Limits of Service Discovery: Investigating Automatic Microsegmentation Policy Generation for Microservices

Conor Kotwasinski*, Li-Kang Tan*, Hongyi Charles Zhou* and Yan Chen†
Department of Computer Science, McCormick School of Engineering and Applied Science
Northwestern University, Evanston, IL 60208
Emails: conorkotwasinski2024@u.northwestern.edu
likang.tan@u.northwestern.edu
charleszhou@u.northwestern.edu
ychen@northwestern.edu

***Abstract*—Microsegmentation policies are an essential aspect of securing microservices-based architectures. These policies define the communication rules between services and restrict access to sensitive data. However, generating these policies can be a challenging and time-consuming task, especially in large-scale microservices environments. Service discovery mechanisms can help to address this challenge by providing insights into the communication patterns and invocation relations between services. By leveraging service discovery tools and visualization techniques, it is possible to automatically and statically evaluate these patterns and generate microsegmentation policies that improve the security and resilience of microservices-based architectures.**

***Index Terms*—Microservices, Microsegmentation, Service Discovery, Network Security, Kubernetes, Zookeeper, Eureka**

I. Introduction

Microservices-based architectures have become increasingly popular in recent years due to their ability to improve agility, scalability, and fault tolerance [21]. Microservices are a software development approach that involves breaking down large applications into smaller, independent services that can be developed, deployed, and scaled independently. However, manually managing communication between these services and ensuring their security can be a challenging task.

One of the critical aspects of securing microservices-based architectures is the generation of microsegmentation policies. These policies define the communication rules between services and restrict access to sensitive data. However, generating these policies can be a time-consuming and challenging task, especially in large-scale microservices environments [6].

Service discovery mechanisms can help to address this challenge by providing insights into the communication patterns and invocation relations between services. Service discovery tools such as Zookeeper and Eureka allow services to register themselves with a central registry and discover other services dynamically. Visualization tools such as Cilium Hubble provide real-time insights into the communication patterns between services.

*The authors contributed equally to this work.

In this context, this paper presents our project that aims to understand microservice architecture and service discovery mechanisms and automate the process of generating microsegmentation policies for microservice architectures. The project involved literature reviews, experimentation, implementation, and evaluation.

The literature review task focused on understanding the existing microservice architecture and service discovery mechanisms. The experimentation task involved deploying and managing microservice applications on a popular container orchestration platform - *Kubernetes*. Visualization tools such as Cilium Hubble were used to understand communication patterns between services after setting up service discovery tools.

We also implemented algorithms to identify service registration and discovery-related code snippets and used trace tools to generate a trace of the calls between different services. Based on these insights, we was able to generate microsegmentation policies that improved the security and resilience of microservices-based applications.

Overall, our project provides valuable insights into microservice architectures and service discovery mechanisms that can be applied in real-world scenarios to improve the security and resilience of microservices-based architectures. The rest of this paper is organized as follows: Section II provides background information on microservice architecture and service discovery mechanisms; Section III provides motivation on the issues this project is trying to solve; Section IV describes related work in this area; Section V presents the reasoning behind the programming language and framework that we chose to use in this project; Section VI describes our approach towards this problem; Section VII presents our results; Section VIII discusses our findings; finally, Section IX concludes this paper with future research directions.

II. Background

A. Microservices

Microservices are a software development technique that organizes an application as a group of loosely coupled services that communicate through well-defined interfaces [1]. The concept of microservices has been around for several decades, but it was not until the rise of cloud computing and containerization technologies that microservices became a practical solution for building large-scale applications. It gained popularity in recent years due to its ability to improve agility, scalability, and fault tolerance compared to monolithic applications.

1) Microservices Architecture: Microservices Architecture is a distributed application whose components are consisted of individual, loosely coupled microservices [2]. By separating a monolithic application, developers can focus on improving individual components of the application, and scale them up and down in response to changes in demand.

However, the microservices paradigm also creates challenges such as managing communication between services in large-scale environments, testing and monitoring services, and securing microservices. To manage communication between services, a service discovery mechanism is necessary to enable services to discover each other when there is a need to call other services. Testing and monitoring microservices can be more complex due to the need to test each service independently, and the potential for interactions between services that may cause issues. Securing microservices also requires careful configuration of network policies to manage communications between services and limit unauthorized access [2].

