Helm-ET: Reducing Exposure to Lateral Movement in Kubernetes Artifacts

Jacopo Bufalino
CNAM, France
Aalto University, Finland
jacopo.bufalino@lecnam.net

Jose Luiz Martin Navarro
Universitat de València, Spain
Aalto University, Finland
jose.martinnavarro@aalto.fi

Aleksi Peltonen
CISPA Helmholtz Center for Information Security, Germany
aleksi.peltonen@cispa.de

Tuomas Aura
Aalto University, Finland
tuomas.aura@aalto.fi

***Abstract*—Modern cloud applications consist of containerized microservices deployed to a virtual computing environment, such as a Kubernetes cluster. Policies are needed to block unintended and potentially harmful interactions between the microservices, which attackers could exploit for discovery and lateral movement. However, it is challenging for cluster administrators to define strict network policies because the interactions between the components are not clearly defined. Enabling them often requires manual inspection of the declarative configuration and the source code of the applications. This paper proposes a novel approach to creating Kubernetes network policies that restricts access between microservices within a cluster. Based on the principles of modularity and information hiding, we identify service composition patterns in cloud applications and use them to create network policies. The policy generation is implemented as an open-source tool, Helm-ET, which we evaluate on 451 Helm charts across three datasets. The results show that the proposed approach can significantly reduce the internal attack surface in the Kubernetes cluster (92.71% less allowed connections), achieving comparable results to state-of-the-art tools. However, compared to other solutions, Helm-ET is faster (<100ms vs 79 seconds) and the policies before the application deployment.**

***Index Terms*—Kubernetes, Helm, Service Composition, Network Policies, Security, Microservices**

I. Introduction

Modern cloud-native infrastructures are built as collections of independently deployable microservices that communicate through application programming interfaces (APIs) [1]. Similarly to how software libraries provide reusable, self-contained components, microservices are integrated to compose intricate applications. Cloud platforms, such as Kubernetes [2], maintain the state of the containerized services based on declarative specifications, which are also modular and reusable. Deployment tools like Helm [3] introduce an additional abstraction layer with a template language for automating and customizing the application deployment and configuration.

Despite advances in cloud infrastructure technologies, recent research [4], [5], [6] shows that declarative infrastructure and application configurations often do not follow best security practices. For example, many publicly shared Helm charts lack network policies, which are essential for securing the Kubernetes cluster. This can be exploited by attackers for discovery and lateral movement [7]. Effective security policy generation is difficult for the microservice users since it requires comprehensive knowledge of the software components and their intended connections. The more dependencies an application has, the more difficult it is to implement the principle of least privilege.

One of the core challenges in securing cloud platforms is applying network isolation, which is a technique for limiting lateral movement and attacker capabilities at the network layer [8], [9]. While physical networks are often highly segmented, virtual cloud environments frequently have all services running in one open network with no internal boundaries. For example, in Kubernetes, all microservices in the same layer-3 cluster network are by default allowed to communicate with each other. Although it is technically possible to define a network policy that limits internal access, configuring it for microservice architectures remains a challenge.

In this paper, we develop a new policy configuration method to restrict access between microservices in the Kubernetes cluster using the natural boundaries of the declarative application specification. To achieve this goal, we employ a three-stage study. First, we review publicly available cloud applications and identify their network access patterns. Building upon these patterns, we define access rules to isolate the applications. Second, we implement a policy generation process based on the service composition patterns. Finally, we evaluate the solution by comparing it with state-of-the-art policy generation tools and with user-defined policies.

Our main contributions are the following:

1) We propose a new methodology to automatically generate network policies based on the naturally occurring composition patterns of cloud applications.
2) We introduce Helm-ET, an open-source implementation of the proposed approach for Helm declarative specifications.
3) We conduct a quantitative and qualitative study of our methodology using 451 publicly available Kubernetes applications and large benchmark applications. We compare the results against other state-of-the-art policy generation tools and show that our solution achieves considerable improvements.

The rest of the paper is organized as follows. Section II provides background information on cloud technologies. Section III summarizes related work. Section IV explains the goals of the proposed method and the main challenges it has to solve. Section V analyzes service composition in Helm charts and proposes access rules for them. Section VI describes the implementation. Section VII evaluates the proposed method. Section VIII discusses the results and future work. Finally, Section IX concludes the paper.

II. Background

This section provides an overview of Kubernetes and its networking model, the Helm package manager, and the concepts of modularity and information hiding as used in this paper.

A. Kubernetes and its networking model

Kubernetes [2] is an open-source orchestration tool for managing containerized applications. Its primary virtual compute units are *Pods*, which comprise one or more containers. They are deployed into a virtual environment called *cluster*. Pods are transient entities with dynamic names and addresses. Other compute resources are collections of Pods with rules that handle their scheduling, replication, and scaling. These resources can be grouped into *namespaces* for project management and resource allocation purposes. Every Kubernetes resource may also be given *labels*, which are key-value pairs. An abstraction called *Service* is used to give permanent names and, optionally, static IP addresses to sets of compute resources. Microservices in the cluster discover each other by resolving the Service name using DNS.

The Kubernetes networking model creates a *flat virtual cluster network* where, by default, all Pods can connect to each other and all Services with IP-based protocols. The easy integration of microservices over the flat cluster network is arguably one of the reasons behind the success of Kubernetes [10]. As a result, developers do not need to include network configuration for the application to work.

B. NetworkPolicy resource

Network-layer access between the compute resources in the Kubernetes cluster network can be restricted by defining a native Kubernetes resource called *NetworkPolicy*. It is typically used for network segmentation, i.e., isolating applications from each other, but it can also be used for more fine-grained access control policies between microservices. The network policy is expressed as *ingress* or *egress* rules on the individual Pods or collections. The allowed connections in these rules can be defined based on namespaces, labels, IP addresses, and port numbers.

The NetworkPolicy resource has several limitations. For instance, policies are defined on the network and transport layers and cannot be filtered by domain names or HTTP headers. Rules identify Pods based on labels but Services based on their IP addresses, which creates a confusing mixture of two different identifier spaces. Furthermore, only *allow* rules can be specified. The policies are additive, meaning that when multiple allow rules exist for the same Pod, all those connections are allowed. Somewhat unexpectedly, when there are no ingress or egress rules for a Pod, all traffic is allowed in that direction. Only when the first ingress or egress rule is added, the default policy for the specific Pod and direction changes to deny. These limitation affect the complexity of developing policies incrementally or composing policies from different sources.

C. The Helm package manager

Helm [3] is a package manager and templating language for Kubernetes that simplifies the installation and management of cloud resources. It packages applications as collections of manifests and templates, called *charts*. To deploy an application into a cluster, Helm renders the chart into YAML files with Kubernetes resource declarations. Helm charts are composable, i.e., they can import other charts as *dependencies*. This enables the reuse of microservice specifications written by third parties. The dependencies may import entire microservices or cloud applications into the new deployment. Charts are typically stored in private repositories but can also be shared via public registries, such as Artifact Hub [11]. In addition to storing charts, Artifact Hub collects and displays associated metadata, such as user rankings and author information.

D. Modularity, information hiding and module hierarchy

Modularity, in the context of complex engineering systems, fulfills three purposes: managing the system's complexity, decoupling components to enable parallel work, and accommodating future uncertainty [12]. Individual modules are designed and produced independently, with interfaces enabling their compatibility. Information hiding is a design principle of modular software structures [13]. It states that system details likely to change independently should be hidden to reduce exposure to the module interface. Information hiding aims to reduce the complexity of the whole system through two key features: the internals of each module should be easy to modify without affecting the whole system, and modules should be easy to use by users without any previous knowledge of the module's internal structure. In safe programming languages, modularity is a security feature that restricts access across module boundaries [14].

The module hierarchy [15] is a system structure [13] where modules can depend on each other, e.g., module A *uses* or *depends upon* module B. This principle enables the ability to change modules in the system. For example if we consider the hierarchical software structure as a tree, a subtree can be reused in a different application.

III. Related work

This section summarizes related research in the area of policy generation and service composition in the cloud.

A. Policy generation using source code

Generating policies based on the application source code is a popular approach in the literature. Previous research proposed using the source code to create an ontology model of the applications, which can be used to formally define their security properties [16], [17], [18]. Other approaches increase the security of the application by requiring changes to the source code [19] or the underlying cloud infrastructure [20], Li et al. [21] presented a work on automatic policy generation for Kubernetes applications called AutoArmor. Their proposal builds Istio service mesh (L7) policies based on the analysis of environment variables and the source code of cloud applications. Although their solution does not require changes in the source code, it can only be applied if all microservices are developed in a programming language supported by the tool (Java, JavaScript, Python, C#, Go and Ruby). Similarly, AutoArmor relies on the service mesh implementation.

Additionally, by avoiding using policies at the mesh level our approach avoids the performance penalty of proxying each request at each endpoint.

B. Runtime policy generation

Measuring connectivity and application requirements at runtime is another common approach to policy generation. Kubsec [22] generates AppArmor profiles for containers based on system calls. Neuvector [23] generates network rules (L3) based on application traffic inside the Kubernetes cluster[1]. The general approach is to record network traffic while the application is running to learn which policies are needed. The runtime learning approach assumes that attackers are not present during the learning phase. Although this assumption may be reasonable in some cases, it does not hold for policy updates throughout the cluster lifecycle. The main problem of using runtime policy generation is how tightly dependent are the policies to the traffic coverage used during the learning period. This can lead to valid connections being blocked because they were not active during the learning, breaking the application functionality.

In comparison, our static approach does not depend on any type of traffic to generate the network policies, it only needs access to the application configuration. Furthermore, one of the goals of our solution is to avoid application failures due to the policies generated, which translates on specific steps in our methodology.

C. Infrastructure as code analysis

Several tools analyze the declarative cloud configurations and recommend changes to harden the system [24], [25], [26], [27]. They typically compare the configuration against a list of well-known best practices. For example, Blaise and Rebecchi [4] analyze Kubernetes Helm chart deployments by translating them into topological graphs. They then derive a security score and potential attack paths based on the MITRE ATT&CK framework [28].

Inspired by these solutions, our methodology is also built taking into account best practices in network hardening in Kubernetes clusters. The key difference is that our approach directly generates network policies that can be applied to the cluster, instead of leaving the work to the cluster administrator.

[1]Neuvector can also handle L7 policies, but those are not automatically generated.

D. Cloud service composition

Cloud service composition and Service Oriented Architecture (SOA) have been extensively discussed in literature [29], [30], [31]. They mainly consider ecosystems of services located in different parts of the cloud and operated by different companies. Proposed solutions often have reliability and quality of service as their primary goals. In contrast, our work leverages service composition within a Kubernetes cluster to improve security.

IV. Reducing internal attack surface

This section presents a motivating example and the high-level goals for our work as well as the main challenges.

A. Motivating example

The MITRE ATT&CK matrix for Kubernetes [7] is a repository of attack methods categorized by techniques and objectives. As motivation for our work, we briefly explain one example of how weak network policies can be exploited to steal data. Splunk [32] is a popular observing and monitoring tool, frequently deployed in Kubernetes clusters using charts with multiple dependencies and no defined network policies. The recent *CVE-2024-53247* vulnerability [33] allows unauthenticated users to conduct remote code execution attacks as follows:

1) *Initial access:* the attacker exploits *CVE-2024-53247* to achieve remote code execution in a Kubernetes Pod running Splunk; 2) *Execution:* the attacker spawns a reverse shell, gaining access to the Pod; 3) *Discovery:* the attacker exploits the lack of policies to fingerprint the cluster network; 4) *Lateral movement:* the attacker leverages the poor isolation to gain access to other privileged pods such as the open-telemetry Pod; 5) *Impact:* the attacker gains access to sensitive data.

B. Goals and solution

The primary goal of our work is to *reduce the internal attack surface within Kubernetes clusters before application deployment* by providing a way of automatically discovering possible network connections. We aim for a balance between closing down unnecessary access and retaining the open cluster architecture that simplifies application development. To achieve this goal, we propose a novel methodology that relies on the chart declarative specification and does not depend on runtime or source code inspection of the microservices. This allows for policy generation even when the code of the microservices is not available to the user (i.e., binaries) and when users want to secure their cluster before deploying applications by enforcing sane default rules. Our tool generates Kubernetes network policies based on the static component structure of the services, which can be directly deployed to the cluster. In addition, it allows administrators to edit the generated rules (e.g., allow access to external resources) without detailed knowledge of the communication between the individual components. The principles of policy generation are discussed in Section V and the implementation in Section VI.

C. Challenges

Reducing the internal attack surface in a Kubernetes cluster is a daunting task due to the numerous challenges involved. We will now explore these challenges and our countermeasures.

a) Kubernetes flat address space: The flat network architecture in the Kubernetes cluster simplifies software integration but also makes the system more vulnerable to attacks by increasing the internal attack surface between the microservices. The need for network segmentation has been recognized [34], and the technical solution in Kubernetes is to define network policies that create boundaries between applications. However, configuring a strict policy in the flat address space is a laborious and error-prone task for several reasons. First, it is hard for the cluster maintainers to qualify unused or unintended access for third-party applications. Second, default-deny and least-privilege policies are difficult to implement due to the lack of information about the application structure and significant risk of misconfiguration [35]. Our goal is to automate the generation of a network policy that significantly reduces the internal attack surface.

b) Application complexity and third-party dependencies: It is a common practice to compose applications from microservices produced by third parties without knowing their detailed structure. Many third-party components do not provide pre-defined network policies [36], leaving the network segmentation to the cluster administrator. However, creating strict network policies would require intimate knowledge of the components, their interconnections, and network usage. We observe that a lot of information about the interconnections and network access requirements can be obtained from the declarative configurations, such as Helm charts.

c) Limitations of the declarative configuration: The declarative information in a chart only includes the exposed services and ports; it does not tell which client microservices should connect to them. This is a natural consequence of the loose coupling of microservices, where the designers of a service cannot know who will access it or what other resources there will be in the cluster. In fact, the user intent is hidden between the lines of the declarative specification and container configuration. Our goal is to infer the network connections from the chart structure and create policies that reflect the intended communication.

V. Service composition analysis

Before implementing access control rules for Kubernetes applications, it is necessary to understand how the different components interact with each other. This involves analyzing the deployment configurations and the communication patterns between services. By understanding these interactions, we can avoid unnecessary access. *Service composition* occurs when multiple charts are deployed to the same cluster or when one chart imports another.

Service composition information can be extracted from the chart dependencies and the container configuration. The first uses the Helm chart dependency tree, and the second uses the declarative configuration files and environment variables to identify the service interactions. The two approaches alone are not sufficient to capture all the communication patterns for several reasons. First, the chart dependencies method does not capture the communication patterns between compute units within the same chart and between siblings. Second, container configuration alone fails when connectivity information (i.e., hostname and port) has default values written in the source code or when the information is not available in plaintext (e.g., base64 encoded or in a template file). Finally, the container configuration method does not take into account access to external services, such as Kubernetes APIs or DNS servers.

Our methodology combines these two approaches to gain a comprehensive understanding of the communication patterns and generate network policies.

A. Composition patterns

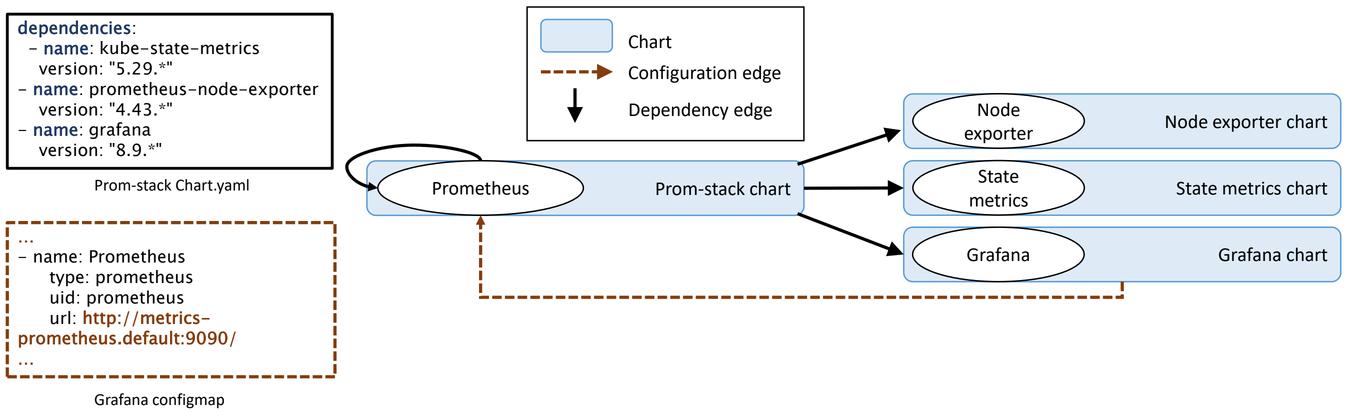
Aiming to understand how the different Kubernetes applications are composed, we analyzed a set of 446 publicly available Helm charts (see Section VII-A) from Artifact Hub. We analyzed the declarative configuration and the configuration information which, combined with the design principles described in Section II-D, provide the foundation for the network policy creation. During the analysis, we computed the two network graphs from the declarative configuration and the dependencies information. We noticed that Helm charts can have multiple levels of dependencies and complex interactions. Although dependencies and declarative configuration often form a Directed Acyclic Graph (DAG), our analysis revealed that network connections in the applications form Directed Cyclic Graphs (DCG). This is mainly due to self-loops and callbacks between applications. By looking at each application's DCG, we find the natural boundaries for network segmentation and interaction across the network segments. One such example is shown in Fig. 1 where the kube-prometheus-stack chart imports several dependencies and, among them, Grafana, which analyzes metrics from Prometheus, creating a loop.

B. Access rules

The dependency and configuration graphs serve different purposes. The former helps understand the application's logical structure and identify the components that need to communicate, and the latter provides information about the actual network connections between components. While the configuration graph can be directly translated into access policies, the dependency graph must incorporate information hiding and modularity principles to ensure proper network segmentation. Specifically, access should only be allowed from a component to itself and to its direct dependencies, keeping each module as an isolated unit with a well-defined interface. This approach aligns with the principle of least privilege, as it reduces access between components.

VI. Implementation

This section explains the policy generation process and introduces Helm-ET, its open source implementation.



```
dependencies:
  - name: kube-state-metrics
    version: "5.29.*"
  - name: prometheus-node-exporter
    version: "4.43.*"
  - name: grafana
    version: "8.9.*"
```

```
...
- name: Prometheus
    type: prometheus
    uid: prometheus
    url: http://metrics-prometheus.default:9090/
...
```

Fig. 1: Connectivity graph (DCG) of the kube-prometheus-stack Helm chart from Artifact Hub. The edges of the dependency graph are shown in black, and the configuration graph edges are shown as dashed brown lines. The boxes on the left display some of the information sources used to generate the graphs.

A. *Policy generation process*

The policy generation process takes a Helm Chart as input and produces a set of Kubernetes network policies and a configuration file as the output. The process has three steps.

Step 1 – Resource enumeration and nodes generation: We start by enumerating the dependencies, the compute resources, the corresponding Services, labels and selectors. We derive the dependency information from the Helm Chart structure and use selectors and labels to link compute units to services.

The outcome of this process is a list of nodes for the dependency and configuration graphs.

Step 2 – Dependency and Configuration graph construction: This step generates the dependency graph from the chart specifications and the connectivity graph based on the application configuration and environment variables. CC denotes the set of all components included in the chart. For every component in $c \in CC$, we collect the list of ancestors A_c and descendants D_c in the component graph. We generate the Configuration graph from environment variables, container information, ConfigMaps and Secrets objects. We match the variables with services and compute unit names. These connections form the edges in the Configuration graph.

Step 3 – Connectivity graph generation: The dependency and configuration graphs are merged into the final connectivity graph. The final connectivity graph contains the union of the edges of both graphs. This graph is used to generate the network policies in the next step.

Step 4 – Policy generation: Algorithm 1 compiles the policy generation. The detailed process is implemented as follows:

(i) The policy generation requires as an input the chart and its dependencies CC, the mappings for the ancestors A and descendants D, calculated in Step 1, and the Configuration graph from Step 2 (lines 1-4) .

(ii) The first policy created for every chart is the default deny rule $Allow(c, \emptyset)$ (line 9).

Algorithm 1 Policy generation

```
Require:
 1: $CC$, chart components
 2: $A_c$ for $c \in CC$, chart ancestors
 3: $D_c$ for $c \in CC$, chart descendants
 4: $ConfigurationGraph$
Ensure: $policy_c,\ \forall c \in CC$
 5: $AllowIngress(from, to)$, generates Ingress access rule
 6: $AllowEgress(from, to)$, generates Egress access rule
 7: $Allow(a, b) \leftarrow AllowEgress(a, b) \cup AllowIngress(b, a)$
 8: for $c \in CC$ do
 9:     $policy_c \leftarrow Allow(c, \emptyset)$ // Default Deny
10:     add $Allow(c, c)$ to $policy_c$
11:     for $d \in D_c$ do
12:         for $cu, svc_{cu} \in d$ do
13:             add $AllowEgress(c, cu)$ to $policy_c$
14:             add $AllowEgress(c, svc_{cu})$ to $policy_c$
15:         end for
16:     end for
17:     for $a \in A_c$ do
18:         for $cu, svc_{cu} \in a$ do
19:             add $AllowIngress(cu, c)$ to $policy_c$
20:         end for
21:     end for
22:     add $AllowEgress(c, ClusterDNS)$ to $policy_c$
23:     add $AllowEgress(c, Internet)$ to $policy_c$
24:     for edges $(c, dst) \in ConfigurationGraph$ do
25:         if $dst = ApiServer$ then
26:             add $AllowEgress(c, ApiServer)$ to $policy_c$
27:         else
28:             add $Allow(c, dst)$ to $policy_c$
29:         end if
30:     end for
31: end for
```

(iii) *Allow*(c,c) enables access from each chart to itself to allow computing units within the same chart to communicate (line 10).

(iv) The following rules are applied to the descendants D_c (lines 11-16) and ancestors A_c (lines 17-21) of every chart component c:

a) Access is allowed from the chart to its descendants (*AllowEgress*), both to the compute units and the services (lines 13-14).

b) Mirroring rules allow ingress (*AllowEgress*) from the compute units of the ancestor to the chart c (line 19).

(v) Default Egress connections are allowed to DNS port 53 in the `kube-system` namespace, *ClusterDNS*, and addresses outside the cluster, *Internet* (lines 22-23).

(vi) Finally, rules from the Configuration graph are computed (lines 24-30). All edges are included in the policy (line 28), but only Egress is required to access the Kubernetes APIs (lines 25-26).

The set of rules allowing outbound connections, is a necessary tradeoff between limiting connectivity and avoid breaking applications. For example, microservices use DNS to discover each other in the cluster. Many applications also need access to the Internet and other external networks for security updates, software license validation, or similar functionality. As the focus of this paper is internal attack surfaces, selectively disabling the outbound access is outside the scope of the current discussion.

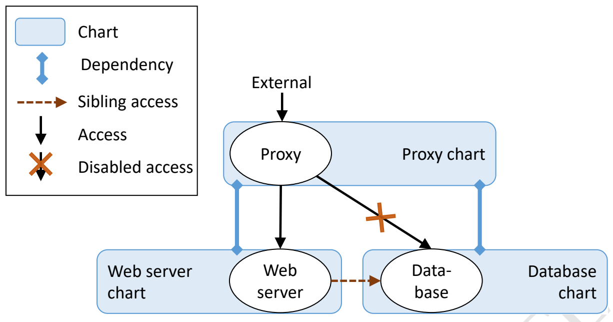
B. Helm Edge-Trimmer (Helm-ET)

Helm-ET[2] is an open source implementation of the policy generation process, written in Go. It runs as an extension of the Helm software, modifying the rendered chart based on the policy configuration. Helm-ET automatically performs all the steps of the proposed approach, including resource labeling and creating the NetworkPolicy objects.

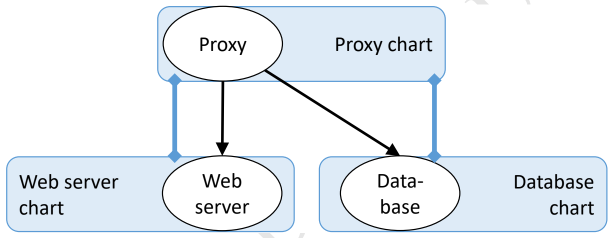
1) Policy generation: During the rendering process, Helm-ET generates NetworkPolicy objects and a policy configuration template. The network policies are automatically added to the cluster, whereas the template is saved to a separate file and may be edited by the administrator. The configuration template includes information about the dependencies and interactions between the system components. Each dependency includes a list of allowed incoming connections, and the higher-level component orchestrates sibling access between its dependencies.

2) Policy customization: Manual customization is an optional step tailored to chart developers and administrators who want to tune the results of automatic policy generation. For example, the cloud platform provider may require ingress to specific IP subnets for service discovery and metadata. The configuration template allows users to (i) enable sibling access between selected charts, (ii) restrict access to selected recursive dependencies, and (iii) restrict outbound access from selected charts to services outside the cluster or to DNS in

[2]Available at: https://github.com/kubesonde/helmet



(a) Chart with rule customization.



(b) Chart with dependencies.

Fig. 2: Comparison of chart configurations.

TABLE I: Dataset description

Dataset	Description	No. Charts w/ Network Policies	No. Charts w/o Network Policies	Total
D1	OSS microservices	0	2	2
D2	Artifact Hub	41	394	446
D3	DeathStarBench	0	2	2
		$\sum$ 41	398	450

the cluster. Fig. 2 has an example of an application with rule customization to allow sibling access and disable access from the proxy to the database. An example of policy customization is available in Appendix A.

VII. Evaluation

This section evaluates Helm-ET and compares it with the state of the art. First, we introduce the dataset and experimental setup. Then, we present the results of a security (**E1**) and a performance (**E2**) evaluation. All experiments are run in a cluster of three virtual machines, each equipped with an 8-core SkyLake processor and 32 GB RAM. We use K3s [37] version v1.28.8, and Calico version v1.27.3 as the CNI.

A. Dataset

Table I summarizes the three datasets used in our evaluation. The first dataset (**D1**) is used to compare Helm-ET with the state of the art. It comprises two OSS applications: *Bookinfo* [38] (v1.18.0) and *Online Boutique* [39] (v0.9.0), which we adapt to be deployed as Helm Charts. These applications and the experimental setup were chosen to allow comparing

TABLE II: Dataset D3 details

Application	**No. Pods**	**No. Open Ports**		
		Defined	Undefined	Total
Media Microservices	33	39	54	93
Social Network	27	33	35	68
	$\sum$ 60	72	89	161

our tool with AutoArmor[3] [40]. For traffic generation, we use the load testing tool Locust [41], with each load test running for 1000 seconds with 10 concurrent users.

The second dataset (**D2**) is used to evaluate the security improvement and policy generation time of Helm-ET. It comprises 446 Helm charts obtained from Artifact Hub [11]. We only consider applications that have received at least five stars (user endorsement indicating interest on the application) and that have been updated at least once in the year 2024.

The third dataset (**D3**, Table II) is used to evaluate scalability of Helm-ET against real-world deployments. For that, we employ the Death Star [42] benchmark, which is a collection of Helm charts that deploy large-scale microservices applications. The dataset comprises three charts: (1) Social Network, (2) Hotel Reservation, and (3) Media Microservices. We left out of the experiments Hotel Reservation because the application crashed when deployed.

Consistency: We use the same cluster setup and configuration for all tests to ensure consistency and repeatability. Each experiment begins by installing one chart into an empty cluster. Once the application is running, we measure the internal attack surface. We then repeat the same process by installing the application with the policies generated by Helm-ET. When comparing our results with the state of the art, we also make sure that the microservices are deployed to the same Kubernetes nodes on each execution to avoid unnecessary variation. Whenever possible, the charts are installed with their default configuration. Otherwise, we manually set the minimal configuration parameters required for running the application. The settings are the same when the application runs alone and when it runs with Helm-ET generated policies. The automatically generated policies are deployed without any editing.

False positives: We verify that each chart is correctly running by inspecting the *liveness and readiness* probes of each Pod before and after running the experiment. We assume that the probes have been correctly configured by the chart creators. We store the real number of allowed connections using Kubesonde [36]. Over a total of 451 Helm charts, we were able to successfully install and run 387 of them. The remaining 64 applications — all of which belonged to the D2 dataset — failed to run due to missing dependencies, misconfigured Helm charts, or other issues. This means that Helm-ET was able to generate policies for over 85% of the applications. The remaining 15% of the applications were excluded from the evaluation. The reasons for failure are detailed in Section VIII-B.

[3]AutoArmor is not publicly available. We contacted the authors but received no response. Thus, only the published policy examples could be used for the comparison.

B. *Security evaluation (**E1**)*

The security analysis consists of three parts. First, we evaluate the case when *no policies are available in the chart* (**E1a**). Then, we compare the results of Helm-ET with state-of-the-art tools such as Neuvector [23] and AutoArmor [40] (**E1b**). Finally, we evaluate Helm-ET with charts that have predefined network policies (**E1c**).

*1) Applications without network policies (**E1a**):* This experiment measures the reduction in the number of allowed connections when deploying Helm-ET on applications that do not have pre-configured network policies (dataset D2). The results are summarized in Figure 3. It shows the reduction in lateral movement and the number of allowed connections before and after applying Helm-ET. The results show that Helm-ET reduces the number of allowed connections on average by 10% for applications with one dependency, and by more than 50% for applications with four dependencies.

*2) Comparison with other tools (**E1b**):* This experiment compares the attack surface reduction of Helm-ET with other state-of-the-art tools (dataset D1). Specifically, we compare Helm-ET with AutoArmor [21] and Neuvector [23] by analyzing L3 connections (reachability of ports) for all test cases, and L7 connections (access to application-layer interfaces) for a subset of them. L3 connections are measured with Kubesonde [36], and L7 connections are obtained from the server responses to the HTTP or gRPC requests. The results are summarized in Figure 4a.

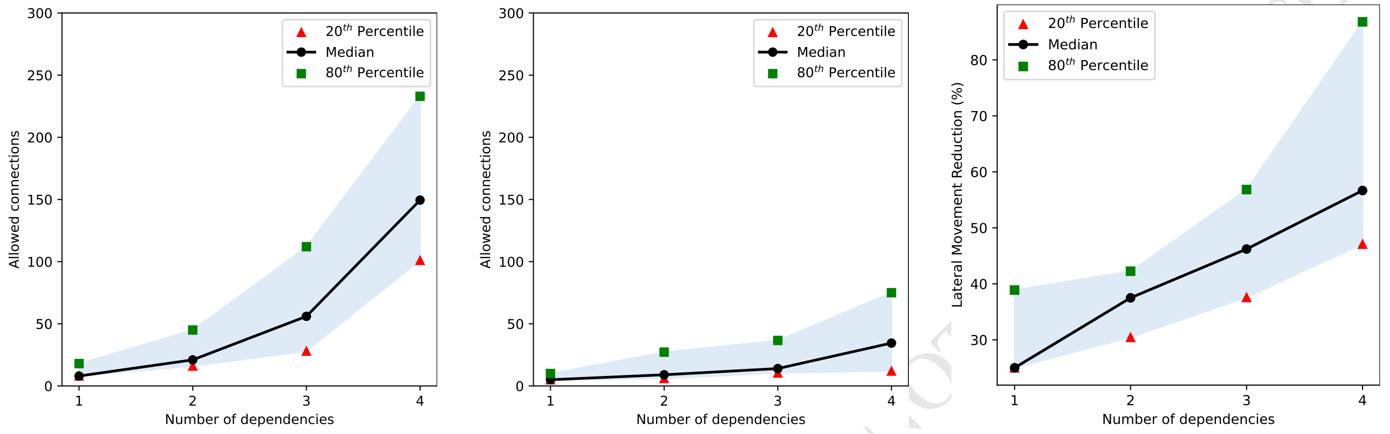
AutoArmor controls L7 connections effectively but does not prevent L3 access. This allows an attacker who is already in the cluster to scan ports and discover the services. The main reason why Helm-ET blocks fewer connections than AutoArmor is that it uses the standard Kubernetes NetworkPolicy resource, which cannot block direct access to ports that implement a Service. The direct access to Services is not considered a security weakness because the client could connect to the same destinations via the Service. However, not being able to block the direct access complicates security posture monitoring by creating seemingly unnecessary open ports.

For L3 connections, Neuvector generates network rules while running a load test for five minutes (see Table III). We use a list of endpoints from AutoArmor test data [40] for consistent comparison. As can be seen in Fig. 4a, Neuvector restricts the number of connections more than Helm-ET. However, this happens at the cost of blocking valid connections, due to the rule generation being based on load testing. For example, in the BookInfo test case, an endpoint[4] is missing from the load test, which causes it to be blocked by both AutoArmor and Neuvector.

[4]`GET /api/v1/products/<product_id>/ratings`

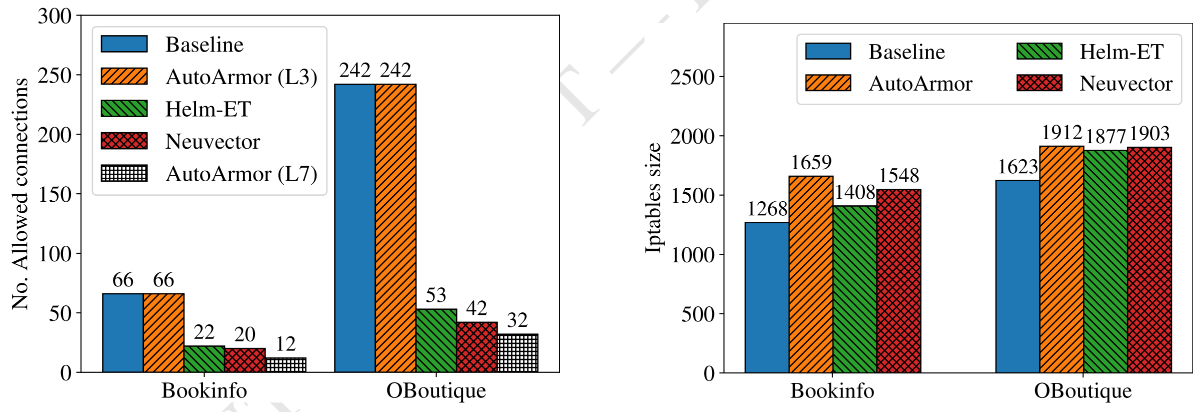
TABLE III: End-to-end latency (in milliseconds) of *Bookinfo* and *Online Boutique* with different configurations.

Application	Endpoint	Baseline	Helm-ET	Neuvector	AutoArmor
Bookinfo	GET /productpage	97	96	96	97
Online Boutique	POST /cart/checkout	34	34	36	510
	GET /cart	26	26	27	257
	GET /	32	32	33	44
	POST /cart	41	42	43	272
	POST /setCurrency	43	44	44	48
	GET /product/*	25	26	27	260



(a) Before applying Helm-ET (b) After applying Helm-ET (c) Lateral movement reduction (%)

Fig. 3: Comparison of allowed connections before and after applying Helm-ET for applications without network policies (**E1a**).



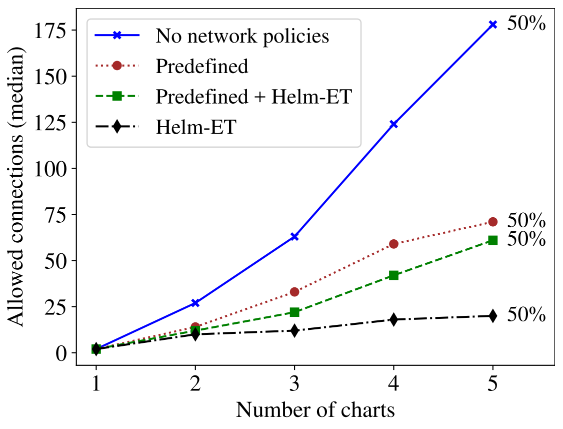
(a) Allowed L3 and L7 connections (**E1b**). (b) Number of iptables rules.

Fig. 4: Number of allowed connections and iptables rules.

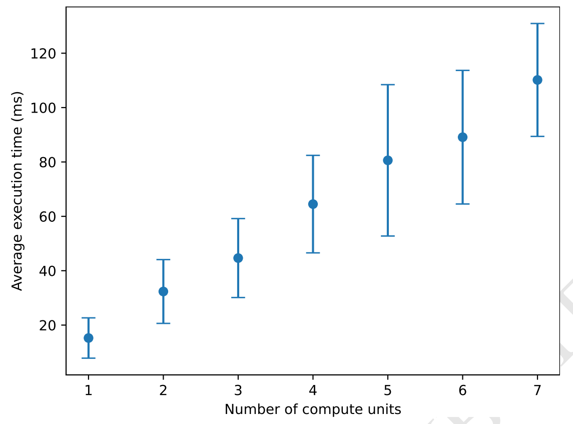
*3) Comparison with pre-defined policies (**E1c**):* This experiment compares the network policies generated by Helm-ET with those already included in applications (dataset D2), containing 41 applications from Artifact Hub [11] that include a pre-defined network policy (see Table I). We compare the number of connections when the application (1) does not have any network policy, (2) uses the predefined policies, (3) uses Helm-ET policies, and (4) uses a combination of both predefined and Helm-ET policies.

Figure 5a shows the median number of possible connections for the different policies. The horizontal axis is the number of charts in the application. The reason for the pre-defined policies not being as strict as Helm-ET policies is that the policies typically only block incoming traffic to the chart, and do not include a cluster-wide default deny policy. Helm-ET creates default deny policies for all compute resources. Note that combining Helm-ET policies with pre-defined policies results in a less strict policy because, after setting a default deny policy everywhere, the network policy rules are additive allow rules.

The number of charts with predefined policies is relatively small (41 charts), which explains the variation of the results. Nevertheless, the policies generated by Helm-ET are able to restrict the number of connections better than the chart or



(a) Median number of allowed connections for charts with NetworkPolicy (**E1c**).



(b) Helm-ET execution time, with standard deviation, for the Artifact Hub dataset (**E2b**).

Fig. 5: Execution time and number of allowed connections

application developer policies. Since the combination of both policies is additive, this experiment supports the usage of Helm-ET both for charts that have missing network policies or that include them.

*C. Application performance (**E2**)*

This experiment evaluates the application performance in three parts. First, we analyze the end-to-end latency and iptables rule size of Helm-ET compared with the state of the art (**E2a**). Second, we measure the policy generation time of Helm-ET (**E2b**). Finally, we measure the performance of Helm-ET in real-world deployments (**E2c**).

*1) End-to-end performance (**E2a**):* This experiment compares the latency of HTTP requests of Helm-ET with other state-of-the-art tool (dataset D1). As shown in Table III, the baseline, Helm-ET, and Neuvector have a negligible difference of a few milliseconds. In comparison, AutoArmor shows considerably higher latencies because the policies are enforced at the application layer, and the connections in Istio traverse two application-layer proxies.

We also measure the number of cluster-wide iptables rules, shown in Fig. 4b. The number includes iptables rules in all network namespaces on all nodes in the cluster. Interestingly, the number of rules is the lowest in Helm-ET and the highest in AutoArmor, even though the latter enforces access policies at the application layer. The explanation is that Istio in AutoArmor uses iptables to route traffic to the sidecar proxy in the network namespace of every Pod.

*2) Generation time (**E2b**):* This experiment measures the cost of the policy generation process (dataset D2). The results (see Figure 5b) show that Helm-ET analyzes the application charts and generates the network policy in the range of milliseconds. The generation time appears to increase linearly with the number of compute units in the application. Each chart in our dataset creates one or more compute units (Pods or Deployments). In comparison, the reported performance of AutoArmor [40] is four seconds for a single microservice and 79 seconds for an app with four microservices.

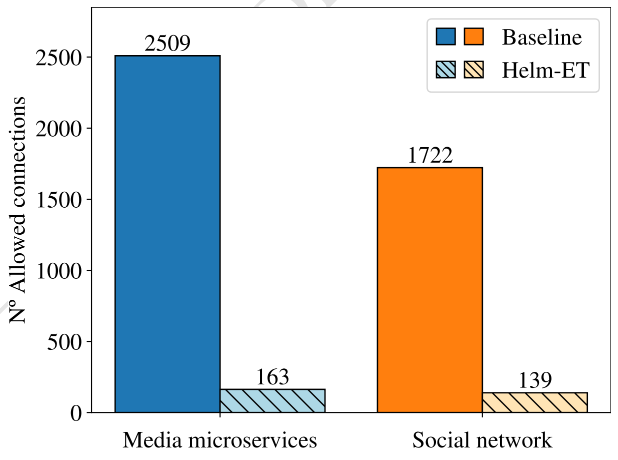


Fig. 6: Helm-ET's achieved reduction in the number of allowed connections for applications in dataset D3 (**E2c**).

*3) Scalability (**E2c**):* This experiment measures the performance of Helm-ET against large-scale deployments (dataset D3). The importance of this experiment is twofold. First, it tests the robustness of our methodology in a realistic scenario. The applications in D3 are missing network policies. As shown in Table II, manually defining the policies would be daunting due to the large number of pods and ports per application. In addition, the microservices in the dataset contain a total of 89 ports that are not part of the declarative specification and that would otherwise be blocked if e.g., a configuration-only approach was used.

This experiment also supports the fact that the effectivity of the reduction scales with the number of microservices. Figure 6 shows the difference in the number of allowed connections before and after applying Helm-ET. Although the size of D3 does not allow for a comprehensive comparison, the average reduction of 92.71% in the number of allowed connections supports the trend observed in Fig. 3c with dataset D2.

VIII. Discussion

This sections discusses the significance of the results, limitations of the proposed methods, and some directions for future work.

A. *Significance of the results*

Deployment tools, such as Helm, simplify the deployment process of cloud applications by providing a declarative and, to some extent, composable configuration language for the application configurations. Nevertheless, the security configuration of cloud services is still mostly a manual task performed by the cluster administrator. Therefore, there is demand for automated security configuration methods and tools that help the administrator.

In particular, we observe that most published applications and Helm charts do not include Kubernetes network policies, which leaves their configuration to the cluster administrator. This is not an easy task because the administrator often does not understand the detailed interaction between the third-party applications and their component microservices.

We show that reasonable network security policies can be generated from static application structure on the microservices' level. The results in Section VII indicate that the generated policies significantly reduce the internal attack surface in the Kubernetes cluster. The policy generation process is fast and understandable. For most applications, the policies can be generated automatically. For the small number of charts that require further access between dependencies, we show that simple administrator input is sufficient. The administrator can override the default policy on the component level, such as "allow access from chart A to chart B".

Our approach achieves comparable security improvement with state-of-the-art in network policy generation. The difference is that we do not rely on source-code analysis or learning from recorded traffic, which are slow and error-prone processes. Our approach is based on static analysis of the declarative application configurations, and even then, on the highest level of features in them. Comparison with predefined network policies shows that the generated policies can be stricter than policies published by application and chart developers.

B. *Limitations of automated policy generation*

There are cases where our automated policy generation methodology is unable to produce accurate policies due to missing information in the declarative application specifications. First, applications sometimes open network ports that are not declared in the chart [36]. The default deny policy of Helm-ET will prevent access to such undeclared ports. Second, some charts do not respect modularity when importing dependencies. For instance, they flatten all the dependencies on a single level. This makes it impossible for Helm-ET to generate accurate policies based on the chart structure.

Another issue arises when charts rely on undeclared dependencies. A typical example is an application that produces metrics collected by an external component in the cluster network address space. The default deny policy will block access to such undeclared dependencies, and the administrator must edit the policy to allow the access.

Fortunately, these issues are not common (as shown in Section VII) indicating that applications tend to respect the modularity and information hiding principles. However, such issues are significant because they can lead to false positives in the generated policies. The administrator must be aware of these limitations and manually adjust the policies when necessary.

We also observed that the additive policy composition in Kubernetes is not conducive to incremental network hardening. Adding rules or combining policies does not result in a tighter policy. To achieve tight network policy in Kubernetes, it is necessary to apply a cluster-wide default deny policy and then incrementally allow access. We saw this in Fig. 5a, where the combined policy is less strict. Also, while network-layer policies of Helm-ET could be combined with most L7 policies, this does not apply to the Istio service mesh because of the way it redirects ports.

C. *Future work*

The current paper is focused on the internal attack surface in the Kubernetes cluster. The microservice architecture, however, it not restricted to services inside one cluster or one cloud platform. The methodology presented in this paper could be extended to cover such distributed cloud applications.

For the longer term, there is a trend towards reuse and importing of components. The way microservices are currently deployed to the Kubernetes cluster makes only limited use of the composability. In the future, we foresee a shift towards more compositional application deployments, where major components are imported and reused based on Helm charts or similar declarative specifications. The popularity of the charts shared on Artifact Hub is one indication of such a direction. For cluster administrators and developers, using components with well-defined interfaces and hidden internals reduces the cognitive burden of the system design [29]. Thus, research is needed on the semantics of composable application specifications and on security enforcement based on them.

IX. Conclusion

This paper presents a novel approach to creating network policies for Kubernetes clusters. The work was motivated by the lack of isolation boundaries in cloud applications, with less than 10% of the tested applications providing a network policy. We developed a methodology for generating network-layer isolation policies from the compositional structure of the application. We implemented the methodology for Helm charts and evaluated it with an extensive set of published charts. The approach can be used in any Kubernetes cluster, without installing new resources and with negligible latency. The evaluation results show that our approach can automatically apply policies to over 85% of the top downloaded Helm charts and reduce the number of unnecessary connections by 50% on average.

REFERENCES

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Springer, 2017.

[2] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

[3] Helm, "Helm: The package manager for Kubernetes," 2024, Accessed on April 17, 2024. [Online]. Available: https://helm.sh

[4] A. Blaise and F. Rebecchi, "Stay at the Helm: secure Kubernetes deployments via graph generation and attack reconstruction," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE, 2022, pp. 59–69.

[5] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 164–175.

[6] M. Chiari, M. De Pascalis, and M. Pradella, "Static analysis of infrastructure as code: a survey," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2022, pp. 218–225.

[7] Y. Weizman, "Secure containerized environments with updated threat matrix for Kubernetes," March 2021, Accessed on April 17, 2024. [Online]. Available: https://www.microsoft.com/en-us/security/blog/?p=93183

[8] OWASP, "OWASP: Cloud-native application security top 10," 2022, Accessed on April 17, 2024. [Online]. Available: https://owasp.org/www-project-cloud-native-application-security-top-10

[9] National Security Agency (NSA) and Cybersecurity and Infrastructure Security Agency (CISA), "Kubernetes hardening guidance," 2022, Accessed on April 17, 2024. [Online]. Available: https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/2716980/nsa-cisa-release-kubernetes-hardening-guidance

[10] Kubernetes, "Kubernetes design proposals," 2017, Accessed on April 17, 2024. [Online]. Available: https://github.com/kubernetes/design-proposals-archive/blob/main/network/networking.md

[11] The Linux Foundation, "Artifact Hub," 2024, Accessed on April 17, 2024. [Online]. Available: https://artifacthub.io

[12] C. Y. Baldwin and K. B. Clark, *Modularity in the Design of Complex Engineering Systems*. Springer, 2006.

[13] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of ACM*, vol. 15, no. 12, pp. 1053–1058, 12 1972.

[14] A. Mettler, D. A. Wagner, and T. Close, "Joe-E: A security-oriented subset of Java," in *Proceedings of the 2010 Network and Distributed System Security Symposium*, vol. 10. The Internet Society, 2010, pp. 357–374. [Online]. Available: https://www.ndss-symposium.org/ndss2010/

[15] E. W. Dijkstra, "The structure of "THE"-multiprogramming system," *Commun. ACM*, vol. 26, no. 1, pp. 49–52, January 1983.

[16] C. Banse, I. Kunz, A. Schneider, and K. Weiss, "Cloud property graph: Connecting cloud security assessments with static code analysis," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021, pp. 13–19.

[17] N. Chondamrongkul, J. Sun, and I. Warren, "Automated security analysis for microservice architecture," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2020, pp. 79–82.

[18] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, October 2020, pp. 443–458. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/ghavanmnia

[19] K. A. Torkura, M. I. Sukmana, and C. Meinel, "Integrating continuous security assessments in microservices and cloud native applications," in *Proceedings of The 10th International Conference on Utility and Cloud Computing*. ACM, 2017, pp. 171–180.

[20] Y. Sun, S. Nanda, and T. Jaeger, "Security-as-a-service for microservices-based cloud applications," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015, pp. 50–57.

[21] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic policy generation for inter-service access control of microservices," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021, pp. 3971–3988. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/li-xing

[22] H. Zhu and C. Gehrmann, "Kub-Sec, an automatic Kubernetes cluster AppArmor profile generation engine," in *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE, 2022, pp. 129–137.

[23] Suse, "NeuVector: Full lifecycle container security," 2024, Accessed on April 17, 2024. [Online]. Available: https://www.suse.com/neuvector

[24] Bridgecrew, "Checkov: Policy-as-code for everyone," 2024, Accessed on April 17, 2024. [Online]. Available: https://checkov.io

[25] Aquasecurity, "kube-hunter: Hunt for security weaknesses in Kubernetes clusters," 2024, Accessed on April 17, 2024. [Online]. Available: https://github.com/aquasecurity/kube-hunter

[26] Zegl, "kube-score: Kubernetes object analysis with recommendations for improved reliability and security," 2024, Accessed on April 17, 2024. [Online]. Available: https://github.com/zegl/kube-score

[27] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source Kubernetes manifests: An empirical study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–36, 2023.

[28] MITRE, "MITRE ATT&CK v12," October 2022, Accessed on April 17, 2024. [Online]. Available: https://attack.mitre.org/versions/v12

[29] A. Vakili and N. J. Navimipour, "Comprehensive and systematic review of the service composition mechanisms in the cloud environments," *J. Netw. Comput. Appl.*, vol. 81, no. C, pp. 24–36, March 2017.

[30] B. Martino, G. Cretella, and A. Esposito, "Cloud services composition through cloud patterns: a semantic-based approach," *Soft Comput.*, vol. 21, no. 16, pp. 4557–4570, August 2017.

[31] A. Jula, E. Sundararajan, and Z. Othman, "Review: Cloud computing service composition: A systematic literature review," *Expert Syst. Appl.*, vol. 41, no. 8, pp. 3809–3824, June 2014.

[32] Cisco Systems, Inc., "Splunk: Data analytics and security platform," 2025, Accessed on Feb 25, 2025. [Online]. Available: https://www.splunk.com

[33] Splunk Inc., "Remote code execution through deserialization of untrusted data in Splunk Secure Gateway app (CVE-2024-53247)," 2024, Accessed on February 25, 2025. [Online]. Available: https://advisory.splunk.com/advisories/SVD-2024-1205

[34] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci, "Understanding the security implications of Kubernetes networking," *IEEE Security & Privacy*, vol. 19, no. 5, pp. 46–56, 2021.

[35] D. Lewis and R. de Jong, "Kubernetes network policies done the right way," 2025, Accessed on February 25, 2025. [Online]. Available: https://isovalent.com/books/kubernetes-network-policies-done-the-right-way-by-isovalent/

[36] J. Bufalino, M. Di Francesco, and T. Aura, "Analyzing microservice connectivity with Kubesonde," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. ACM, 2023, pp. 2038–2043.

[37] Rancher Labs, "K3s: Lightweight Kubernetes," 2024, Accessed on April 18, 2024. [Online]. Available: https://k3s.io

[38] Istio, "Bookinfo application," 2024, Accessed on April 18, 2024. [Online]. Available: https://istio.io/latest/docs/examples/bookinfo

[39] GoogleCloudPlatform, "Microservices demo," 2024, Accessed on April 18, 2024. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[40] X. Li, Y. Chen, and Z. Lin, "Towards automated inter-service authorization for microservice applications," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, ser. SIGCOMM Posters and Demos '19. ACM, 2019, pp. 3–5.

[41] Locust, "Locust," 2024, Accessed on April 17, 2024. [Online]. Available: https://locust.io

[42] Y. Gan, C. Waldspurger, S. Venkataraman, Q. Li, N. Satish, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019, pp. 3–18.

Appendix A
Policy customization

Listing 1 shows an extract of the automatically generated policy for the charts in Fig. 2b. Listing 2 is an extract of an administrator-edited policy configuration of the default template in Listing 1. The edited charts are depicted in Fig. 2a. To reflect the communication patterns in the application, the administrator has added configuration lines to enable sibling access from the web server to the database and commented out lines to disable direct access from the proxy to the database. Helm-ET generates the NetworkPolicy object from this edited policy description.

Listing 1: Helm-ET generated policy for Figure 2b

```
Name: "proxy-policy"
ComponentSelector:
  chart: "proxy"
Ingress:
  Allow:
    Resources:
      - Ports:
          - port: "80"
            protocol: "TCP"
Interactions:
  - From:
      chart: "proxy"
    To:
      chart: "webserver"
  - From:
      chart: "proxy"
    To:
      chart: "database"
---
...
```

Listing 2: Administrator-edited policy for Figure 2a

```
Name: "proxy-policy"
ComponentSelector:
...
Interactions:
  - From:
      chart: "proxy"
    To:
      chart: "webserver"
  # Enable communication between siblings:
  - From:
      chart: "webserver"
    To:
      chart: "database"
  # Disable direct access from proxy to database:
  # - From:
  #     chart: "proxy"
  #   To:
  #     chart: "database"
---
...
```