporate new security policy engines or existing security tools into the system as needed. In other words, it should convert the uniform security policies into a form that specific security systems can understand and then apply the policies through those systems. The adapter-based design enhances scalability, facilitating seamless integration of new security tools as the system evolves. Additionally, a policy-converting API pool should be provided and automatically recommended to facilitate the easy development of plug-in adapters when a new security system is introduced, thereby maximizing the system's flexibility and extensibility.

3) R3: Policy Validation

The framework should include a mechanism to verify the validity of security policies to ensure they are applied effectively and are suitable for the actual environment.

Specifically, it should monitor in real-time for misconfigurations or conflicts, it can respond promptly to any problems, thus preventing policy conflicts in advance, enhancing system reliability, and minimizing the risk of security incidents caused by incorrect policy enforcement.

4) R4: OWNERSHIP TRACKING FOR POLICY

The framework should systematically track and manage the ownership and lifecycle of resources associated with security policies. Since security policies are closely related to dynamically changing resources in the cluster, understanding how associated resources are affected when policies are deleted or changed is crucial. Therefore, the framework should notify administrators of such changes and prevent potential security vulnerabilities caused by the resource changes associated with the security policies. This ownership tracking mechanism enhances visibility into policy enforcement and resource management across the cluster, enabling administrators to maintain a clear overview of the security posture at all times.

B. SYSTEM ARCHITECTURE AND WORKFLOW

This section presents the overall architecture of KubeAegis and explains its components. As illustrated in Figure 4, our framework consists of three main components: KubeAegis Policy (KAP), Operator, and Adapter. Our framework supports two distinct phases. First, the adapter generation phase aims to create new adapters by analyzing the custom resource definition (CRD) and recommending relevant APIs. Second, the policy enforcement phase automates the policy validation and enforcement processes from end to end.

1) KubeAegis POLICY (KAP[1])

It is a unified security policy designed to support consistent security measures within the cluster, addressing security requirements at the system, network, and cluster levels. As depicted in Figure 5, the entire scheme comprises a component for selecting the target resource to which KAP is applied, an actual request rule, and a status that indicates the associated resource and adapter policy (AP). The request rule includes an action and a resource property for each category, and its detailed definition can be found in Appendix.

2) KubeAegis OPERATOR

The operator consists of six key modules: adapter generator, KAP monitor, validator, exporter, status manager, and reporter. The *adapter generator* aims to facilitate the automatic generation of adapter templates by analyzing the CRD of a new security system (i.e., the adapter generation phase in Figure 4) and it comprises three submodules: CRD analyzer, template generator, and API recommender. The analyzer extracts the necessary information from the given CRD, while the template generator creates the adapter templates based on this information. The API recommender calculates the similarity between field descriptions and API methods using a Sentence Robustly Optimized BERT [59][2] Pretraining Approach (SRoBERTa [19], [20]). This model is based on the ideas of Sentence Bidirectional Encoder Representations from Transformers (SBERT [19]) but use the Robustly Optimized BERT Pretraining Approach (RoBERTa [20]) instead of BERT to improve embedding performance. It then uses cosine similarity [60] to suggest appropriate API methods. Subsequently, it communicates with the generated adapter to monitor its latest status.

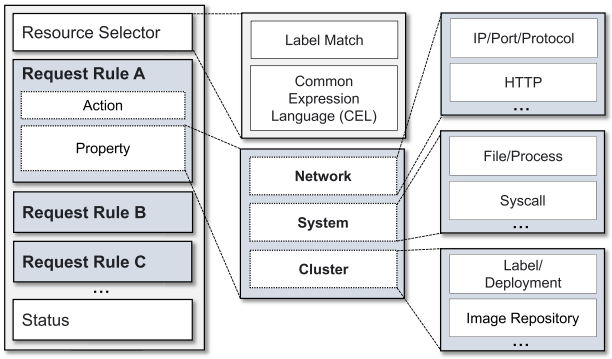

FIGURE 5. Partial views of KAP structure: The request rule involves network, system, and cluster properties.

All other modules, except for the adapter generator, are involved in the policy enforcement phase. First, the *KAP monitor* continuously tracks the creation, deletion, and update events of the KAP. It promptly detects each event and forwards it to the validator to ensure that subsequent tasks can be executed. The *validator* literally verifies the validity of the KAP. It consists of two submodules: the resource validator and the property validator. The resource validator checks the existence and status of the resource referenced by the KAP, while the property validator ensures that the request rules defined by the KAP are valid and that its preconditions are met. This process reviews the network rules, system rules, and cluster rules. If an error is detected, it is recorded, and the cause of the error is conveyed to the administrator.

The *exporter* is responsible for transferring the verified KAP to the appropriate adapter by referencing the adapter ConfigMap. It also notifies the adapter when the KAP is deleted, ensuring that the related adapter policy is removed. The *status manager* updates and manages the status of the KAP. It records the status at each stage of policy creation, verification, and enforcement to maintain current status information. Additionally, it tracks the ownership of resources related to the KAP to minimize the impact on these resources when they are changed or deleted. The *reporter* communicates the policy enforcement results to the user in a JSON format. It also notifies the user when there is a change

[1]In this paper, KubeAegis Policy will be referred to as 'KAP' for short, and due to space limitations, its detailed structure will be appended later.

[2]It stands for Bidirectional Encoder Representations from Transformers.

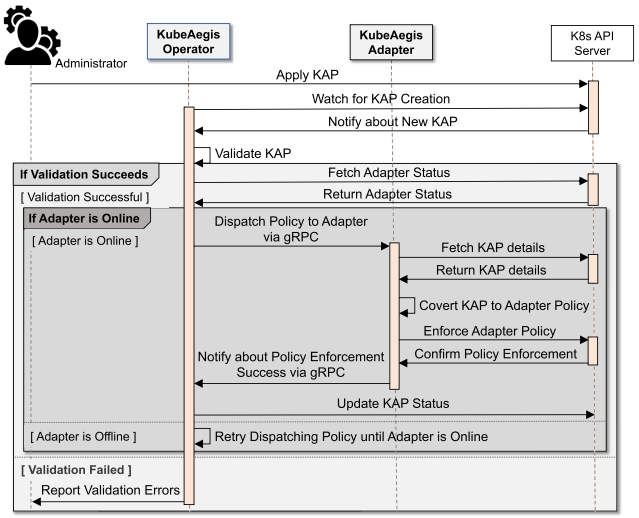

FIGURE 6. Sequence diagram for handling a KAP creation event.

in the resource to which the KAP is applied or when a change in ownership occurs.

C. KubeAegis ADAPTER

The adapter consists of four submodules: adapter manager, KAP receiver, policy converter, and policy enforcer. The *adapter manager* oversees the lifecycle of the adapter, ensuring it is ready to receive KAPs and updating its status accordingly. Additionally, it periodically informs the adapter generator in the operator about the online and offline status. The *KAP receiver* receives the verified KAP from the exporter and sends it to the converter. The *policy converter* then transforms the received KAP into adapter policies (APs) compatible with each security system (e.g., Cilium [29], KubeArmor [31], Kyverno [34]). The *policy enforcer* applies the policies to the cluster and notifies the status manager of the enforcement results to keep the policy status current.

V. KubeAegis DESIGN DETAILS

This section details the features of KubeAegis to satisfy the core design requirements (§IV-A) in the following order: unified policy handling (§V-A), flexible scalability via plug-in adapters (§V-B), fine-grained policy validation (§V-C), and persistent ownership tracking (§V-D).

A. UNIFIED POLICY HANDLING

The KubeAegis operator monitors and immediately handles KAP resource events to centrally manage heterogeneous security policies. The following describes the processing workflow for create, delete, and update events for the KAP, excluding the read operation. Here, we assume the presence of a specific KubeAegis adapter.

1) CREATE EVENT PROCESSING

As shown in Figure 6, the create event begins with the monitor in the operator detecting the creation of a KAP. The created KAP first undergoes a validation process, during which it carefully checks the current status of the cluster and the conditions required by the KAP. For example, it examines whether the resources referenced in the policy exist and whether the attributes required by the policy are met, thereby determining if the policy is suitable for the cluster environment. If the validation fails, an error is returned to indicate a problem with the policy configuration, and the process terminates. Conversely, if the validation is successfully completed, the operator sends the policy to the appropriate adapter via gRPC communication [61]. The adapter receives the policy information, converts it into their own policy, applies it, and notifies the operator that the policy has been successfully implemented. Based on this information, the operator updates the status of the KAP (i.e., ownership tracking) to complete the policy creation process. On the other hand, if the adapter is offline, the operator periodically attempts to send the policy until the adapter comes online.

2) UPDATE EVENT PROCESSING

In the case of the update event, the process involves modifying the existing policy to reflect the changed requirements. The operator retrieves the information of the updated KAP and compares it with the existing policy to identify the changes. Following this, the same policy verification process as in the create event proecssing is performed on the updated KAP. If the verification is successfully completed, the modified policy is delivered to the appropriate adapter, and the subsequent process mirrors that of the create event handling. Notably, multiple security policies can be defined in one KAP. Thus, if a KAP defining three system policies is applied and an update is performed to remove one system policy, the adapter will delete exactly the relevant adapter policy through the update event handling.

3) DELETE EVENT PROCESSING

For the delete event, it involves removing KAPs that are no longer needed in the cluster. The operator immediately detects the deletion event of the KAP and notifies the relevant adapter. The adapter then searches for the adapter policy associated with the deleted KAP through ownership tracking and requests the actual deletion of the policies via the Kubernetes API. The associated policies are then removed from the cluster. After confirming that the policy deletion has been completed successfully, the adapter notifies the operator of the result. Lastly, the operator then performs a final check to ensure that all deletion tasks have been executed correctly, completing the deletion procedure.

B. FLEXIBLE SCALABILITY VIA PLUG-IN ADAPTERS

KubeAegis is designed to effectively manage security policies provided by heterogeneous security systems through a plug-in adapter design. This design enables each adapter to receive and convert these policies into a compatible format

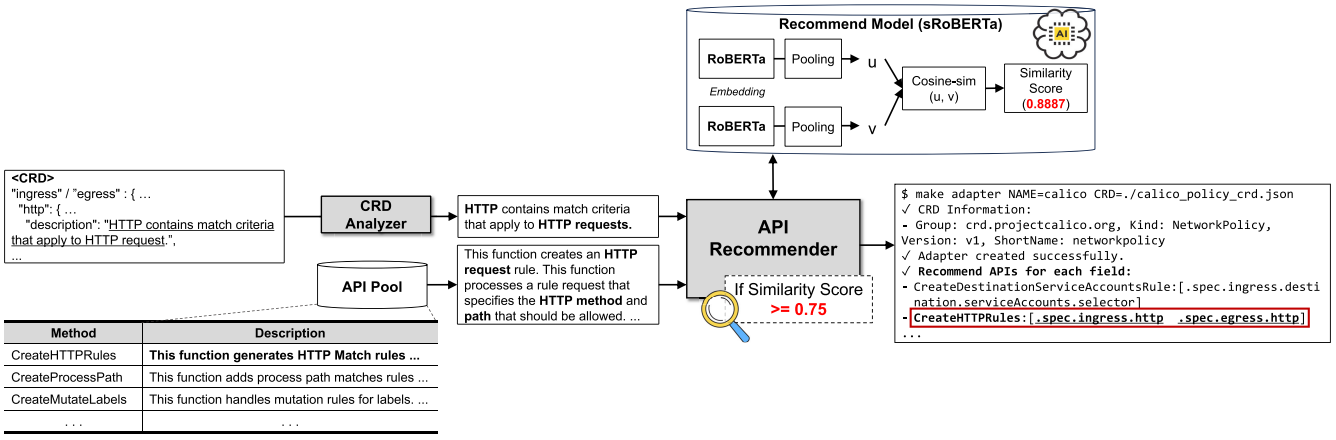

FIGURE 7. Example of the API recommendation process for generating a new adapter.

for their respective systems. However, the extension method for adapters necessitates the development and distribution of a new adapter for each newly introduced security policy enforcement platform, which is inefficient when done manually. Thus, KubeAegis automates the generation of basic adapter code and recommends suitable APIs based on the policy scheme of the adapter systems. As illustrated in Figure 7, the adapter maker module automatically creates the foundational code of the adapter in a template format. The module reads the Custom Resource Definition (CRD) file to extract details such as group, type, version, and field descriptions. It then replicates the predefined adapter template directory, replacing placeholders in the template with actual values derived from the CRD, thereby completing the adapter code. During this process, the necessary packages are automatically imported, and the configuration file, including their network settings, is updated.

After generating the adapter template, KubeAegis provides the API recommendation mechanism to efficiently produce the final adapter code by utilizing the Sentence-RoBERTa(SRoBERTa) model based on each adapter CRD, as shown in Figure 7. The detailed API recommendation steps are described in Algorithm 1. The API recommendation process begins by initializing the SRoBERTa model, a pre-trained language model for natural language processing tasks (line 1), including setting up the hardware. Next, the algorithm calculates embeddings (lines 2-4), where it extracts field descriptions from the adapter CRD and predefined API descriptions. These descriptions are transformed into numerical embedding vectors using the SRoBERTa model, enabling computational analysis. The algorithm then computes the cosine similarity between the encoded descriptions (lines 5-6), generating a similarity matrix S that quantifies semantic relatedness between fields and APIs. The Cosine Similarity ranges from -1 to 1, calculated by measuring the angle between two vectors in a multidimensional space, as shown in line 6. This results in a similarity score $S[i, j]$ for each pair of field and API descriptions, where higher values indicate greater semantic similarity. Additionally, a dictionary R is initialized to store the API recommendations for each field (line 7).

The core API recommendation process is encapsulated in a loop that iterates over each field d_i in the set of field descriptions D. For each iteration, the algorithm performs several steps: It optionally applies a weighted similarity calculation (lines 10-11). This step allows for the incorporation of field importance if required, using a weighting factor α_i. The weighted similarity is calculated as $W[i, j] = S[i, j] \times \alpha_i$ for each API description. If no weighting is necessary, the original similarity scores $S[i, j]$ are used directly as $W[i, j]$. The algorithm then identifies the API with the highest similarity score for the current field (lines 12-13). If this best match score exceeds a predefined threshold τ (set to 0.75 in this case), the field description is associated with the corresponding API in the recommendation dictionary R (lines 14-17). To provide more comprehensive recommendations, the algorithm also stores additional high-scoring matches that exceed the threshold τ, excluding the best match(lines 18-22). This process ensures that each field is carefully analyzed and matched with the most relevant APIs, while also considering potential secondary matches that may be useful. After processing all fields, duplicates are removed from the recommended APIs for each field(lines 23-26). This step ensures that the final recommendations are refined and ordered by their similarity scores. The process then concludes by returning the refined dictionary R of API recommendations (line 27), providing a comprehensive set of suggested APIs for each field in the adapter CRD. This enhanced algorithm leverages the power of pre-trained language models and similarity metrics, while also incorporating optional weighting for field importance. The result is a context-aware API recommendation system that can potentially improve the efficiency and accuracy of adapter code generation, adapting to the specific needs and priorities of different fields within the CRD.

The threshold τ is set to 0.75 as empirically, values greater than this have demonstrated strong similarity between the field description and API description vectors in our

Algorithm 1 API Recommendation Using SRoBERTa With Weighted Similarity

Input: Field descriptions $D = \{d_1, d_2, \ldots, d_m\}$, API method descriptions $A = \{a_1, a_2, \ldots, a_n\}$, threshold $\tau = 0.75$
Output: Recommended APIs for each field R

1. Initialize SRoBERTa model and device
a. Embedding Calculation
- Encode field descriptions:

$$E_D \leftarrow \text{SRoBERTa.encode}(D)$$

- Encode API descriptions:

$$E_A \leftarrow \text{SRoBERTa.encode}(A)$$

b. Cosine Similarity Calculation
- Compute cosine similarity between encoded descriptions:

$$S[i,j] = \frac{E_D[i] \cdot E_A[j]}{\|E_D[i]\| \|E_A[j]\|}$$

2. Initialize Recommendation Dictionary

$$R \leftarrow \{\}$$

3. API Recommendation Process
for *each field* d_i *in* D **do**
  a. Weighted Similarity Calculation (Optional)
  - If a weighting factor is required based on field importance, apply the weighting:

$$W[i,j] = S[i,j] \times \alpha_i$$

  - If no weighting is applied, use $S[i,j]$ directly (i.e., $W[i,j] = S[i,j]$).
  b. Find Best Matching API
  - Identify the index of the maximum similarity score:

$$\text{best\_match\_idx} \leftarrow \arg\max(W[i,:])$$

  - If the best match score exceeds the threshold τ:
  if $W[i, best\_match\_idx] \geq \tau$ **then**
    api_name $\leftarrow A$[best_match_idx]
    R[api_name].append(d_i)
  c. Handle Additional Matches
  - Store additional matches that meet the threshold τ, excluding the best match:
  for *each* w_{ij} *in* $W[i,:]$ **do**
    if $w_{ij} \geq \tau$ *and* $j \neq best\_match\_idx$ **then**
      sub_field_matches.append((w_{ij}, $A[j]$));
4. Remove Duplicates
- Remove duplicates from the recommended APIs:
for *each api_name in* R **do**

$$R[\text{api\_name}] \leftarrow \text{list}(\text{set}(R[\text{api\_name}]))$$

return R

experimentation. This threshold reduces noise and avoids low-relevance matches. If the threshold is too low, inaccurate recommendations increase, while if it is too high, relevant recommendations may be missed. Thus, 0.75 is chosen as a balance point to provide reliable matching while including sufficiently relevant recommendations. Through this process,

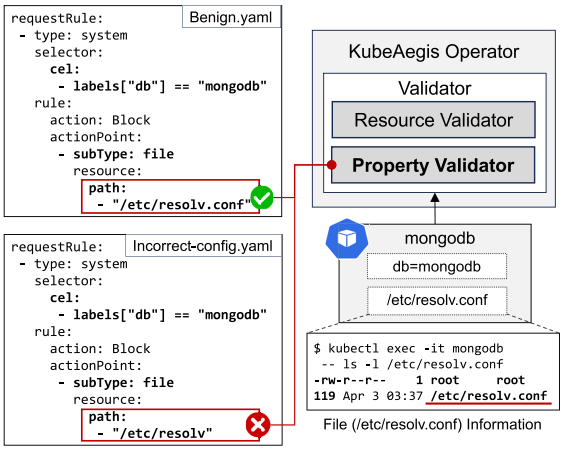

FIGURE 8. Example of property validation for KAP.

API recommendations are made based on the similarity between field and API description.

C. FINE-GRAINED POLICY VALIDATION

The KAP policy validation consists of two main parts: the resource validation, which checks the existence and basic context of the resource to be applied by the KAP, and the property validation, which checks the detailed properties of the resource specified in the KAP. If either validation fails, the operator does not send the KAP policy to its relevant adapter and the validator immediately reports any errors found to the administrator.

1) RESOURCE VALIDATION

The resource validator in KubeAegis operator, which is responsible for resource validation, verifies the actual resources used in the cluster. In other words, it checks whether the resources to which the KAP will be applied actually exist in the cluster. For example, as shown in Figure 8, the expression `labels["db"]=="mongodb"` in the `Benign.yaml` file reflects the intention that a pod with the `db` label of `mongodb` should exist. Thus, the validator searches for the pod with the corresponding labels to confirm whether this intention is met. If the pod with the `db=mongodb` label exists, the policy passes validation and is deemed suitable. Conversely, in `Incorrect-config.yaml`, if no pod with the `db=mongodb` label exists, the validation fails, which indicates that the policy does not match the actual cluster environment, and the administrator is notified with the errors.

2) PROPERTY VALIDATION

In addition to the resource validation, the property validation checks the detailed properties of the target resource to which the KAP will be applied. First, for the network policies, since a KAP with incorrect network properties may lead to network vulnerabilities, it should verify whether the port defined by

the KAP is listening on the target resource and whether the CIDR block points to a valid address range. Additionally, it performs protocol verification procedures such as TCP or UDP. For the system policies, it verifies the executable status of scripts, commands, and system calls within the cluster, as well as the existence and read-only status of target files. Finally, in the cluster policy, it checks whether the annotations and labels of the pods satisfy the requirements and verifies the existence of the image.

For example, as shown in Figure 8, the property validator process verifies the status of the `/etc/resolv.conf` file that must exist in the pods within the actual cluster. In `Benign.yaml`, the policy is approved by confirming that the `/etc/resolv.conf` file exists and is readable. On the other hand, in `Incorrect-config.yaml`, the policy is rejected because the relevant file, such as `/etc/resolv`, does not exist or has an incorrect file path. In this manner, the property validator checks whether all properties of the KAP are satisfied within the cluster and notifies the administrator of any errors that occur during the validation process through logging.

D. PERSISTENT OWNERSHIP TRACKING

As the unified security policy, the KAP has an ownership of the adapter policy (AP) that is actually enforced into the cluster. In the same way, the AP has an ownership of specific resources (e.g., pods, deployments) to which it is applied. Consequently, one KAP instance can be managed through direct or indirect associations with the APs and the resources. This ownership and lifecycle are systematically controlled leveraging the owners reference mechanism from Kubernetes [62], where ownerReferences is specified in the metadata field of the resource to register the corresponding owner. It enables continuous tracking of the status changes in the owned resource, such as deletions or modifications, and notifies the owning KAP. Upon creation, a KAP is automatically linked to related resources, including the AP and the ownership relationships are adjusted to reflect the latest status whenever it is updated. If a KAP is deleted, all associated relationships with the resources and the APs are also removed.

Conversely, if a resource associated with the KAP is deleted or modified, the AP detects the change, revalidates the current state of the resource, updates the ownership status, and notifies the operator along with the relevant KAP information. For instance, as illustrated in Figure 9, initially, the KAP owns the AP, which in turn owns Pod 1 and Pod 2, both labeled with `app=webapp`. If Pod 1's label changes to `svc=storage`, the AP releases its ownership of Pod 1 and informs the operator, including the KAP details. The resource whose ownership has changed is no longer managed by the AP. Through this mechanism, KAP dynamically manages the ownership relationships between resources and policies within the cluster, ensuring accurate tracking of which KAPs are involved with specific APs and resources.

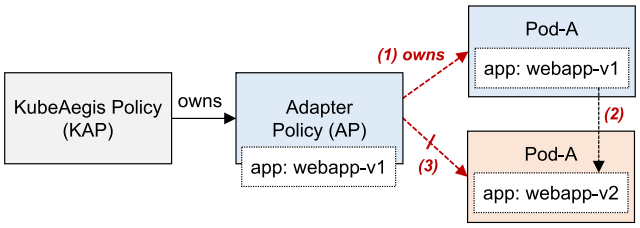

FIGURE 9. Example of ownership release through ownership tracking: Pod-A Owned by the AP (1) Has its label modified (2), Resulting in the release of ownership of Pod-A (3).

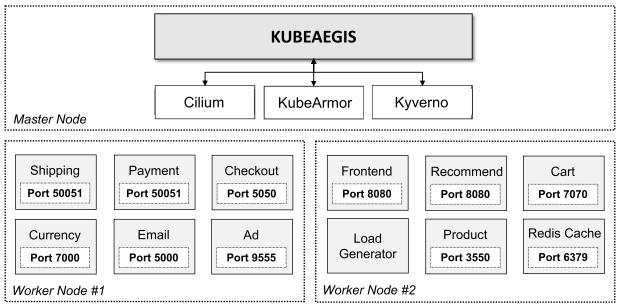

FIGURE 10. Overview of test environments: KubeAegis and three different adapters deployed on the master node.

VI. EVALUATION

This section demonstrates the effectiveness of our system in managing unified security policies within a real-world environment, with a focus on functionality, coverage, and performance.

A. IMPLEMENTATION

We have implemented our system using a combination of Go and Python to verify its feasibility and effectiveness. KubeAegis currently includes policy adapters for four well-known open-source security systems: Cilium [29], Calico [30], KubeArmor [31], and Kyverno [34]. These adapters enable KubeAegis to manage the actual security policies of these systems. Additionally, to support new adapters, we implemented an API recommendation mechanism using the SRoBERTa model [19]. This model embeds CRD field descriptions and API descriptions that we provided, then calculates cosine similarity to recommend appropriate APIs for converting KAP into AP. In summary, to support the design features described in Section IV-A, we implemented the operator and four types of adapters, comprising about 10.9K lines of code.

B. TEST ENVIRONMENTS

This experimental environment was built on a server equipped with an Intel Xeon Silver 4210R CPU, 256GB RAM, and a 2TB HDD. A Kubernetes cluster was established using three Ubuntu 22.04 virtual machines (VMs). Figure 10 illustrates the container configuration within the cluster for this experiment. One VM operates as the master node, while the other two function as worker nodes. Especially,

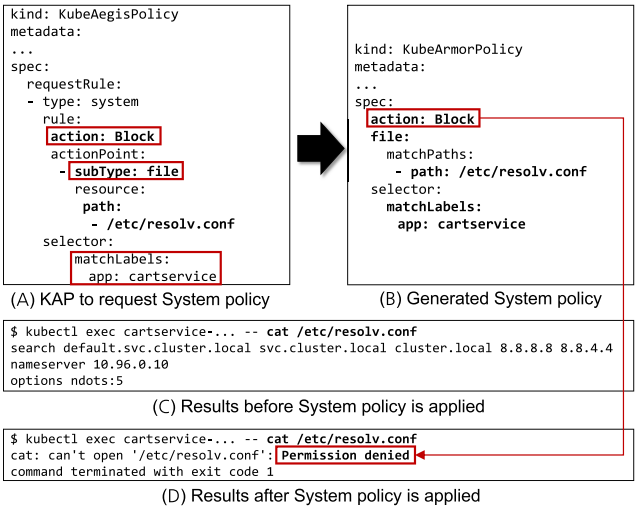

FIGURE 11. Results of KAP conversion and enforcement.

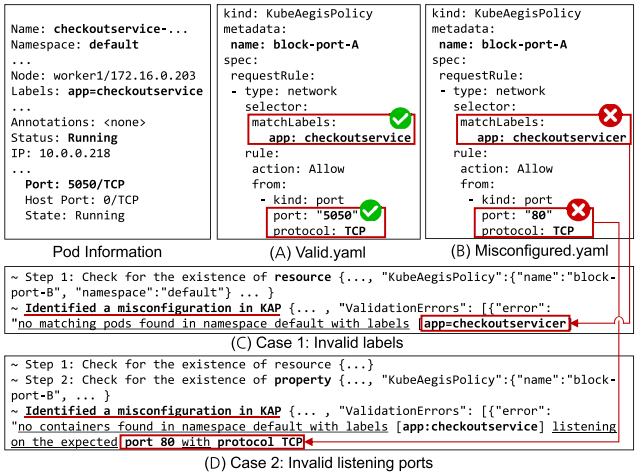

FIGURE 12. Results of KAP validation scenarios: Valid (A) and Misconfigured (B).

containerd [63] is used as the container runtime, and communication between the containers is facilitated by the Cilium overlay network. For testing, the Online Boutique [64] microservice was used as the test cloud native application. This e-commerce demo application provides functionalities for users to browse products, add items to the shopping cart, and complete purchases. The application comprises 11 independently deployed and managed services (i.e., pods). Among the services, the load generator continuously sends requests to simulate actual user shopping behavior, enabling system load testing and performance evaluation. KubeAegis runs on the master node, with Cilium or Calico, KubeArmor, and Kyverno systems, while the applications are distributed across the two worker nodes.

C. FUNCTIONAL CORRECTNESS

1) BASIC POLICY CONVERSION

The most fundamental function of our system is to accurately translate the KAP security policy to the target AP. To evaluate this, a system security policy designed to prevent DNS manipulation attacks is created as a KAP and then enforced, as illustrated in Figure 11. This security policy disallows access to the `/etc/resolv.conf` file for all processes within the container. As shown in Figure 11(A), the policy is enforced that restricts access to the `/etc/resolv.conf` file of the `app=cartservice` pod. This policy is immediately converted to the KubeArmor policy, as shown in Figure 11(B), and enforced in the cluster. Before applying this policy, the DNS configuration file could be freely viewed using the `cat /etc/resolv.conf` command in the `cartservice` pod, as depicted in Figure 11(C). However, after the policy is applied, attempting to access the file results in the 'Permission denied' message, as shown in Figure 11(D). This demonstrates that the KAP was successfully converted to the AP and enforced, confirming that the policy effectively restricted access as intended.

2) RESOURCE AND PROPERTY VALIDATION

In this evaluation, we assess the policy validation function for resource and property targeting network security policies. As shown in Figure 12, the `app=checkoutservice` pod provides a service through port 5050/TCP, and two network KAPs are prepared for testing based on this setup. One is configured with the correct label, port, and protocol according to the current environment, as shown in Figure 12(A), while the other, shown in Figure 12(B), is configured with a non-existent pod and incorrect properties.

The first case is for the resource validation that the KAP will be applied to, so we tried to enforce a KAP policy on a resource with a label that does not exist in the current cluster due to a simple typo (i.e., not checkoutservice, but checkoutservicer). However, since the corresponding resource does not exist, the policy creation process is halted, and the error information is promptly reported through the log, as shown in Figure 12(C). Second, we verify that the properties written in each policy are correctly configured for the property validation. In this case, it checks whether the port specified in the network policy is actually the service port used by the pod. As shown in Figure 12(D), our system accurately detected the misconfiguration (i.e., `Incorrect-config.yaml`) by identifying a port number that the resource is not servicing and provided the correct feedback. As a result, we confirmed that KubeAegis accurately judges the validity of the policy resource, identifies incorrect settings, and provides appropriate feedback.

3) OWNERSHIP TRACKING

Figure 13 illustrates how our system effectively operates with centralized ownership tracking. First, we create a KAP called `kap-dns-manipulate` (A) and track and manage the KubeArmorPolicy (i.e., adapter policy, AP), which is derived from the KAP (B). By setting the name and UID information of the KAP in the `metadata.ownerReferences` field

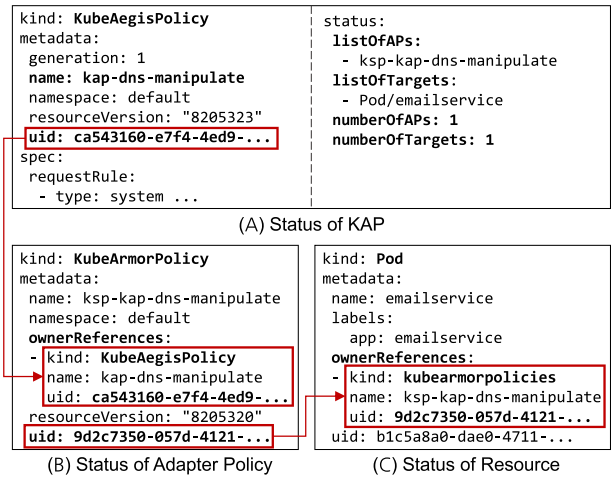

FIGURE 13. Results of ownership tracking linked by KAP.

of the AP, we confirm that the dependency on the KAP has been properly established. Similarly, we can verify that the resource to which the AP is applied has a dependency on the AP, and that ownership is derived from the KAP. Additionally, as shown in `kap-dns-manipulate` (A), when the user queries the KAP using the provided command, all the APs and their resources that have ownership in the cluster are displayed at a glance. This ensures that when a KAP is deleted or modified, the dependent APs are properly updated or removed.

D. COVERAGE

1) POLICY RULE COVERAGE

Table 2 shows that the KAP provided by our system covers a wide range of rules from a security perspective for each network, system, and cluster. First, in terms of the network policy, the KAP supports traffic control rules based on labels, CIDR, port, protocol, HTTP, and FQDN in L3, L4, and L7 respectively. Therefore, considering Cilium, which supports L3, L4, and L7 rules, and Calico, which supports rules except for FQDN, KAP can control both network policy engines as the adapters. Second, for the system security, the KAP can cover KubeArmor and Tetragon, which are representative system security engines in the cloud-native environments, because it supports the detailed rules that block or trace file access, process execution, network and system calls. Lastly, the KAP provides essential security rules for the cluster, addressing functions such as resource modification and image verification, supported by Kyverno. Specifically, the KAP fully supports functionality for mutation, validation, and image verification. However, it only partially supports generate and cleanup functions, as these involve numerous features that are not directly relevant to security. In summary, the KAP has the ability to manage and apply various security rules in an integrated manner across networks, systems, and clusters. This enables more efficient and consistent security management by integrating the functions provided by existing individual policy engines.

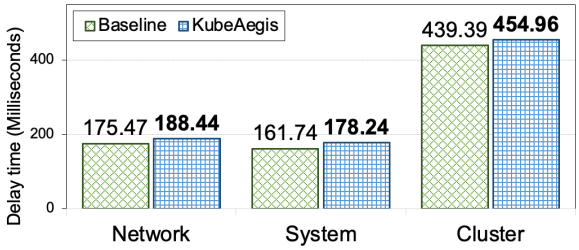

FIGURE 14. Comparison of delay times between direct enforcement and KubeAegis.

E. PERFORMANCE

1) POLICY CONVERSION DELAY

Since our system includes an additional layer to manage various security enforcement systems, there is an inevitable delay in converting the KAP compared to directly enforcing policies for each adapter. To evaluate the extent of this delay, we compared the delay time between directly applying APs and using our system. We measured the time in milliseconds taken to input and apply files where each security policy was defined. This process was repeated 100 times using a shell script, and the average delay time was derived by analyzing the collected delay time data. The experimental results are shown in Figure 14. For the network policies (Cilium), the average delay time was 175.47 milliseconds when applied directly and 188.44 milliseconds when enforced through our system, representing an additional delay of approximately 7.4%. Using the same evaluation method, the system policies (KubeArmor) showed an average delay time increase of approximately 10%, while the cluster policies (Kyverno) indicated an increase of approximately 3.5%.

Overall, an increase of 7% to 10% in delay time was observed when using KubeAegis, but the impact on the user experience was minimal. These results suggest that policy enforcement through our system incurs a slight increase in delay time compared to direct application methods, but it is within an acceptable range considering the advantages of ease of management, consistency of policy application, and reduction of errors.

2) API RECOMMENDATION PERFORMANCE

Our system provides a conversion API pool that translates KAPs to each AP to support the new security policy enforcement system. As shown in Table 3, a total of 67 APIs are available for networks, systems, and clusters. The table displays the number of APIs used and the lines of code added to create each adapter. The basic adapter code that communicates with the the operator is automatically generated (328 lines) using the kubeaegis-sample template. Consequently, the user only needs to modify the converter that translates KAPs to the corresponding APs. For instance,

TABLE 2. Summary of security policy coverage in three categories (Network, System and Cluster) by KubeAegis KAP. ○ means fully supported, and △ means partially supported.

Coverage	Network Policy						System Policy					Cluster Policy				
	L3		L4		L7		File	Process	Network	Capability	Syscall	Generate	Mutate	Validate	VerifyImage	Cleanup
	Label	CIDR	Port	Protocol	HTTP	FQDN										
Kubernetes [12]	○	○	○	○												
Cilium [29]	○	○	○	○	○	○										
Calico [30]	○	○	○	○	○											
KubeArmor [31]							○	○	○	○	○					
Tetragon [33]							○	○	○	○	○					
Kyverno [34]	○	○	○	○								○	○	○	○	○
KubeAegis	○	○	○	○	○	○	○	○	○	○	○	△	○	○	○	△

TABLE 3. Summary of adapter generation overheads and API Recommendation Metrics: Accuracy, Precision, Recall, and F1 Score.

Adapter Name	Code Line (Call Line)	#Recomm. APIs	Acc.	Prec.	Rec.	F1.
Template	328 (-)	-	-	-	-	-
Cilium [29]	378 (50)	23	0.81	0.88	0.91	0.86
Calico [30]	389 (61)	21	0.82	0.86	0.95	0.90
KubeArmor [31]	357 (29)	11	0.80	0.80	1.00	0.88
Tetragon [33]	344 (16)	5	0.83	1.00	0.83	0.90
Kyverno [34]	385 (57)	7	0.87	1.00	0.87	0.93

in the case of the Cilium adapter, 23 APIs were recommended from the API pool based on the Cilium CRD analysis results, and 50 lines of code were added to the basic template, resulting in a total of 393 lines of code, which means the adapter was created with minimal effort. In conclusion, users only need to add the necessary conversion code to the basic adapter template, thereby saving time and effort required for writing rule codes.

In addition, Table 3 presents the results of evaluating the API recommendation performance of our system based on the accuracy, precision, recall, and F1 score indices. This evaluation determines whether the API with the highest similarity for each rule is used as the actual conversion API. For Cilium, the system demonstrates a high precision of approximately 95% and a relatively high recall, indicating that a significant portion of the actual APIs match the recommended APIs. For the Calico and KubeArmor adapters, the results showed relatively lower values compared to Cilium, but they are still at a reasonable level. Similarly, for Tetragon, while most of the actual APIs were recommended, the precision was slightly lower due to some incorrectly recommended values. Finally, the Kyverno adapter exhibited high performance indices, showing that most of the recommended APIs were accurate and included all actual APIs. As a result, we can confirm that the system can recommend APIs with mostly high accuracy for various CRDs.

VII. LIMITATION AND DISCUSSION

Like other research works, our system has some limitations and requires improvements to implement more intelligent and comprehensive security policy management. In this section, we describe the limitations of the current design and propose enhancements to improve the capabilities of KubeAegis in various aspects of security policy management and enforcement.

A. AUTOMATIC POLICY GENERATION WITH LLM

KubeAegis currently requires users to create a KAP (unified security policy) following a specific format. Although the KAP structure is intuitive and simple in YAML format, users still need to overcome a learning curve to create such security policy. Therefore, there is room for improvement in automatically reflecting the intentions of the users in a more natural manner. To this end, we can utilize the Large Language Model (LLM) [65], which has been actively studied recently. Specifically, we plan to introduce a mechanism that analyzes the user's security intentions described in unstructured natural language and automatically converts them into an actual policy for our system. This approach will enable users to manage complex security policies by expressing their intentions in natural language without needing to learn a new policy language. Furthermore, we can consider a methodology that automatically infers and generates security policies by analyzing the resource configuration files deployed in the target cluster, eliminating the need for users to explicitly write their intentions.

B. SUPPORTING MULTI-CLUSTER/CLOUD ENVIRONMENTS

Our system currently operates in a single cluster. However, due to reasons such as availability and security, the adoption of multi-cluster setups, the integration of heterogeneous cloud platforms (i.e., multi-cloud), and hybrid clouds is increasing. Consequently, the complexity of security policy management is significantly rising, leading to numerous misconfiguration incidents. Therefore, a methodology that understands the challenges of security policy management in such complex environments and automatically maintains and manages policy consistency is needed. To address this, our system plans to evolve into a more comprehensive centralized policy management system. This will involve deploying lightweight agents in each distributed heterogeneous cluster and cloud platform to collect the context of the security policies.

C. SEMANTIC CONFLICT RESOLUTION

Although our system provides the resource and property validation mechanisms to minimize the conflicts with the current state of the cluster to which the security policy is applied, it has limitations in handling the semantic conflicts

between the policies. For example, a network security policy may allow communication between the specific services, while a system security policy may block the protocol and port number used by such service at the system call level. In such situations where different security policies conflict, one policy may nullify the intent of another, resulting in a security vulnerability. To address this issue, several advanced analysis algorithms can be employed. For instance, in the case of our system, we plan to propose a methodology that detects conflicts by constructing a graph with each service point or network information as a node and analyzing the reachability between the nodes. Additionally, we intend to introduce a feature that automatically assigns priorities and enforces the most critical policies first in the event of a conflict as a post-response mechanism.

VIII. CONCLUSION

In this paper, we propose KubeAegis, a novel security policy framework tailored for cloud-native environments. Our system reduces the complexity of heterogeneous policy management by unifying configuration methods and policy languages across various security tools. Concurrently, it offers flexible scalability for new security systems through API recommendation mechanisms. Notably, the proposed policy pre-verification and automatic resource tracking functions prevent failures caused by incorrect policy enforcement. This paper is the first to highlight the necessity of integrated policy management and verification, issues that have not been thoroughly studied before. KubeAegis offers a valuable reference implementation applicable to newer security systems, and we anticipate it will enhance both the convenience and security of container environment management.

APPENDIX

KubeAegis POLICY (KAP) SPECIFICATION

KubeAegis is a unified security policy designed to support consistent security measures within a cluster, satisfying system, network, and cluster-level security requirements. Listing introduces the complete schema of the KubeAegis Policy(KAP):

```
apiVersion: security.kubeaegis.com/v1
kind: KubeAegisPolicy
metadata:
  name: [policy name]
  namespace: [namespace name]
spec:
  enableReport: [true|false]
  requestRule:
    - type: [network|system|cluster]
      selector:
        kind: [pod|namespace
               |service|deployment]
        namespace: [namespace name]
        matchLabels:
        [key1]: [value1]
        cel:
         - [cel expression]
      rule:
        action: [Allow|Block|Log|
                 |Trace|Enforce|Audit]
      from:
        - kind: [endpoint|entities|namespace|
                 |serviceAccounts|cidr|port
                 |protocol|fqdns]
          labels:
            - [key1]: [value1]
          args: [<arg1>, <arg2>, \ldots]
          port: [port number]
          protocol: [TCP|UDP|ICMP]
      to:
        - kind: [endpoint|namespace|
                 |serviceAccounts|entities
                 |cidr|port|protocol|fqdns]
          labels:
            - [key1]: [value1]
          args: [<arg1>, <arg2>, \ldots]
          port: [port number]
          protocol: [TCP|UDP|ICMP]
      actionPoint:
        - subType: [http|file|process|network|
                    |syscalls|capabilities|
                    |generate|mutate|validate|
                    |verifyImage|cleanup]
          resource:
              path: [resource path]
              pattern: [pattern]
              kind: [resource kind]
              filter:
                - condition: [any|all]
                  key: [filter key]
                  operator: [Equals|In
                             |NotIn|Exists]
                  value: [filter value]
              details:
                - [key1]: [value1]
                - [key2]: [value2]
status:
  status: [policy enforcement status]
  lastUpdated: [last update time]
  numberOfAPs: [number]
  listOfAPs: [<name1>, <name2>, \ldots]
  numberOfTargets: [number]
  listOfTargets: [<name1>, <name2>, \ldots]
```

The start of a KAP defines basic information such as `apiVersion`, `kind`, and `metadata`. `apiVersion` and `kind` are the same in all cases, and `metadata` contains the name of the policy and namespace. Next, the `enableReport` in the `spec` specifies whether the policy should generate reports.

Then, `requestRule` consists of `selector` that specifies the targets to which the policy applies, the `type` of policy (network, system, cluster), and the `rule` to request. `requestRule` is designed to define in detail which policy is being requested and is able to specify multiple request rules simultaneously. `selector` specifies the targets to which the policy will be applied based on a label via `selector.matchLable`, and if necessary, you can use Common Expression Language (CEL) in `selector.cel` to define more complex selection criteria. In addition, `selector.kind` enables specifying resource types (e.g. Pod, Namespace, Service, Deployment, etc.) to select targets. If you only set `resource.kind` and do not specify a label, the policy applies to all objects of that resource type.

`rule` consists of an `action` (Allow, Block, Log, Trace, Enforce, Audit), traffic direction (`from`, `to`), and

`actionPoint` that defines the specific action the policy will take. `actionPoint` defines what the policy will do and can be broken down into various subtypes at the network, system, and cluster levels. Each `subtype`'s details are simply defined in `actionPoint.resource`. Network-level KAPs use to and from to define the direction of communication between network entities. You can specify resources here such as endpoint, entity, FQDN, etc. and include labels or additional arguments for the resource as needed. Additionally, you can specify a port number (`port`) and `protocol`, and by specifying http in `subType`, you can specify methods in `resource.details` subset: [GET, POST] to define a detailed request. At system level, `subtype` such as File, Process, Network, Capabilities, and Syscalls control interactions with system resources. For example, in File, you can make a request by defining a file path(`resource.path`) and writing something like 'readOnly: true' in `resource.details`. Finally, at the cluster level, `subtype` like generate, mutate, validate, verifyImage, and cleanup enable you to manage cluster-wide resources and policies. For instance, if you select mutate, you can make a request by specifying the resource you want to modify in `resource.kind` and writing something like label: app = nginx in `resource.details`.

Lastly, `status` contains the status and last update information for the policy. This enables tracking of the policy's enforcement status and change history, which is useful for determining if the policy is current, enforced, or needs to be updated. The `numberOfAPs` shows the number of adapter policies (APs) associated with the KAP, and the `listOfAPs` shows a list of the names of the adapter policies (APs). In addition, `numberOfTargets` and `listOfTargets` indicate the number and list of target resources to which the policy is currently applied.

REFERENCES

[1] CloudZero. (2024). *The Cloud Cost Playbook—101 Shocking Cloud Computing Statistics (Updated 2024)*. [Online]. Available: https://www.cloudzero.com/blog/cloud-computing-statistics/

[2] TechPriceCrunch. (2024). *26 Relevant Cloud Adoption Statistics in 2024*. [Online]. Available: https://techpricecrunch.com/blog/cloud-adoption-statistics/

[3] R. Hat. (2024). *Kubernetes Adoption, Security, and Market Trends Report 2024*. [Online]. Available: https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview

[4] C. Software. (2024). *2024 Cloud Security Report*. [Online]. Available: https://engage.checkpoint.com/2024-cloud-security-report

[5] S. Khan, I. Kabanov, Y. Hua, and S. Madnick, "A systematic analysis of the capital one data breach: Critical lessons learned," *ACM Trans. Privacy Secur.*, vol. 26, no. 1, pp. 1–29, Feb. 2023.

[6] B. Jabiyev, O. Mirzaei, A. Kharraz, and E. Kirda, "Preventing server-side request forgery attacks," in *Proc. 36th Annu. ACM Symp. Appl. Comput.*, Mar. 2021, pp. 1626–1635.

[7] (2024). *Amazon Web Services*. [Online]. Available: https://aws.amazon.com/?nc1=h_ls

[8] K. Jayasinghe and G. Poravi, "A survey of attack instances of cryptojacking targeting cloud infrastructure," in *Proc. 2nd Asia–Pacific Inf. Technol. Conf.*, Jan. 2020, pp. 100–107.

[9] Microsoft. (2024). *Microsoft Azure: Cloud Computing Services*. [Online]. Available: https://azure.microsoft.com/en-us

[10] MITRE. (2023). *CVE-2023-23383*. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-23383

[11] G. E. Rodríguez, J. G. Torres, P. Flores, and D. E. Benavides, "Cross-site scripting (XSS) attacks and mitigation: A survey," *Comput. Netw.*, vol. 166, Jan. 2020, Art. no. 106960.

[12] Kubernetes. (2024). *Kubernetes*. [Online]. Available: https://kubernetes.io/

[13] J. Devanesan. (2022). *Automation Key to Keeping Applications Secure in the Cloud-Native Era*. [Online]. Available: https://techhq.com/2022/04/automation-key-to-keeping-applications-secure-in-the-cloud-native-era/

[14] S. Xu, Q. Zhou, H. Huang, X. Jia, H. Du, Y. Chen, and Y. Xie, "Log2Policy: An approach to generate fine-grained access control rules for microservices from scratch," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2023, pp. 229–240.

[15] S. Zhang, S. Li, P. Chen, S. Wang, and C. Zhao, "Generating network security defense strategy based on cyber threat intelligence knowledge graph," in *Proc. Int. Conf. Emerg. Netw. Archit. Technol.* Cham, Switzerland: Springer, 2022, pp. 507–519.

[16] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic policy generation for inter-service access control of microservices," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 3971–3988.

[17] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *Proc. 23rd Int. Symp. Res. Attacks, Intrusions Defenses (RAID)*, 2020, pp. 443–458.

[18] Y. Li, C. Huang, L. Yuan, Y. Ding, and H. Cheng, "ASPGen: An automatic security policy generating framework for AppArmor," in *Proc. IEEE Int. Conf Parallel Distrib. Process. Appl., Big Data Cloud Comput., Sustain. Comput. Commun., Social Comput. Netw. (ISPA/BDCloud/SocialCom/-SustainCom)*, Dec. 2020, pp. 392–400.

[19] H. Choi, J. Kim, S. Joe, and Y. Gwon, "Evaluation of BERT and Albert sentence embedding performance on downstream NLP tasks," in *Proc. 25th Int. Conf. Pattern Recognit. (ICPR)*, Jan. 2021, pp. 5482–5487.

[20] S. Minaee, T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, and J. Gao, "Large language models: A survey," 2024, *arXiv:2402.06196*.

[21] S.-Y. Huang, C.-Y. Chen, J.-Y. Chen, and H.-C. Chao, "A survey on resource management for cloud native mobile computing: Opportunities and challenges," *Symmetry*, vol. 15, no. 2, p. 538, Feb. 2023.

[22] M. Chauhan and S. Shiaeles, "An analysis of cloud security frameworks, problems and proposed solutions," *Network*, vol. 3, no. 3, pp. 422–450, Sep. 2023.

[23] T. Theodoropoulos, L. Rosa, C. Benzaid, P. Gray, E. Marin, A. Makris, L. Cordeiro, F. Diego, P. Sorokin, M. D. Girolamo, P. Barone, T. Taleb, and K. Tserpes, "Security in cloud-native services: A survey," *J. Cybersecurity Privacy*, vol. 3, no. 4, pp. 758–793, Oct. 2023.

[24] A. Y. Wong, E. G. Chekole, M. Ochoa, and J. Zhou, "Threat modeling and security analysis of containers: A survey," 2021, *arXiv:2111.11475*.

[25] Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, and P. Clarke, "Decomposition of monolith applications into microservices architectures: A systematic review," *IEEE Trans. Softw. Eng.*, vol. 49, no. 8, pp. 4213–4242, Aug. 2023.

[26] Kubernetes. (2023). *Controller Pattern*. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/controller/

[27] Kubernetes. (2024). *Kubernets—Operator Pattern*. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[28] Kubernetes. (2024). *Kubernetes—Custom Resources*. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[29] Cilium. (2024). *Cilium—EBPF-Based Networking, Observability, Security*. [Online]. Available: https://cilium.io/

[30] Tigera. (2024). *Project Calico*. [Online]. Available: https://www.tigera.io/project-calico/

[31] KubeArmor. (2024). *Kubearmor—Runtime Security Enforcement*. [Online]. Available: https://kubearmor.io/

[32] T. F. Project. (2024). *Falco*. [Online]. Available: https://falco.org/

[33] C. Tetragon. (2024). *Tetragon—EBPF-Based Security Observability and Runtime Enforcement*. [Online]. Available: https://tetragon.io/

[34] Kyverno. (2024). *Kyverno*. [Online]. Available: https://kyverno.io/

[35] Kubewarden. (2024). *Kubewarden—Kubernetes Dynamic Admission at Your Fingertips*. [Online]. Available: https://www.kubewarden.io/

[36] Cilium. (2024). *Cilium—Overview of Network Policy*. [Online]. Available: https://docs.cilium.io/en/latest/security/policy

[37] Calico. (2024). *Calico Policy*. [Online]. Available: https://docs.tigera.io/calico/latest/network-policy/get-started/calico-policy/

[38] EBPF. (2024). *EBPF—Dynamically Programs the Kernel for Efficient Networking, Observability, Tracing, and Security*. [Online]. Available: https://ebpf.io/
[39] KubeArmor. (2024). *Kubearmor-Policy SPEC for Containers*. [Online]. Available: https://docs.kubearmor.io/kubearmor/documentation/security_policy_specification
[40] T. L. K. Archives. (2024). *Linux Security Module Usage*. [Online]. Available: https://www.kernel.org/doc/html/v4.19/admin-guide/LSM/index.html
[41] Kyverno. (2024). *Policies | Kyverno*. [Online]. Available: https://kyverno.io/policies/
[42] X. Wang, H. He, Y. Li, C. Li, X. Hou, J. Wang, Q. Chen, J. Leng, M. Guo, and L. Wang, "Not all resources are visible: Exploiting fragmented shadow resources in shared-state scheduler architecture," in *Proc. ACM Symp. Cloud Comput.*, Oct. 2023, pp. 109–124.
[43] Z. Wang, S. Zhu, J. Li, W. Jiang, K. K. Ramakrishnan, Y. Zheng, M. Yan, X. Zhang, and A. X. Liu, "DeepScaling: Microservices autoscaling for stable CPU utilization in large scale cloud systems," in *Proc. 13th Symp. Cloud Comput.*, Nov. 2022, pp. 16–30.
[44] Y. Wu, H. Wu, D. Luo, Y. Xu, Y. Hu, W. Zhang, and H. Zhong, "Serving unseen deep learning models with near-optimal configurations: A fast adaptive search approach," in *Proc. 13th Symp. Cloud Comput.*, vol. 1, Nov. 2022, pp. 461–476.
[45] M. You, J. Nam, M. Seo, and S. Shin, "HELIOS: Hardware-assisted high-performance security extension for cloud networking," in *Proc. ACM Symp. Cloud Comput.*, Oct. 2023, pp. 486–501.
[46] M. V. Le, S. Ahmed, D. Williams, and H. Jamjoom, "Securing container-based clouds with syscall-aware scheduling," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, vol. 9, Jul. 2023, pp. 812–826.
[47] S. Y. Lim, B. Stelea, X. Han, and T. Pasquier, "Secure namespaced kernel audit for containers," in *Proc. ACM Symp. Cloud Comput.*, vol. 13, Nov. 2021, pp. 518–532.
[48] S. Song, S. Suneja, M. V. Le, and B. Tak, "On the value of sequence-based system call filtering for container security," in *Proc. IEEE 16th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2023, pp. 296–307.
[49] A. Van't Hof and J. Nieh, "BlackBox: A container security monitor for protecting containers on untrusted operating systems," in *Proc. 16th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2022, pp. 683–700.
[50] S. Li, W. Wang, J. Yang, G. Chen, and D. Lu, "Golgi: Performance-aware, resource-efficient function scheduling for serverless computing," in *Proc. ACM Symp. Cloud Comput.*, Oct. 2023, pp. 32–47.
[51] A. S. Jacobs, R. J. Pfitscher, R. H. Ribeiro, R. A. Ferreira, L. Z. Granville, W. Willinger, and S. G. Rao, "Hey, lumi! using natural language for intent-based network management," in *Proc. Annu. Tech. Conf. (USENIX ATC)*, 2021, pp. 625–639.
[52] Otterize. (2024). *Otterize—Automate Workload IAM*. [Online]. Available: https://otterize.com/
[53] Kafka. (2024). *Apache Kafka*. [Online]. Available: https://kafka.apache.org/
[54] (2024). *Open Policy Agent*. [Online]. Available: https://www.openpolicyagent.org/
[55] (2024). *Gatekeeper—Policy Controller for Kubernetes*. [Online]. Available: https://github.com/open-policy-agent/gatekeeper
[56] H. Kang and S. Shin, "*Verikube*: Automatic and efficient verification for container network policies," *IEICE Trans. Inf. Syst.*, vol. E105.D, no. 12, pp. 2131–2134, 2022.
[57] Y. Li, X. Hu, C. Jia, K. Wang, and J. Li, "Kano: Efficient cloud native network policy verification," *IEEE Trans. Netw. Service Manage.*, vol. 20, no. 3, pp. 3747–3764, Sep. 2022.
[58] Y. Li, C. Jia, X. Hu, and J. Li, "Kano: Efficient container network policy verification," in *Proc. IEEE Symp. High-Performance Interconnects (HOTI)*, Aug. 2020, pp. 63–70.
[59] A. Rogers, O. Kovaleva, and A. Rumshisky, "A primer in BERTology: What we know about how BERT works," *Trans. Assoc. Comput. Linguistics*, vol. 8, pp. 842–866, Dec. 2020.
[60] Wikipedia. (2024). *Wikipedia, Cosine Similarity*. [Online]. Available: https://en.wikipedia.org/wiki/Cosine_similarity
[61] gRPC. (2024). *gRPC*. [Online]. Available: https://grpc.io/
[62] (2022). *K8sowners*. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/owners-dependents/
[63] Containerd. (2024). *Containerd—An Open and Reliable Container Runtime*. [Online]. Available: https://github.com/containerd/containerd
[64] OnlineBoutique. (2024). *Online Boutique*. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo
[65] M. U. Hadi, R. Qureshi, A. Shah, M. Irfan, A. Zafar, M. B. Shaikh, N. Akhtar, J. Wu, and S. Mirjalili, "A survey on large language models: Applications, challenges, limitations, and practical usage," *TechRxiv*, Jul. 2023, doi: 10.36227/techrxiv.23589741.v1.

BOM KIM is currently pursuing the M.S. degree with the Department of Computer Science and Engineering, Incheon National University. Her research interests include the automation of intent-driven security policy generation and validation for advanced cloud-native environments.

SEUNGSOO LEE received the B.S. degree in computer science from Soongsil University and the M.S. and Ph.D. degrees in information security from KAIST, in 2020. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Incheon National University. His research interests include developing secure and robust cloud/network systems against potential threats.