2) Common components of Microservices: There are several common components for managing microservices architecture challenges, including:

a) Service Mesh: A service mesh is a dedicated infrastructure layer that manages service-to-service communication within a microservices architecture. It provides features such as service discovery, load balancing, traffic routing, and security.

b) API Gateway: An API gateway is a service that sits between the client and the microservices, acting as an entry point to the system. It provides a single interface for clients to interact with the microservices, and can handle tasks such as authentication, rate limiting, and caching.

c) Containerization: Containerization is a technology that allows services to be packaged into lightweight, portable containers that can be deployed on any infrastructure. It provides benefits such as consistency in deployment environments and easier management of service dependencies.

d) Orchestrators: Orchestration systems, such as *Kubernetes* or *Docker Swarm*, automate the deployment, scaling, and management of containerized microservices. They provide features such as service discovery, load balancing, and self-healing to ensure the reliability and availability of microservices.

B. Service Discovery Mechanisms

Service discovery mechanisms provide a solution to the challenges of service discovery in microservices architecture. Typically, these mechanisms involve a central registry or directory that maintains information about available services and their locations. The central registry or directory can be deployed as a standalone service or incorporated as part of an existing service, such as a load balancer.

The dynamic nature of microservice architecture means that an instance of a service can be running on any host or port, and the availability of instances cannot be guaranteed as a service may be scaled up or down. This dynamic characteristic poses a challenge in service discovery, requiring the deployment of a service discovery mechanism to enable services to discover each other when there is a need to call other services.

1) Common methods of Service Discovery: There are several common methods to perform service discovery in microservices architecture, including:

a) DNS-Based Service Discovery: DNS-based service discovery allows services to be registered with a DNS server, which resolves service names to IP addresses. This approach provides a simple, scalable solution for service discovery but can have limitations in larger environments with multiple DNS servers.

b) Client-Side Service Discovery: In this approach, each client maintains a local registry of available services, which it uses to locate services as needed. This approach provides more flexibility than DNS-based service discovery but requires more coordination between clients.

c) Server-Side Service Discovery: In this approach, a central registry or directory maintains information about available services and their locations, which can be used by clients to locate services. This approach provides a more centralized solution but can have limitations in larger environments with multiple registries or directories.

C. Common Implementations for Service Discovery

There are several common implementations for service discovery in microservices architecture, each with its own advantages and disadvantages. Some popular implementations include:

a) Zookeeper: Zookeeper is a centralized service discovery solution that provides a hierarchical namespace for managing distributed systems. It is designed to be highly available and can handle large-scale environments. Zookeeper is often used in combination with Apache Kafka for distributed messaging and stream processing.

b) Kubernetes default (kube-dns): Kubernetes uses a server-side service discovery mechanism by default. This is implemented through the Kubernetes Service object and the kube-dns service, which provides a DNS-based interface for discovering services within a Kubernetes cluster. This approach provides a simple and scalable solution for service discovery in Kubernetes clusters.

c) Eureka: Eureka is an open-source service discovery solution developed by Netflix. It is designed to be highly available and can handle large-scale environments. Eureka provides a REST-based interface for discovering services and integrates well with other Netflix OSS components such as Hystrix and Ribbon.

d) Consul: Consul is a service mesh solution that provides service discovery, service mesh, and configuration management functionality. It provides a centralized registry of services and supports multiple service discovery mechanisms, including DNS-based service discovery and HTTP-based APIs. Consul also provides features such as service health checking and distributed key-value store.

e) etcd: etcd is a distributed key-value store that can be used for service discovery, configuration management, and coordination in distributed systems. It provides a simple API for storing and retrieving data, and supports features such as service discovery and leader election. etcd is often used with *Kubernetes* as a backend for storing configuration data.

D. Microsegmentation Policies

Microsegmentation is a network security technique that enables organizations to divide their networks into smaller, more granular segments, allowing them to better control access to critical assets and reduce the attack surface of their networks. Microsegmentation policies are a key component of this approach, as they define the rules that govern communications between different segments of the network.

In terms of microservices architecture, microsegmentation policies can be implemented at various levels, including the host, container, and network levels. At the host level, microsegmentation policies can be enforced using tools like firewalls or security groups, which restrict network traffic based on IP addresses, ports, and protocols. At the container level, microsegmentation policies can be enforced using container network plugins, which enable policies to be defined and enforced for each container. At the network level, microsegmentation policies can be enforced using software-defined networking (SDN) technologies, which enable network traffic to be directed and controlled using policies defined in a central controller.

Kubernetes provides built-in support for microsegmentation policies through its network policies feature. *Kubernetes* network policies enable cluster administrators to define and enforce policies that control traffic between pods in a *Kubernetes* cluster. Network policies are implemented using *Kubernetes* network plugins, known as Container Network Interfaces (CNIs), which enable policies to be defined and enforced at the network and transport level (Layer 3 and 4) [33].

III. Motivation

Generating effective microsegmentation policies for microservices-based architectures is a challenging task, with several key obstacles to overcome. One of the primary challenges is ensuring that the generated policies are correct and consistent with the intended security goals and requirements. Another challenge is scalability, as the complexity of microservices-based applications can quickly grow and become difficult to manage.

To address these challenges, policy generation needs to be automatic [6], and generated policies need to optimize for performance and security. Moreover, microservices-based applications are often dynamic and constantly evolving, necessitating policies that are adaptive and able to handle changes in the application topology and behavior. As such, policy generation for microservices-based architectures requires a comprehensive and holistic approach that addresses these challenges and provides scalable, adaptive, and transparent policy management solutions.

Generating security policies automatically in distributed systems has been a challenging task. Efforts in this regard can be categorized into four categories based on how they acquire logic or security intent. Document-based approaches [12]–[15] utilize natural language processing to infer security policies from application documents that precisely capture the high-level intentions of developers. However, extracting them reliably is not simple, and there are limitations to their accuracy, scalability, and agility. As an example, Text2Policy [14] attained an average recall of 89.4%. History-based approaches [16]–[18] mine security policies from historical operations, while model-based approaches formally model software behavior and generate security policies accordingly. However, these approaches also have their limitations. For example, P-DIFF [18] infers access control rules by monitoring access records, but its average accuracy is only 89%. On the other hand, model-based [19], [20] techniques explicitly model the behavior of software and provide security policies as a result. However, building and updating the system model is seldom agile or scalable when supporting regularly iterated and massive microservice applications. The final approach, proposed by Li *et al.* [6], is static analysis, leveraging the small and universally applicable library imports of microservices to create policies. With this method, AUTOARMOR was able to generate service level (L7) policies which it deploys through the Istio proxy service while maintaining the full functionality for the services [6].

Our decision to adopt the static-analysis approach utilized by AUTOARMOR was motivated by its proven efficacy and accuracy in generating policies for microservices. As mentioned in Li *et al.*'s next steps [6], our goal is to integrate docker images into our analysis, as some services can be directly implemented through images, without requiring the source code to be built.

In addition, we plan to utilize a service registry to map a pod or instance in the call graph, leveraging Kubernetes pods' ability to call servers outside of Kubernetes. This information can be stored within Kubernetes as a service registry pod or outside of Kubernetes as a service registry server. However, it's important to note that AUTOARMOR's L7 policies are currently managed through Istio's proxy service, which is not applicable to servers operating outside of Kubernetes [6].

To further strengthen our security measures, we have decided to replace L7 policies with L3 and L4 policies. Although Istio provides L7 policies, it acknowledges that a compromised or maliciously behaving service instance may bypass the proxy [22]. Incorporating L3 and L4 policies will enable us to mitigate such occurrences and substantially improve our overall security posture.

IV. Related Work

Minna *et al.* [3] discuss the security implications of *Kubernetes* (K8s) networking components, which have been largely unexplored. The authors highlight that the mental model of networking between microservices, derived from physical networks, significantly departs from reality. To address this gap, Minna et al. propose a dynamic risk analysis framework for K8s networking components that consider the interplay between network policies and network traffic. The authors' work builds upon previous research on software-defined networks (Yoon *et al.*, [4]) and security enforcement network stacks for container networks (Nam *et al.*, [5]).

Li *et al.*, [6] proposed an approach for automatic policy generation of inter-service access control in microservices. Their system generates fine-grained access control policies within Istio based on the invocation logic among microservices, using a combination of static analysis and dynamic tracing. The authors evaluated their approach on a set of microservices and demonstrated that it can effectively generate access control policies with high accuracy while reducing the manual configuration effort required for access control.

V. