IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 16 August 2023, accepted 27 August 2023, date of publication 30 August 2023, date of current version 7 September 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3310281

RESEARCH ARTICLE

Kunerva: Automated Network Policy Discovery Framework for Containers

SEUNGSOO LEE[1] AND JAEHYUN NAM[2]
[1]Department of Computer Science and Engineering, Incheon National University, Incheon 22012, Republic of Korea
[2]Department of Computer Engineering, Dankook University, Yongin 16890, Republic of Korea

Corresponding author: Jaehyun Nam (jaehyun.nam@dankook.ac.kr)

This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korea Government through MSIT (Development of Darkweb Hidden Service Identification and Real IP Trace Technology) under Grant 2022-0-00740.

ABSTRACT Containerization has gained significant popularity in cloud-native applications, offering lightweight and portable capabilities, with container orchestration platforms such as Kubernetes, simplifying deployment and management. However, the presence of human errors, especially misconfigurations, continues to pose substantial security risks to containers. One specific challenge lies in generating effective network security policies, given the intricate nature of label-based container management and the dynamic characteristics of container deployments. This paper introduces KUNERVA, an innovative and automated solution specifically designed to tackle the critical security challenge in container environments. KUNERVA focuses on policy discovery utilizing network logs to generate a minimum set of network security policies to achieve maximum network traffic coverage while ensuring the security isolation between containers. To enhance the reliability of the generated policies, KUNERVA seamlessly integrates with a policy enforcement system, Gatekeeper, for accurate policy verification. Consequently, KUNERVA ensures the discovery of an efficient and effective network policy set, blocking the enforcement of malicious network policies.

INDEX TERMS Container security, network security policy, policy discovery, policy verification.

I. INTRODUCTION

Recently, containers have increasingly been used as a virtualization technology to develop, deploy, and run applications or software infrastructures. Portworx and Aqua Security surveys revealed that 87% of organizations were using containers in 2019, compared to 67% in 2017 [1]. With this popularity, the container market was predicted to expand from 1.2 billion in 2018 to 4.9 billion (USD) by 2023 [2]. A container can be defined as a stand-alone executable unit of software that includes an entire runtime environment [3], which promotes modularity and reproducibility, two main reasons for containers being widely used [4]. The use of containers brings several benefits, including fast software delivery, application code reuse across environments, and reduction of infrastructure costs. Especially in such environments, container orchestration platforms, such as Kubernetes [5], are commonly used to facilitate the deployment and management of large-scale containerized applications.

Despite containers' numerous benefits, security configuration and policy management remain significant concerns [1], [4]. From previous research [6], they highlight the infeasibility and error-prone nature of manually configuring various security policies for containerized applications. A notable example occurred in 2019 when attackers exploited a misconfigured container in Capital One's system, gaining access to 30 GB of application data containing the sensitive personal and financial information of 106 million individuals [7]. In addition, the dynamic and complex nature of container workloads significantly complicates security configuration and policy, further exacerbated by emerging security attacks, vulnerabilities, and malware. Socchi et al. [8] predict that the average number of new vulnerabilities caused by system packages in containers is expected to increase by approximately 105 vulnerabilities per year between 2019 and 2025. Furthermore, recent studies reveal that existing Linux-based

The associate editor coordinating the review of this manuscript and approving it for publication was Somchart Fugkeaw.

security implementations, such as Linux Security Module (LSM) utilized in containers, are vulnerable to cyber attacks due to their obsolescence and inability to accommodate the latest security vulnerabilities and malware behavior [9]. Hence, it is crucial to establish and enforce accurate and effective security policies to prevent such attacks and threats.

However, the dynamic nature of containers introduces several practical challenges in generating security policies. First, it is exceedingly challenging to manually create and enforce security policies in environments where containers are constantly being created and terminated in real-time. Second, as the role of labels attached to containers is instrumental in crafting these policies, it is vital to understand the context of these labels that provide and maintain this understanding consistently throughout policy discovery and enforcement. Lastly, the verification of policy accuracy is of paramount importance. Even seemingly minor errors (e.g., specifying an incorrect service port) can lead to severe outcomes, including denial of service or information leakage [10], [11].

This paper introduces KUNERVA to address those significant challenges. KUNERVA is the first automated, container-aware framework that conducts a comprehensive discovery of network security policies, while being highly attuned to the dynamics of container operation. KUNERVA is designed to efficiently handle large volumes of network logs with minimal impact on performance. This approach facilitates the creation of highly effective network policies encompassing maximum data flow between containers. In addition, KUNERVA seamlessly integrates with Gatekeeper [12], a well-regarded open-source policy engine, to ensure the accuracy and validation of the generated network policies. Our evaluation demonstrates that KUNERVA enables the automated discovery and verification of network security policies under the complexities of containerized environments while incurring negligible performance overhead, even in scenarios involving large volumes of network logs.

In summary, our main contributions are as follows:

- Minimax discovery method: the proposed approach automatically aggregates network logs and generates the minimum set of policies, thereby enabling maximal network flow coverage among containers while maintaining rigorous security isolation.
- Effective safety measures: to ensure the accurate validation of the discovered network security policies, the proposed approach implements a measure designed to prevent the enforcement of potentially malicious policies that could lead to information leakage situations.
- Practical evaluation: we have tested the proposed approach within the context of a real-world web-based e-commerce microservices application on Kubernetes, a leading container orchestration platform. This effectively demonstrates its capability to successfully discover and validate network policies.

The rest of this paper is organized as follows: Section II provides the background and challenges in generating network policies. Section III reviews previous studies and their limitations. The overall design of KUNERVA and its workflow are presented in Section IV and Section V respectively. The implementation of KUNERVA is described in Section VI, while its evaluation results are summarized in Section VII. Section VIII discusses the limitations of the current design. Finally, we conclude this paper in Section IX.

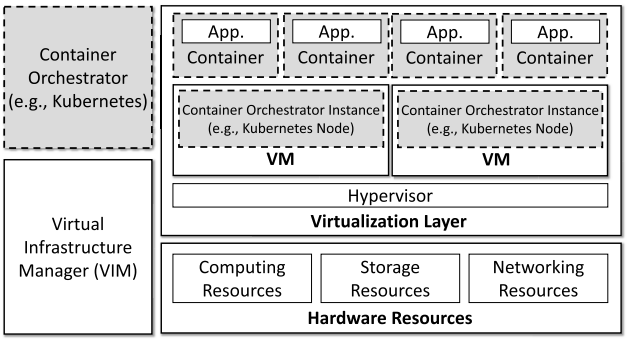

FIGURE 1. ETSI Container Environment Architecture [19]. Gray-colored components are related to containers.

II. BACKGROUND AND CHALLENGES

This section provides a background on containerization and describes the key challenges to realize our framework.

A. BACKGROUND

1) CONTAINERIZATION

Cloud environments offer numerous benefits for application deployment, including scalability, reliability, and observability. Traditionally, frameworks such as OpenStack [13] have allowed organizations to build their cloud infrastructures using virtual machines (VMs) [14]. However, containerization has surfaced as an appealing alternative due to limited portability and significant overhead imposed by VMs. The adoption of containers provides a myriad of benefits for practitioners in software development and deployment, including agility, portability, reproducibility, modularity, and flexibility [9], [15], [16], [17]. With these benefits, microservices architecture has incorporated containers to boost the operational efficacy of large-scale systems [18].

As depicted in Figure 1, a container is a standardized software unit that packages source code, dependencies, and all other executable executables together [19]. This comprehensive encapsulation enables software developers to run containerized software across diverse computing platforms. There exist various solutions to facilitate the creation, operation, and maintenance of containers (e.g., Docker [3] and Podman [20]). They are often referred to as container engines [21] or container runtimes [17], underlining their pivotal role in the realm of containerized applications.

While containers and virtual machines (VMs) share similarities in isolation and resource allocation, they fundamentally differ in their abstraction levels [15], [17]. Containers operate at the operating system (OS) level, encapsulating elements such as binaries, libraries, and executables, while

VMs provide hardware-level abstraction (e.g., CPU, Memory, and storage). This OS-level abstraction allows containers to be more resource-efficient and have quicker startup times on a host machine compared to VMs [17]. A hypervisor manages the lifecycle of VMs, whereas a container orchestration platform, such as Kubernetes [5], indirectly oversees containers throughout their entire life cycle, including scheduling, deployment, patching, and deletion. Depending on an orchestration platform, containers can be managed and organized collectively (e.g., *pod*[1] in the case of Kubernetes).

2) NETWORK POLICY ENFORCEMENT IN CLOUD

Network policy enforcement is vital in securing containerized applications within a cloud environment [22]. While a single-node-based container environment (e.g., Docker) implements network policies within bridge networks using iptables to regulate network flow, a multi-node-based container environment (e.g., Kubernetes) necessitates the deployment of various network plugins, known as container network interfaces (CNIs). These CNIs, such as Calico [23], Cilium [24], and Weave Net [25], facilitate network policy enforcement within the container cluster. Once one of the CNIs is in place, administrators can regulate network traffic within the cluster by manipulating *NetworkPolicy* resources - creating, updating, or deleting them as needed. This task involves defining and implementing rules dictating how pods interact with one another and external networks. By enforcing these network policies, administrators can effectively control network traffic, isolate sensitive workloads, and shrink the attack surface, significantly reducing potential security threats [26].

In Kubernetes, for instance, network policies are specified in a declarative way using YAML or JSON configuration files [27]. These files comprise three primary parts: *a selector, ingress rules, and egress rules*. The selector is used to determine the pods to which the network policy applies. On the other hand, ingress rules establish which incoming connections are permissible, whereas egress rules define the allowed outgoing connections. Each rule can be formulated based on various criteria (e.g., pod labels, namespace labels, and IP address ranges). Upon the application of a network policy, the network plugin translates the defined rules into low-level network constructs, such as iptables rules [28], eBPF programs [29], or OpenFlow entries [30]. This conversion ensures the enforcement of the intended network behavior at the infrastructure level.

Let us consider a scenario, as depicted in Figure 2, to understand how network policies operate in a container-based cloud environment. Here, four pods (Pod A, B, C, and D) are deployed within a two-node Kubernetes cluster, each pod possessing unique label information. An egress policy applied to Pod B initially allows network flows directed towards pods labeled `app:david` (i.e., Pod D), targeting port 6379. Conversely, Pod D permits network traffic only

[1]In Kubernetes, the term 'pod' is commonly used to denote a container or a group of containers. This paper uses both terms interchangeably.

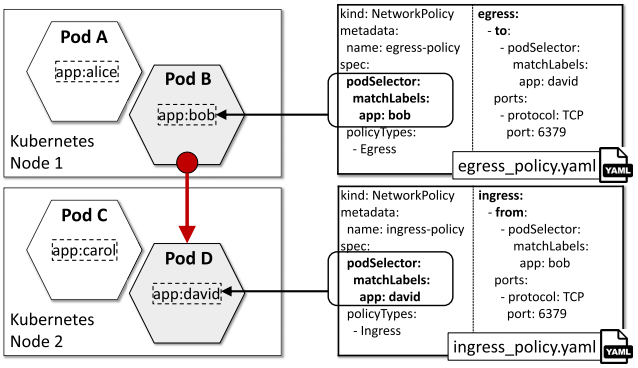

FIGURE 2. Example of network policy enforcement in cluster. Dashed-line box refers to the label assigned to each pod.

from pods bearing the label `app:bob` (i.e., Pod B), and the destination port is 6379, which aligns with its service port. All other connections are effectively denied.

B. CHALLENGES IN NETWORK POLICY GENERATION

Generally, in the absence of any deployed network policy, all pods within a cluster are free to communicate with each other. Hence, as the previous section underscores, network policies must be accurately established in the cluster to ensure its security and isolation aspects. Nevertheless, manually generating appropriate network policies can be extremely time-consuming and laborious. Here, we delineate three key challenges that surface during network policy generation.

1) C1: AUTOMATION

One significant challenge in network policy discovery is automation, which involves managing a multitude of containers efficiently. This task can render policy management complicated and prone to errors. As the number of labels, namespaces, and pods in a cloud environment proliferates, defining and maintaining adaptive policies for changes through manual efforts becomes increasingly challenging. Furthermore, an automated system should possess the capability to intelligently create, update, and delete policies in response to shifts in the cloud environment.

Moreover, implementing an automation process in network policy generation necessitates the development of algorithms and methodologies. They should be capable of interpreting the relationships between containers and network connections, and then formulate suitable network policies in line with application requirements and security constraints. These processes should also consider the integration with various network plugins, such as Cilium [24]. This necessitates seamless compatibility between the orchestration system, the network policy API, and the chosen network plugin.

2) C2: CONTAINER-AWARENESS

The second challenge in generating network policies for container systems lies in *container-awareness*, the capacity to recognize and handle network policies within the context of dynamic, container-based environments. Containers

and their corresponding workloads can be instantaneously created, scaled, or terminated on demand, leading to a constantly changing network topology. This dynamic nature poses unique challenges for designing and enforcing network policies.

Furthermore, understanding the label context of containerized applications is another challenge in defining network policies from a container-awareness perspective. For instance, most container orchestration systems (e.g., Kubernetes) group containers into pods and allocate labels to these pods for identification, and network policies highly rely on these labels to establish communication rules. However, these labels can be dynamically updated based on the administrator's needs. As a result, network policies should be flexible enough to allow essential communication between containers while ensuring security and isolation.

3) C3: VERIFICATION

The third challenge in orchestrating network policies for container-based orchestration systems is guaranteeing that the established policies accurately mirror the intended communication norms and restrictions while preserving security and isolation parameters. Consequently, a verification process is essential for identifying and rectifying any errors or misconfigurations in the network policies, which could result in unintended negative consequences such as unauthorized access or data leakage.

An integral part of the verification process involves validating the correctness of the network policy rules, particularly within ingress or egress directives. This process necessitates confirming that these rules promote requisite communication paths between pods while eliminating unauthorized connections. Given the significance of labels and selectors in establishing communication parameters within Kubernetes network policies, it becomes imperative to accurately configure these elements to align with the desired network behavior. Therefore, an efficient verification process should incorporate real-time network behavior monitoring mechanisms to evaluate the implications of any policy alterations on the overall security and performance of the cluster.

III. RELATED WORK

A. CONTAINER SECURITY

A substantial amount of research highlights that Docker images contain a significant number of high-risk vulnerabilities, ranging from 30% to 90% [31], [32], [33], underscoring a severe concern with these images. Although various vulnerability assessment tools are available for Docker images, the question raised is whether such tools are used in production environments and if their use could impede the deployment process. Thus, some studies [22], [34], [35], [36] have been conducted on container security, primarily focusing on verifying the security of container images [34], ensuring their integrity [35], [36]. However, these studies limit their focus to a single security aspect, such as creating vulnerability-free container images or integrity attestation. They do not provide solutions for verifying and enforcing security policies during runtime. For instance, Ahamed et al. present a vulnerability-centric approach for identifying and assessing vulnerabilities in Docker container images [34]. Similarly, other works offer solutions for container integrity attestation throughout the entire lifecycle of containers and their underlying images [35], [36].

On the other hand, the majority of container network security solutions [37], [38] have primarily concentrated on container network performance while largely overlooking fine-grained policy enforcement. A small number of recent studies have explored security issues in container networks. Bui [39], Comb et al. [40], and Chelladhurai et al. [41] conducted analyses of Docker container network security. Nam et al. [22] proposed a robust, secure network stack that enforces the principle of least privilege for container network access by restricting connectivity solely to the inter-dependencies between the container itself and any required containers necessary for composing a service.

B. KUBERNETES SECURITY

Kubernetes [5], an open-source container orchestration platform, revolutionizes the automation of deploying, scaling, and managing containerized applications. Initially developed by Google [42], its maintenance is now under the purview of the Cloud Native Computing Foundation (CNCF) [43]. Offering a robust infrastructure for managing containerized workloads and services, Kubernetes emphasizes both declarative configuration and automation. Its capabilities extend to efficient resource utilization and streamlining the management of large-scale, distributed applications across diverse computing environments.

The majority of existing solutions [44], [45], [46] propose reactive measures, which detect security policy violations after they occur, potentially creating larger attack windows and elevating security risks. Sysdig [45] offers a security attack detection approach at the system-call level, while Falco [44] presents an online anomaly detection tool for containerized applications. KubAnomaly [46] is a learning-based anomaly detection system that provides real-time monitoring capabilities in Kubernetes. Conversely, KubeArmor [47] is a runtime security enforcement system that protects containers from potential threats and vulnerabilities by monitoring and enforcing security policies at the container level using extended Berkeley Packet Filter (eBPF) and Linux Security Module (LSM). Moreover, OPA(Open Policy Agent) is a security policy engine [48], and Gatekeeper [12], as its sidecar, serves as an enforcement tool designed explicitly for Kubernetes.

C. SECURITY POLICY VERIFICATION

Several proactive security compliance verification approaches [13], [49], [50], [51] have been proposed for non-container

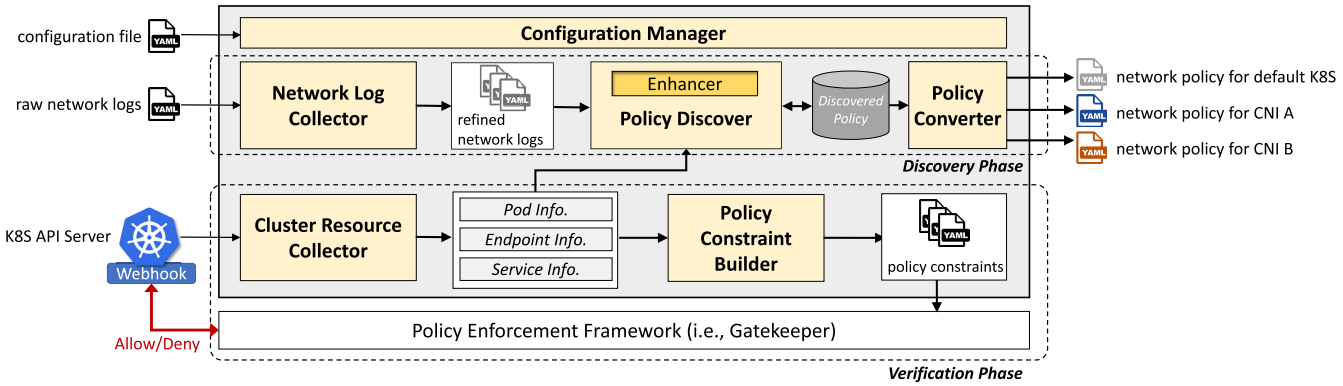

FIGURE 3. Overall architecture of KUNERVA with six key components; (*i*) Configuration Manager, (*ii*) Network Log Collector, (*iii*) Policy Discover, (*iv*) Policy Converter, (*v*) Cluster Resource Collector, and (*vi*) Policy Constraint Builder.

environments such as OpenStack [13]. For instance, Weatherman [49] utilizes graph-based and Datalog-based models to verify security policies in cloud environments. A proactive protection approach [14] for potential security breaches in the cloud is proposed in another work. LeaPS [51] and Proactivizer [52] are also proactive security auditing solutions for cloud environments. However, these approaches are not designed to address the complexities and challenges unique to container environments, such as capturing container-specific events, handling dependencies among diverse types of resources, and deriving predictive models from those dependencies. ProSPEC [53] leverages learning-based prediction to perform computationally intensive tasks, such as security verification, in advance, improving process efficiency while maintaining a lightweight runtime for tasks such as policy enforcement.

In contrast, KUNERVA primarily focuses on discovering network policies and the corresponding verification process. First, it automates policy discovery based on network log information. Second, it automatically updates the previously discovered network policies according to the most recent log information, keeping container awareness at the forefront. Lastly, it verifies the discovered network policies using a popular policy enforcement framework, Gatekeeper [12]. This unique approach distinguishes KUNERVA from other solutions by emphasizing the dynamic nature of policy discovery, adaptation, and verification in containerized environments.

IV. SYSTEM DESIGN

This section clarifies the design considerations motivating KUNERVA. Then, it provides a comprehensive description of its system architecture designed to create network policies specifically for containerized environments while ensuring the accuracy of the network policies through verification.

A. DESIGN CONSIDERATIONS

We first derive the following design considerations that a system must follow to generate network policies tailored for containerized environments.

1) AUTOMATIC DISCOVERY

A system should minimize human intervention and reduce the time in generating network policies to ensure network security and isolation. Since manual policy generation can be error-prone and laborious, with the potential for misconfigurations, a system should be able to automatically compute network policy candidates by leveraging network logs with container-related data (e.g., labels).

2) EFFICIENT EVOLUTION

A system should efficiently and effectively compute network policies with container awareness. Since generating network policies statically applied to individual pods could lead to an overwhelming number of policies, especially in environments with a large number of pods, a system should produce a minimum set of network policies while covering the maximum number of network flows within containerized environments.

3) PROACTIVE VERIFICATION

A system should verify the suitability of the discovered network policies with the intention of cloud-native applications. In the same vein, a system should be able to integrate with a policy enforcement framework (e.g., Gatekeeper [12]) to proactively assess the discovered network policies before applying them to pods, ensuring that only appropriate and suitable network policies are implemented within the intended operation.

B. SYSTEM ARCHITECTURE

This section presents the overall architecture of KUNERVA. As depicted in Figure 3, KUNERVA comprises six primary components: configuration manager, network log collector, policy discoverer, policy converter, cluster resource collector, and policy constraint builder. Based on the design considerations, it is designed to support two distinct phases, represented by the dashed line box in Figure 3. The discovery phase focuses on generating network policies within the cluster by analyzing raw network logs. The verification phase allows for evaluating the suitability of the discovered network policies for individual pods before application.

1) CONFIGURATION MANAGER

The configuration manager plays a role in managing and organizing various settings and configurations associated with KUNERVA. It ensures that the appropriate parameters are provided for the discovery and verification phases and the other components within this framework. An essential responsibility of the configuration manager is instructing the network log converter to connect to a designated log monitor, such as Cilium Hubble [54]. Furthermore, it can direct the policy converter to convert the discovered policy into a specific CNI's network policy in addition to the default Kubernetes network policy.

2) NETWORK LOG COLLECTOR

The network log collector is responsible for processing and transforming raw network logs generated by the log monitor into a structured format that can be easily analyzed and utilized by the policy discover. These network logs contain key information about communication patterns and network flows between different pods within the cluster. However, since each log monitor may include unnecessary details, the network log collector eliminates such data and converts the logs into a structured format that includes only the essential information required for the discovery process.

3) POLICY DISCOVER

The policy discover is a crucial component of KUNERVA, tasked with analyzing the structured network logs and autonomously identifying the communication patterns between pods within the cluster. Through this analysis, this component generates network policy candidates that capture the observed communication behaviors while adhering to the cluster's security and isolation requirements. In addition, the *enhancer* of this component plays a vital role in computing the minimum set of policies necessary to cover the maximum network flows and efficiently updating policies from older to newer versions.

4) POLICY CONVERTER

The policy converter is designed to ensure compatibility and adaptability with various CNIs (e.g., Cilium and Calico) and the default network policy in Kubernetes. Its primary function is to take the network policies discovered by the policy discover and convert them into the specific network policy format required by the target CNI in the containerized cloud environment. This capability enables KUNERVA to be highly versatile, as it can seamlessly integrate with different network plugins, expanding its applicability and usefulness in a wide range of scenarios.

5) CLUSTER RESOURCE COLLECTOR

The cluster resource collector is responsible for extracting and processing relevant Kubernetes resources needed to generate network policies. It interacts with the API server of a container orchestration system, specifically Kubernetes, to gather information about resources such as pods, services, and endpoints. This collected information is then processed into a structured format that the discovery and verification phases in KUNERVA can easily utilize. Through effective management and conversion of resource data, the resource collector ensures that the generated policies accurately align with the current state of the containerized environment.

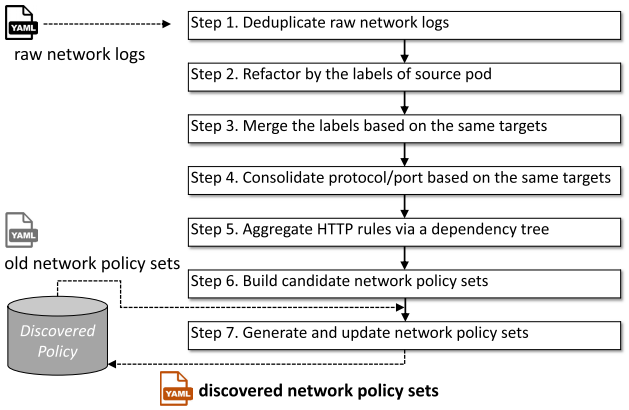

FIGURE 4. Workflow for discovering a network policy.

6) POLICY CONSTRAINT BUILDER

The policy constraint builder is essential for verifying network policies with a policy enforcement framework such as Gatekeeper [12]. Utilizing the Kubernetes resource data, this component generates constraint files in `ConstraintTemplates`, which the enforcement framework requires to evaluate the discovered network policies. By incorporating these constraint files, the enforcement framework can validate the discovered policies against the desired security and isolation requirements for the containerized environment. As a result, this proactive verification of network policies before their application to the cluster reduces the potential for misconfigurations.

C. SYSTEM WORKFLOW

Here, we describe the overall workflow of KUNERVA. As shown in Figure 3, KUNERVA operates in two distinct phases: policy discovery and verification.

1) POLICY DISCOVERY PHASE

The collaboration of the network log collector, cluster resource collector, policy discover, and policy converter components involves a series of steps. First, the network log collector obtains raw network logs from log monitors or databases and filters out unnecessary information.

Then, the policy discover utilizes the refined logs and cluster resource data to generate network policy candidates that capture the observed communication behaviors following the steps as shown in Figure 4. Especially, the policy discovery also utilizes the enhancer module to evolve the policies, and eventually builds network policies in the form of Kunerva network policy (Appendix). The key features of

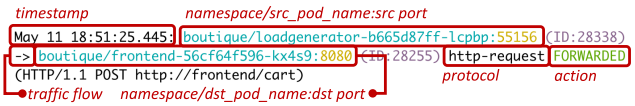

FIGURE 5. An example of a network log generated by the log monitor (i.e., Cilium Hubble).

the enhancer module as they relate to discovery are described in the following sections.

Finally, the policy converter component takes the discovered network policies and converts them into the specific network policy format required by the target CNI. This collaborative process ensures the effective discovery, analysis, and conversion of network policies.

2) POLICY VERIFICATION PHASE

The collaboration of the cluster resource collector and policy constraint builder involves several steps. First, the cluster resource collector retrieves the necessary resource data from the cluster's API server and refines it into the essential information required for the verification process (1). Subsequently, the policy constraint builder utilizes these refined data to construct policy constraints that define the allowed access of pods and endpoints to specific network resources within the cluster (2). Finally, the constraint builder applies these constraints to the policy enforcement framework, enabling proactive verification of the generated network policies to ensure compliance with the intended security and isolation requirements (3). This collaborative process ensures that network policies are thoroughly verified before their application, mitigating the risks of misconfigurations and bolstering overall security.

V. CONTAINER-AWARE POLICY DISCOVERY

This section outlines the container-specialized functionalities of KUNERVA for both the policy discovery and verification phases. It highlights the specific features and capabilities of our framework that are tailored to the containerized environment, enabling effective network policy discovery and proactive verification in such contexts.

A. EFFICIENT CONSOLIDATION

In essence, a network log in KUNERVA represents a single instance of network traffic, and its structure may vary slightly depending on the log monitor used. As illustrated in Figure 5, each network log contains information such as the source and destination pod names, protocol, port details, and flow type (e.g., egress or ingress). However, generating a separate network policy for each network log could lead to an excessive number of policies being created within a short period.

KUNERVA incorporates rule consolidation as a solution to tackle the challenge of a potentially overwhelming number of network policies. The process begins by generating candidate network policies. These policies are then analyzed to identify common attributes, such as matching pod selectors and rule types (e.g., egress or ingress). If the candidate policies with

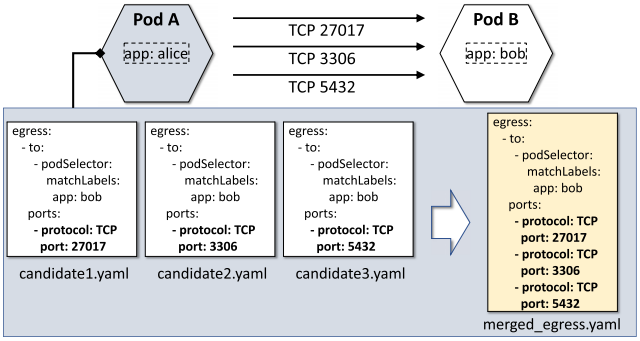

FIGURE 6. An example of merging multiple policy candidates into a single network policy.

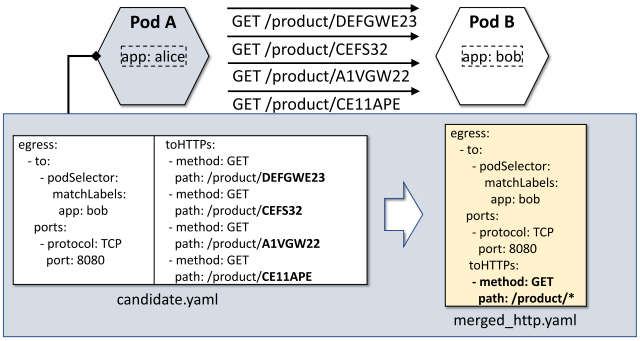

FIGURE 7. An example of merging multiple `toHTTPs` rule candidates into a single policy rule.

matching attributes are found, the policy discover merges them into a single policy while preserving the original security and isolation properties within the cluster. This consolidation approach ensures a more concise and manageable set of network policies, reducing complexity.

Consider the scenario depicted in Figure 6, where we have two pods, Pod A and Pod B. Pod A initiates three distinct network traffic flows to Pod B, each with a different destination port: 27017, 3306, and 5432. The policy discover generates three initial policy candidates in KUNERVA and analyzes their pod selectors and egress policy types. While the label selectors of the candidates are not explicitly shown in the illustration, they point to Pod A. As a result of analyzing the candidates, it is determined that they share identical pod selectors (Pod B) and egress types. Consequently, the enhancer module within the policy discover merges these candidates into a single policy, represented as ''merged_egress.yaml'' in Figure 6. This consolidation simplifies the network policies, reducing redundancy and enhancing overall manageability.

While merging multiple candidate policies into a single policy can improve manageability to some extent, it is not a universal solution. The resulting single policy can still be quite large, depending on the number of policy rules involved. This is especially true when a CNI supports L7 policy rules (e.g., `toHTTPs`). In such cases, a candidate policy can significantly expand in size as more distinct types of HTTP traffic are discovered. For example, if Pod A sends

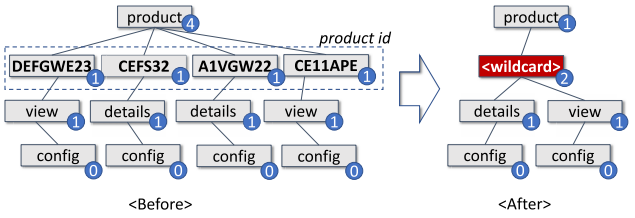

FIGURE 8. An example of an L7 dependency tree. The number within each circle refers to the count of child nodes.

four distinct types of HTTP data to Pod B, the candidate policy would contain four toHTTPs rules, as depicted in Figure 7. Consequently, if there are 100 distinct toHTTPs URLs, the resulting policy could contain 100 individual toHTTPs rules. This highlights the potential for larger policies when dealing with a substantial number of distinct rule variations.

Algorithm 1 Building a L7 Dependency Tree

```
1  BuildDependencyTree (Root, Paths, T)
       Inputs : Root: L7 dependency tree (initially,
                empty), Paths: URI, an array of paths
                split with '/', T: Pre-defined threshold
                for aggregation
2      Root = buildL7Tree(Root, Paths);
       /* traverse the built tree by
          merging the children nodes   */
3      foreach node in Root do
4          if Number of node.Child > T then
5              node.Child = MergeChild(node);
6      return Root;
7  MergeChild (Node)
       Inputs : Node: Current node in the dependency
                tree
8      newChild = NewChildNode();
9      foreach child in Node.Child do
10         if Number of child.Child ≠ 0 then
11             foreach c in child.Child do
12                 if c in newChild.Child then
13                     newChild.Child[c] =
                        append(c.Child);
14                 else
15                     newChild.Child = append(c);
16     newChild.Path = wildcard;
17     newChild.Depth = node.Depth+1;
18     return newChild;
```

To address the challenge of numerous L7 rules, the enhancer module in the policy discover utilizes an L7 dependency tree approach based on multiple toHTTPs rule candidates, as outlined in Algorithm 1. This module constructs the

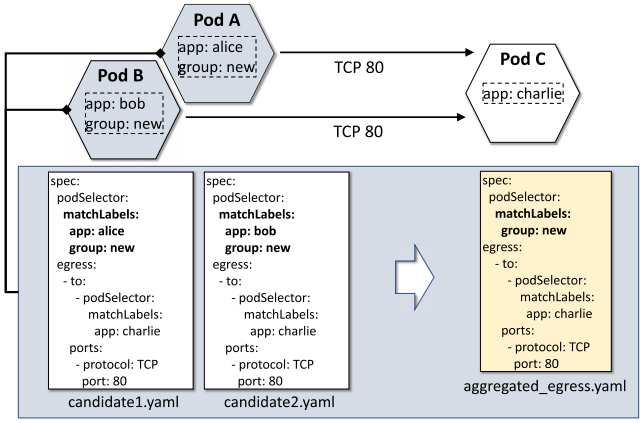

FIGURE 9. An example of aggregating the intersecting labels for multiple pods and merging the candidates into a policy.

L7 dependency tree and traverses its nodes, keeping track of the number of child nodes and their depth simultaneously. When the number of child nodes exceeds a user-defined threshold, the enhancer module merges them into a wildcard. For example, in Figure 8, the enhancer module identifies that the product node has four child nodes at the same depth (Before in Figure 8). To simplify the policy, the enhancer module combines these child nodes into a wildcard (in this case, the threshold value is 3) using regular expressions (After in Figure 8). This approach effectively reduces the complexity and size of the resulting policy by consolidating similar L7 rules.

B. EFFECTIVE INTERSECTION MERGING

Labels play a critical role in efficiently managing and organizing resources, such as pods and services, in containerized cloud environments. In the context of network policy enforcement, labels are especially valuable as they allow for the specification of network policies based on selectors that match the labels associated with pods. This ensures that policies are applied only to the intended pods. By effectively leveraging label information, it becomes possible to generate a minimal set of network policies that can cover the maximum number of network flows, optimizing the policy set and improving overall efficiency.

Consider the scenario depicted in Figure 9, where three pods are deployed, and two of them share the common label 'group:new'. Both of these pods generate identical TCP traffic directed to Pod C on destination port 80. In a naive approach, two separate network policies could be applied to manage this traffic (candidate1 and candidate2 policies in Figure 9). However, for more efficient policy management, our framework consolidates these policies into a single aggregated policy (aggregated policy in Figure 9). This aggregated policy specifies the label 'group:new', thus affecting both Pod A and Pod B simultaneously. This aggregation approach allows for more streamlined and concise policy management, reducing complexity and enhancing overall efficiency.

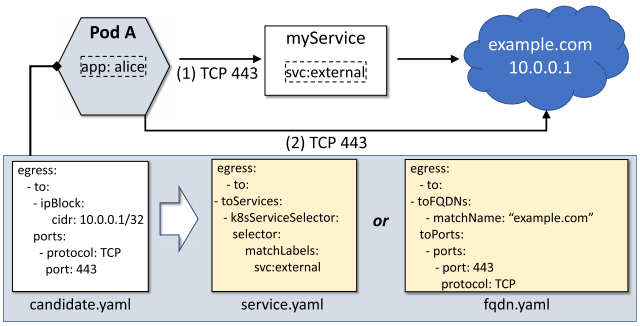

FIGURE 10. An example of replacing a CIDR policy with an external service or FQDN policy as a more flexible form.

C. ADVANCED EVOLUTION

In addition to basic evolution for policy discovery (i.e., merging or aggregating network policies), KUNERVA offers advanced evolution techniques to further improve policy management effectiveness. One such technique involves transforming raw network policies into high-level policies that enhance security within the cluster. This advanced evolution aims to streamline and optimize policies, enabling more robust and effective security measures to be implemented.

The process of evolving raw policies into high-level policies is illustrated in Figure 10. The scenario involves two distinct ways in which Pod A can send traffic to example.com, an external service not residing within the cluster. The first approach entails Pod A directing the traffic to a pre-defined service resource within the cluster, which points to the IP address 10.0.0.1 (1). Alternatively, Pod A can directly send the traffic to example.com (2). However, log monitors only provide traffic information indicating that Pod A sent the traffic to 10.0.0.1, which is challenging to determine whether this traffic originates from the service or a direct connection.

To address the challenge of identifying the origin of traffic directed towards external services, KUNERVA incorporates an internal database that maintains a record of external service resources and domain information related to pod connections. During the network policy discovery process, when a CIDR (Classless Inter-Domain Routing) policy is identified as a candidate, our framework checks if it matches any service resource or domain information in the database. If a match is found, it updates the policy to a service or FQDN (Fully Qualified Domain Name) policy accordingly. This approach enables the generation of more accurate and high-level network policies that align with the actual communication patterns and requirements within the containerized environment, improving overall precision and security.

In summary, the processes outlined in Sections V-A, V-B, and V-C contribute to reducing the complexity and redundancy of network policy rules, resulting in a more efficient and effective representation of the desired network security and isolation requirements in the cluster. They enable network administrators and researchers to work with a more manageable and comprehensible set of network policy rules, enhancing the overall manageability and understandability of the network policies in the containerized environment.

Algorithm 2 Verifying Discovered Policy

```
1  VerifyDiscoveredPolicy (Policy, Pods)
       Inputs : Policy: Discovered policy to be enforced,
                Pods: An array of Pod running in the
                cluster
2      foreach pod in Pods do
3          if pod.Labels in Policy.podSelector then
4              foreach port in pod.ServicePorts do
5                  if port.Protocol is TCP then
6                      if port.port is Policy.L4.tcpPort
                       then
7                          return Allow;
8                  if port.Protocol is UDP then
9                      if port.port is Policy.L4.udpPort
                       then
10                         return Allow;
11                 if port.Protocol is ICMP then
12                     if port.port is Policy.L4.icmpType
                       then
13                         return Allow;
14     return Deny;
```

D. EFFECTIVE SAFETY MEASURES

One significant concern in a containerized cluster is the possibility for pods to send traffic to different ports of a server pod, despite the server pod specifying a specific port number for communication. This situation can expose the server pod to unauthorized access or malicious traffic, leading to the generation of unintended policies during the discovery process based on network logs. To mitigate this issue, the verification of network policies becomes crucial as network policies enforce strict communication rules between pods, ensuring that only the intended ports are accessible and preventing unauthorized access or unintended traffic to the server pod. By incorporating thorough policy verification, as outlined in Algorithm 2, our framework first addresses this problem and enhances the overall security of the cluster.

Then, to ensure minimum safety measures, KUNERVA seamlessly integrates with a policy enforcement system, as depicted in Figure 11. A fundamental assumption is made that all pods specify the open ports in their service resources within the cluster. For instance, when Pod A and its corresponding service resource are created, KUNERVA automatically generates a constraint file specifically for Pod A. This constraint file contains details about the permitted ports and protocols (e.g., 27017 and 3306 in this example), and it is

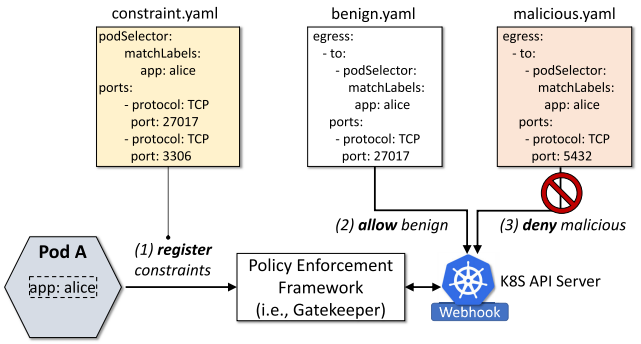

FIGURE 11. An example of verifying the discovered policy (benign) and malicious one.

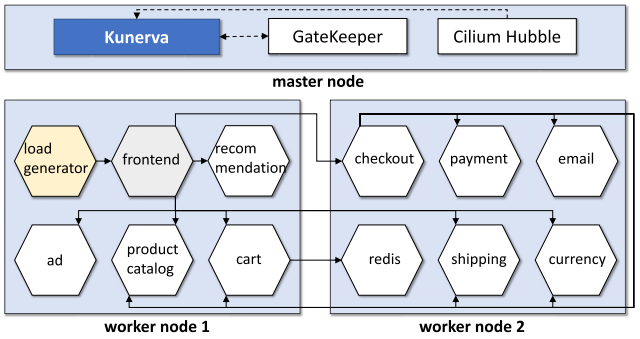

FIGURE 12. Test environments and deployed microservices. The solid lines represent traffic between the pods, while dashed ones indicate traffic that our framework interacts with.

then registered with the policy enforcement system. This integration enables the enforcement system to effectively verify network policies against the allowed ports and protocols specified in the service resources, thereby enhancing adherence to the cluster's intended communication rules.

Subsequently, whenever a network policy (whether discovered or manually created) is applied to the cluster through the Kubernetes API server, the server performs a validation check on the policy for correctness. As depicted in Figure 11, the benign policy can be applied since it aligns with the ports and protocols served by Pod A. On the other hand, the malicious policy is denied because it attempts to access port 5432, which is not served by Pod A according to the specified service resource. This verification process ensures that network policies adhere to the intended security and isolation requirements, minimizing the potential for misconfigurations and unauthorized access within the containerized cluster.

VI. IMPLEMENTATION

To prove the feasibility and effectiveness of KUNERVA, we have implemented a prototype using Go language and shell scripts comprising six key components totaling approximately 7.5K lines of code. The network log collector is designed to interface with the Hubble log monitor [54] in real-time, refining logs to optimize policy discovery. The cluster resource collector gathers essential pod, endpoint, and service data from the Kubernetes API server, which the policy discover and constraint builder utilize. In addition, KUNERVA provides support for MySQL [55] and MongoDB [56] databases for storing discovered policies.

KUNERVA supports three major network policy types: Kubernetes, Cilium, and Calico, enabling the generation of network policies specifically designed for the enforcement mechanisms provided by these CNIs within the cluster. KUNERVA offers flexibility in its operation mode, allowing for real-time or offline network policy discovery. Network policies can be discovered either from pre-collected logs or by connecting to the log monitor and performing the discovery process at predetermined intervals.

VII. EVALUATION

In this section, we conduct a real-world evaluation of KUNERVA to showcase its effectiveness and performance in practical environments.

A. TEST ENVIRONMENT

We set up a Kubernetes cluster with the Cilium overlay network using three virtual machines (VMs). One of the VMs functioned as the Kubernetes master node, while the remaining two VMs served as worker nodes. The VMs were hosted on a physical machine equipped with an Intel Xeon E5-2630v4 CPU and 64 GB of RAM. This hardware configuration provided the necessary resources for running the cluster and conducting performance evaluations.

To test the effectiveness of our system, we utilized the Online Boutique [57] application, which is a cloud-native, microservices-based demo application for e-commerce. The application consists of an 11-tier microservices architecture that enables users to browse items, add them to their cart, and make purchases. Additionally, we incorporated a load generator that continuously sends requests, simulating realistic user shopping flows to the frontend. This resulted in a total of 12 pods for the demo application. As depicted in Figure 12, our framework operated on the Kubernetes master node alongside Gatekeeper and Cilium Hubble, while the demo applications were deployed on separate worker nodes. This setup allowed us to assess the performance and effectiveness of our framework in a realistic environment.

Connectivity Check: As indicated in Table 1, the Boutique microservices applications have their own distinct labels and service ports assigned by default. To enable communication between each pod, a total of 16 egress network policies are required, considering the existence of 16 distinct egress traffic flows between the pods, as illustrated in Figure 12. Each egress network policy corresponds to a specific pod and its associated communication requirements, ensuring the necessary connectivity and access within the microservices architecture of the Boutique application.

To validate the correctness and effectiveness of the discovered network policies, we follow a two-step approach.

TABLE 1. 11 microservices and load generator under test.

Service Name	*Service Port*	*Label*
frontend	8080	app: frontend
cartservice	7070	app: cartservice
redis-cart	6379	app: redis-cart
productcatalogservice	3550	app: productcatalogservice
currencyservice	7000	app: currencyservice
paymentservice	50051	app: paymentservice
shippingservice	50051	app: shippingservice
emailservice	8080	app: emailservice
checkoutservice	5050	app: checkoutservice
recommendationservice	8080	app: recommendationservice
adservice	9555	app: adservice
loadgenerator	-	app: loadgenerator

Firstly, we capture the traffic generated by the load generator, which continuously sends requests to the front end. This traffic is then replayed to verify if the discovered network policies are functioning correctly. We monitor the log monitor and ensure no traffic is being dropped, indicating that the network policies are appropriately configured. Secondly, we continue operating our framework until it successfully discovers the 16 default network policies required for the Boutique application. Throughout this process, we collect and utilize approximately 5,000 network logs, which serve as valuable data for policy discovery and verification.

B. EFFECTIVENESS

1) USE CASE 1: LABEL INTEGRATION

As depicted in Figure 12, the frontend and checkout pods establish communication with four other pods: productcatalog, cart, shipping, and currency. Moreover, these pods send packets to the same service ports associated with these four pods. This observation indicates that the network policies for these pods can be consolidated into a single common network policy by introducing an intersecting label shared by the frontend and checkout pods. By leveraging this consolidation, we can streamline the network policy management process and reduce redundancy, resulting in a more concise and efficient set of network policies within the containerized environment.

Before the integration of labels, as illustrated in Figure 13, KUNERVA discovered a total of 16 egress and 15 ingress policies for the Boutique application. The number of egress policies was one more than the ingress policies due to the load generator not providing any service port, thereby not requiring an ingress policy. To consolidate the frontend and checkout pods, we introduced a common label (label=integrated) to both pods (A-1 and A-2 in Figure 14). After performing the discovery process again, our framework generated a total of 12 egress and 13 ingress policies, including the

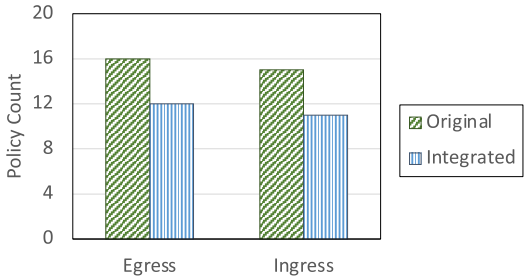

FIGURE 13. Measurements of the changes in the number of policies resulting from label integration.

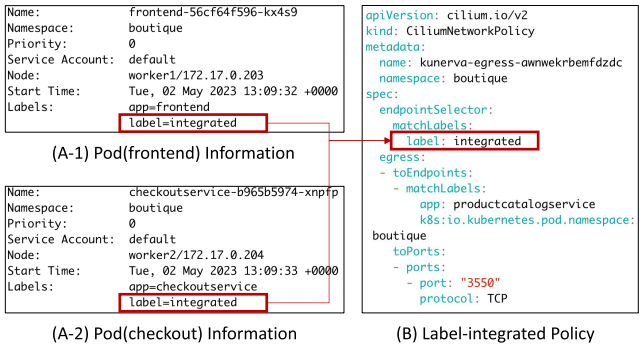

FIGURE 14. The regenerated network policy specification resulting from label integration.

newly introduced label-integrated policy (B in Figure 14). This consolidation resulted in an approximate reduction of 25% in the number of policies required. Following the application of these policies, we verified that there were no dropped packets, confirming the successful implementation and effectiveness of the network policies discovered by our framework.

2) USE CASE 2: HTTP RULE AGGREGATION

To facilitate effective L7 network policy discovery, it is essential for the log monitor to include L7 information, such as HTTP and DNS data, within the network logs. In our evaluation, we utilized Hubble as our log monitor, as it provides this required L7 information by adding annotation marks to the relevant pods. To capture HTTP log information sent to the frontend pod, we specifically annotated the loadgenerator pod, ensuring that the necessary L7 data was included in the network logs collected by KUNERVA. This enabled us to accurately analyze and discover L7 network policies based on the HTTP traffic flows within the containerized environment.

Based on the collected HTTP logs, KUNERVA initially generates an egress policy that includes five separate 'GET' method HTTP rules for the loadgenerator (Before in Figure 15). However, when we activate the enhancer module within the discovery component, the 'GET' rules are successfully aggregated into a single rule, which is then represented as a regular expression (After in Figure 15). This aggregation

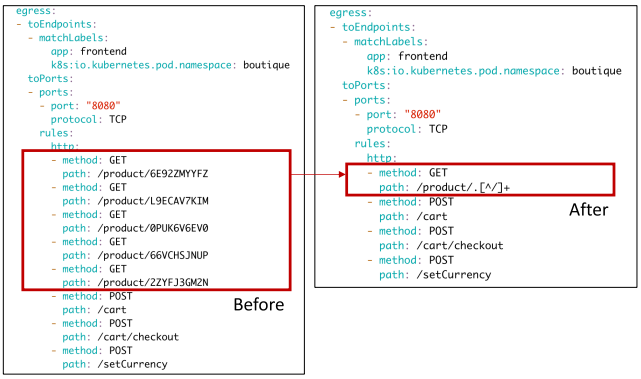

FIGURE 15. Results of toHTTP rule aggregation.

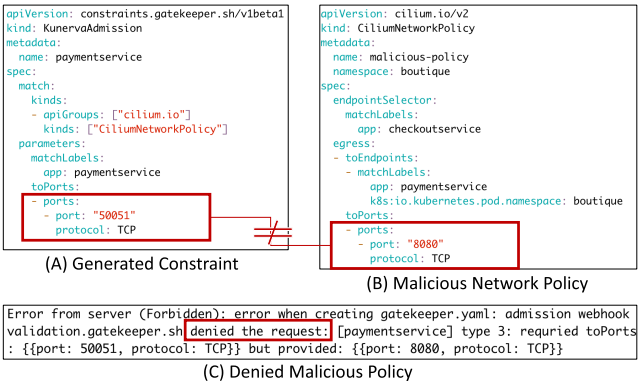

FIGURE 16. Results of label integration.

process allows us to effectively manage the L7 rule within the network policies, reducing complexity and enhancing the manageability of the policies. By consolidating similar L7 rules, our framework optimizes the resulting network policies, leading to improved efficiency and more streamlined policy management within the containerized environment.

3) USE CASE 3: POLICY VERIFICATION

To ensure the correctness of the discovered network policies, KUNERVA integrates with the policy enforcement system, Gatekeeper [12]. To facilitate the accurate verification of the policies generated by our framework, we provide a constraint file that outlines the specific standards and criteria for validating the network policy. This constraint file serves as a guide for the enforcement system to effectively assess the compliance of the network policy. As part of KUNERVA's operation, whenever a new pod is created within the cluster, it automatically generates the corresponding constraint file and registers it with the policy enforcement system. This integration enables the proactive verification of network policies against the intended security and isolation requirements, ensuring adherence to established standards within the containerized environment.

In our test environment, consisting of 11 pods excluding the loadgenerator, KUNERVA generated a total of 11 constraint files. Each constraint file corresponds to a specific pod within the cluster and provides information about the services it offers, including the port numbers and protocols used. For instance, one constraint file specifies that the paymentservice pod provides its service on port 50051 using the TCP protocol (A in Figure 16). Following the application of this constraint file, we attempted to install a malicious network policy that would allow traffic to be sent to port 8080 of the paymentservice (B in Figure 16). However, we verified that the policy enforcement system rejects this malicious policy based on the constraint file (C in Figure 16). This successful rejection demonstrates the effectiveness of KUNERVA in ensuring the enforcement of desired network policies and maintaining the intended security and isolation requirements within the containerized cluster.

C. FUNCTIONAL BENCHMARK SUMMARY

To enable the discovery of network policies within containerized cloud environments, KUNERVA relies on minimal data, including the names of the source and destination pods, as well as port information. This approach ensures the portability and compatibility of the log collector component across different environments. In our evaluation, we utilize Cilium Hubble as the network monitoring system to obtain the necessary network logs for testing. However, it is important to note that KUNERVA can seamlessly integrate with other network monitoring systems that provide the required minimal log data. This flexibility allows our framework to adapt to diverse monitoring setups and effectively leverage network logs for policy discovery in various cloud environments.

Table 2 provides an overview of the different types of network policies and sub-rules supported by KUNERVA. The network policies can be classified into three main types: L3, L4, and L7, depending on the CNI adopted within the containerized cluster. For example, Cilium offers comprehensive support for the full stack, from L3 to L7. On the other hand, default Kubernetes and Calico network policies primarily focus on L3 and L4. The policy converter component in KUNERVA plays a crucial role in converting the discovered network policies into formats compatible with Kubernetes, Cilium, and Calico, ensuring seamless integration and enforcement of the policies within the respective CNIs. This capability enables KUNERVA to accommodate a wide range of network policy requirements in different environments.

D. PERFORMANCE BENCHMARK SUMMARY

To assess the performance impact of KUNERVA, we conducted a systematic evaluation within a Kubernetes cluster. Our evaluation comprised four distinct stages in the workflow: L3-4 log collection exclusively, combined L3-4 log collection with policy discovery, L3-4 and L7 log collection only, and integrated L3-7 log collection with policy discovery. It is noteworthy that all KUNERVA components ran on the master node, with the exception of the network log

TABLE 2. Network policy types and network policy plugins supported by KUNERVA.

		K8s [5]	Cilium [24]	Calico [23]	KUNERVA
L3	Label	✓	✓	✓	✓
	CIDR	✓	✓	✓	✓
L4	Port	✓	✓	✓	✓
	Protocol	✓	✓	✓	✓
L7	HTTP		✓		✓
	FQDN		✓		✓

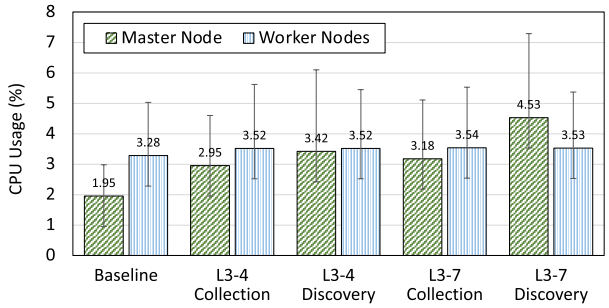

FIGURE 17. Measurements of the changes in CPU usage with different combinations of KUNERVA components.

collector, which operated alongside microservice containers on individual nodes. Throughout the evaluation, we monitored both CPU and MEM usage in 5-minute intervals for each scenario, while executing policy discovery every 10 seconds.

Figure 17 provides the average CPU usage distribution among various configurations of KUNERVA components. In the baseline scenario, microservice containers consume 3.28% of the CPU on worker nodes in total, while Kubernetes components utilize 1.95% on the master node. Upon the initiation of (L3-7) network log collection, the log collector introduces a minimal 0.25% increase in CPU usage on worker nodes, with no discernible impact on the performance of other microservices. Conversely, as L3-4 network logs reach the master node, KUNERVA designates 1% of CPU usage to store these logs within its database. The inclusion of L3-7 network logs results in a 0.23% rise in CPU usage attributed to the processing of URIs. Further, policy discovery from L3-4 logs contributes an additional 0.5% to the CPU usage. In aggregate, activating all functionalities, including L7 policy discovery, culminates in a CPU usage of 4.53%, representing a 2.58% increase compared to the baseline scenario.

Turning to Figure 18, we depict the average memory usage measurements for various KUNERVA component configurations. In contrast to the CPU usage distribution, consistent memory consumption is observed on both the master and worker nodes (400MB and 97MB, respectively), although slight increases are seen when collecting L7 logs and discovering L7 policies. As a result, these measurements reaffirm

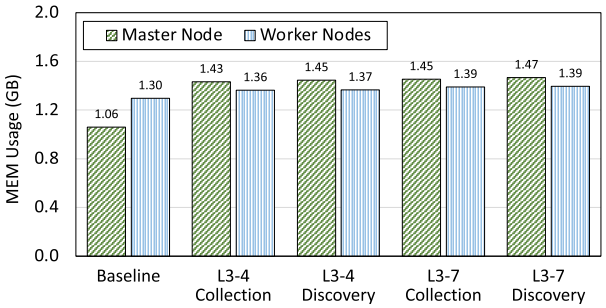

FIGURE 18. Measurements of the changes in MEM usage with different combinations of KUNERVA components.

KUNERVA's capability to deliver container-aware network policy discovery and verification with minimal overhead.

VIII. LIMITATIONS AND DISCUSSIONS

While KUNERVA demonstrates effective and efficient network policy discovery in containerized environments, there are still limitations that can be addressed for further enhancement. In this section, we highlight the current design's limitations and propose potential improvements to our framework, aiming to refine its capabilities and extend its applicability in addressing various challenges related to network policy management and enforcement.

A. SYSTEM POLICY DISCOVERY

Apart from network logs, there are some other systems such as KubeArmor [47], offering system-related logs (e.g., file access and system calls [45], [47]) by leveraging technologies like eBPF (extended Berkeley Packet Filter) and LSM (Linux Security Module). These logs provide valuable insights into container activities and runtime security enforcement. KUNERVA could enhance its capabilities by integrating with such systems to acquire these system-related logs. By leveraging these logs, our framework could extend its policy discovery to include system policies, enabling more comprehensive control over container behavior and enhancing overall security measures within the containerized environment.

B. MULTI-CLUSTER SUPPORT

With the growing demand for multi-cluster deployments in containerized cloud environments, we recognize the need to extend the capabilities of KUNERVA to support policy discovery in such environments. While the current version of our framework operates within a single cluster, we aim to develop strategies to efficiently and effectively handle the complexities that arise from managing numerous pods distributed across multiple clusters and the network logs they generate. By addressing the challenges associated with multi-cluster environments, we can provide organizations with the means to effectively manage network policies and ensure consistent security and isolation across their containerized deployments.

C. POLICY CONFLICT RESOLUTION

Primarily, we operate under the assumption that no network traffic is permitted between pods without network policies in Kubernetes environments. In practice, this can be achieved using blank selector labels within the network policy. Additionally, KUNERVA identifies network policies that have only `allow` actions, thereby minimizing policy conflicts. However, conflicts can still occur under a variety of conditions, such as when policies possess overlapping or contradictory rules. To resolve these conflicts, a variety of solutions can be deployed. For instance, certain policies may hold more importance than others based on the network's security needs and the functionalities of the applications involved. Our future work will involve identifying policy conflicts and prioritizing policies based on these considerations.

IX. CONCLUSION

In this work, we have introduced an innovative automated network policy discovery framework, called KUNERVA, designed for containerized cloud environments. KUNERVA effectively addresses the challenges posed by manual network policy management, significant contributions in container-aware policy discovery, efficient consolidation of policy rules, and seamless integration with policy enforcement systems. Importantly, KUNERVA significantly reduces the risks of misconfigurations, thereby enhancing the overall security posture of a containerized cluster. We have demonstrated the capability of KUNERVA to discover and validate network policies successfully within the context of a real-world web-based e-commerce microservices application on Kubernetes. We have also illustrated the effectiveness of our policy discovery algorithm through its application to various use cases involving real-world microservices.

APPENDIX
NETWORK POLICY SPECIFICATION

KUNERVA offers an expressive network policy model that specifies how containers should communicate with various network entities. Listing 1 illustrates a full definition of KUNERVA's network policy.

The KUNERVA network policy begins with fundamental information such as apiVersion, kind, metadata, outdated, and generatedTime. The apiVersion and kind are constants across all network policies. The metadata section includes the discovered policy name (typically a random string) and the name of the namespace to which it applies. Furthermore, we specify the policy type, either egress or ingress, along with its rule. The toPorts rule can interact with other rules. The status field indicates whether the policy is latest or outdated. When a policy is marked as outdated, the outdated field references the name of the overlapping policy, and the generatedTime field reflects the time the policy was created, based on unix seconds. The selector part is relatively straightforward, and the target pods can be specified based on labels.

```
apiVersion: v1
kind:KunervaNetworkPolicy

metadata:
  name: [policy name]
  namespace: [namespace name]
  type: [egress|ingress]
  rule: [matchLabels|toPorts|toCIDRs
        |fromCIDRs|toEntities
        |fromEntities|toServices
        |toFQDNs|toHTTPs]
  status: [outdated|latest]

  outdated: [overlapped policy name]
  generatedTime: [unix time second]

spec:
  selector:
    matchLabels:
     [key1]: [value1]
     [keyN]: [valueN]

egress:
  - matchLabels:
      [key1]: [value1]
      [keyN]: [valueN]
    toPorts:
    - port: [port number]
      protocol: [protocol]
    toCIDRs:
    - cidrs:
      - [ip addr]/[cidr bits]
    toEntities:
    - [entity]
    toServices:
    - serviceName: [service name]
      namespace: [namespace]
    toFQDNs:
    - matchNames:
      - [domain name]
    toHTTPs:
    - method: [http method]
      path: [http path]
      aggregated: [true/false]

ingress:
  - matchLabels:
      [key1]: [value1]
      [keyN]: [valueN]
    toPorts:
    - port: [port number]
      protocol: [protocol]
    toHTTPs:
    - method: [http method]
      path: [http path]
      aggregated: [true/false]
    fromCIDRs:
    - cidrs:
      - [ip addr]/[cidr bits]
    fromEntities:
    - [entity]
```

Listing 1. Kunerva network policy specification.

In the egress rule, we differentiate between seven types. Firstly, matchLabels is analogous to the selector, thus allowing specification of destination pods based on labels,

which should also encompass the namespace. ToPorts is a list of port filters, where port and protocol denote the port number and its corresponding protocol, respectively. TCP, UDP, and SCTP protocols are supported. Additionally, ToPorts should be combined with other rules, for example, matchLabels+toPorts, toCIDRs+toPorts, toFQDNs+toPorts, and matchLabels+toPorts+toHTTPs. ToCIDR rules define policies limiting external access to a specific IP range, and when combined with ToPorts rules, can refine external IP address restrictions.

ToEntities rules describe entities accessible by the selector, although they are exclusive to Cilium-based CNIs. The applicable entities include host (the local host), remote-node (other hosts in the cluster), and world (equivalent to CIDR 0.0.0.0/0). ToServices rules limit access to services operating within the cluster, although these services should not use selectors. In other words, it supports selector-less services exclusively. Consequently, to use ToServices rules, the service and its endpoints should be designated. ToFQDNs rules define policies incorporating DNS queryable domain names, with multiple distinct names currently supported in separate matchName entries. Lastly, ToHTTPs rules consist of the method and path of the HTTP protocol. If the method is omitted or left blank, all methods are permitted. Typically, ToHTTPs rules are used in combination with matchLabels and ToPorts rules. If paths are aggregated, the aggregate boolean value is set to true.

In the ingress rule, we differentiate between four types: matchLabels, toPorts, fromCIDRs, and fromEntities. These function similarly to their counterparts in the egress rule. Specifically, ToPorts rules within the ingress context represent the destination port information exposed by the selector.

REFERENCES

[1] Portworx and Aqua Security. (2019). *2019 Container Adoption Survey*. [Online]. Available: https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf

[2] Markets and Makers. (2018). *Application Container Market*. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/application-container-market-182079587.html

[3] Docker. *Accelerated Container Application Development*. Accessed: Aug. 31, 2023. [Online]. Available: https://www.docker.com

[4] StackRox. (2022). *The State of Container and Kubernetes Security*. [Online]. Available: https://thechief.io/c/editorial/state-of-kubernetes-security-report

[5] Kubernetes. *Production-Grade Container Orchestration*. Accessed: Aug. 31, 2023. [Online]. Available: https://kubernetes.io

[6] M. U. Haque, M. M. Kholoosi, and M. A. Babar, "KGSecConfig: A knowledge graph based approach for secured container orchestrator configuration," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2022, pp. 420–431.

[7] Twain Taylor. *5 Kubernetes Security Incidents and What We Can Learn From Them*. [Online]. Available: https://techgenix.com/5-kubernetes-security-incidents/

[8] E. Socchi and J. Luu, "A deep dive into Docker Hub's security landscape—A story of inheritance?" M.S. thesis, Dept. Inform., Univ. Oslo, Oslo, Norway, 2019.

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

[10] (2021). *CVE-2020-8554: Man in the Middle in Kubernetes*. [Online]. Available: https://blog.champtar.fr/K8S_MITM_LoadBalancer_ExternalIPs/

[11] ThreatPost. (2021). *'Azurescape' Kubernetes Attack Allows Cross-Container Cloud Compromise*. [Online]. Available: https://threatpost.com/azurescape-kubernetes-attack-container-cloud-compromise/169319/

[12] GateKeeper. *Policy Controller for Kubernetes*. Accessed: Aug. 31, 2023. [Online]. Available: https://github.com/open-policy-agent/gatekeeper

[13] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: Toward an open-source solution for cloud computing," *Int. J. Comput. Appl.*, vol. 55, no. 3, pp. 38–42, Oct. 2012.

[14] S. S. Yau, A. B. Buduru, and V. Nagaraja, "Protecting critical cloud infrastructures with predictive capability," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, Jun. 2015, pp. 1119–1124.

[15] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Trans. Cloud Comput.*, vol. 7, no. 3, pp. 677–692, Jul. 2019.

[16] R. N. Rajapakse, M. Zahedi, M. A. Babar, and H. Shen, "Challenges and solutions when adopting DevSecOps: A systematic review," *Inf. Softw. Technol.*, vol. 141, Jan. 2022, Art. no. 106700.

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

[18] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure DevOps," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Apr. 2016, pp. 202–211.

[19] *Report on the Enhancements of the NFV Architecture Towards 'Cloud-Native' and 'PaaS'*, document ETSI GR NFV-IFA 029, ETSI, Sophia Antipolis, France, 2019.

[20] Podman. *Pod Manager Tool (Podman)*. Accessed: Aug. 31, 2023. [Online]. Available: https://www.podman.io

[21] T. Watts, R. G. Benton, W. B. Glisson, and J. Shropshire, "Insight from a Docker container introspection," in *Proc. 52nd Hawaii Int. Conf. Syst. Sci.*, Aug. 2019, pp. 7194–7203.

[22] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "BASTION: A security enforcement network stack for container networks," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 81–95.

[23] Tigera. *Project Calico*. Accessed: Aug. 31, 2023. [Online]. Available: https://www.projectcalico.org

[24] Cilium. *API-Aware Networking and Security*. Accessed: Aug. 31, 2023. [Online]. Available: https://cilium.io

[25] Weaveworks. *Weave Net*. Accessed: Aug. 31, 2023. [Online]. Available: https://www.weave.works/oss/net

[26] G. Budigiri, C. Baumann, J. T. Mühlberg, E. Truyen, and W. Joosen, "Network policies in Kubernetes: Performance evaluation and security analysis," in *Proc. Joint Eur. Conf. Netw. Commun. 6G Summit (EuCNC/6G Summit)*, Jun. 2021, pp. 407–412.

[27] H. Kang and S. Shin, "*Verikube*: Automatic and efficient verification for container network policies," *IEICE Trans. Inf. Syst.*, vol. 105, no. 12, pp. 2131–2134, 2022.

[28] *Netfilter and IPtables*. Accessed: Aug. 31, 2023. [Online]. Available: https://www.netfilter.org

[29] IO Visor Project. *Extended Berkeley Packet Filter*. Accessed: Aug. 31, 2023. [Online]. Available: https://www.iovisor.org/technology/ebpf

[30] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using OpenFlow: A survey," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, pp. 493–512, 1st Quart., 2014.

[31] J. Gummaraju, T. Desikan, and Y. Turner, "Over 30% of official images in Docker Hub contain high priority security vulnerabilities," Banyan Secur., San Francisco, CA, USA, Tech. Rep. 5, 2015.

[32] O. Henriksson and M. Falk, "Static vulnerability analysis of Docker images," Blekinge Inst. Technol., Karlskrona, Sweden, Tech. Rep. 39, 2017.

[33] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on Docker hub," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2017, pp. 269–280.

[34] W. S. Shameem Ahamed, P. Zavarsky, and B. Swar, "Security audit of Docker container images in cloud architecture," in *Proc. 2nd Int. Conf. Secure Cyber Comput. Commun. (ICSCCC)*, May 2021, pp. 202–207.

[35] M. De Benedictis and A. Lioy, "Integrity verification of Docker containers for a lightweight cloud environment," *Future Gener. Comput. Syst.*, vol. 97, pp. 236–246, Aug. 2019.

[36] W. Luo, Q. Shen, Y. Xia, and Z. Wu, "Container-IMA: A privacy-preserving integrity measurement architecture for containers," in *Proc. Int. Symp. Res. Attacks, Intrusions Defences*, 2019, pp. 487–500.

[37] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim: OS kernel support for a low-overhead container overlay network," in *Proc. Symp. Networked Syst. Design Implement.* Boston, MA, USA: USENIX Association, 2019, pp. 331–344.
[38] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. K. Ramakrishnan, and T. Wood, "OpenNetVM: A platform for high performance network service chains," in *Proc. Workshop Hot Topics Middleboxes Netw. Function Virtualization*, Aug. 2016, pp. 26–31.
[39] T. Bui, "Analysis of Docker security," 2015, *arXiv:1501.02967*.
[40] T. Combe, A. Martin, and R. Di Pietro, "To Docker or not to Docker: A security perspective," *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 54–62, Sep. 2016.
[41] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar, "Securing Docker containers from denial of service (DoS) attacks," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Jun. 2016, pp. 856–859.
[42] Google. *Everything at Google Runs in Containers*. Accessed: Aug. 31, 2023. [Online]. Available: https://cloud.google.com/containers
[43] *Cloud Native Computing Foundation*. Accessed: Aug. 31, 2023. [Online]. Available: https://www.cncf.io/
[44] Falco. *Cloud-Native Security Tool Designed for Linux Systems*. Accessed: Aug. 31, 2023. [Online]. Available: https://falco.org/
[45] Sysdig. *Security Tools for Containers, Kubernetes, and Cloud*. Accessed: Aug. 31, 2023. [Online]. Available: https://sysdig.com
[46] C.-W. Tien, T.-Y. Huang, C.-W. Tien, T.-C. Huang, and S.-Y. Kuo, "KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches," *Eng. Rep.*, vol. 1, no. 5, Dec. 2019, Art. no. e12080.
[47] KubeArmor. *Cloud-Native Runtime Security Enforcement System*. Accessed: Aug. 31, 2023. [Online]. Available: https://kubearmor.io/
[48] Open Policy Agent. *Policy-Based Control for Cloud Native Environments*. Accessed: Aug. 31, 2023. [Online]. Available: https://www.openpolicyagent.org/
[49] S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim, "Proactive security analysis of changes in virtualized infrastructures," in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, Dec. 2015, pp. 51–60.
[50] S. Majumdar, Y. Jarraya, T. Madi, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "Proactive verification of security compliance for clouds through pre-computation: Application to OpenStack," in *Proc. 21st Eur. Symp. Res. Comput. Secur. Comput. Secur. (ESORICS)*. Heraklion, Greece: Springer, Sep. 2016, pp. 47–66.
[51] S. Majumdar, Y. Jarraya, M. Oqaily, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "LeaPS: Learning-based proactive security auditing for clouds," in *Proc. Eur. Symp. Res. Comput. Secur.* Cham, Switzerland: Springer, 2017, pp. 265–285.
[52] S. Majumdar, A. Tabiban, M. Mohammady, A. Oqaily, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi, "Proactivizer: Transforming existing verification tools into efficient solutions for runtime security enforcement," in *Proc. Eur. Symp. Res. Comput. Secur.* Cham, Switzerland: Springer, 2019, pp. 239–262.
[53] H. Kermabon-Bobinnec, M. Gholipourchoubeh, S. Bagheri, S. Majumdar, Y. Jarraya, M. Pourzandi, and L. Wang, "ProSPEC: Proactive security policy enforcement for containers," in *Proc. 12th ACM Conf. Data Appl. Secur. Privacy*, Apr. 2022, pp. 155–166.
[54] Cilium Hubble. *Network, Service & Security Observability for Kubernetes*. Accessed: Aug. 31, 2023. [Online]. Available: https://github.com/cilium/hubble
[55] P. DuBois and M. Widenius, *MySQL*. Indianapolis, IN, USA: New Riders, 2000.
[56] K. Banker, D. Garrett, P. Bakkum, and S. Verch, *MongoDB in Action: Covers MongoDB Version 3.0*. New York, NY, USA: Simon and Schuster, 2016.
[57] Google Cloud Provider. *Online Boutique*. Accessed: Aug. 31, 2023. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

SEUNGSOO LEE received the B.S. degree in computer science from Soongsil University, the M.S. degree in information security from KAIST, and the Ph.D. degree in information security from KAIST, in 2020. He is 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.

JAEHYUN NAM received the B.S. degree in computer science and engineering from Sogang University, South Korea, and the M.S. and Ph.D. degrees from the School of Computing (Information Security), KAIST. He is an Assistant Professor with the Department of Computer Engineering, Dankook University, South Korea. His research interests focus on networked systems and security. He is especially interested in performance and security issues in cloud and edge computing systems.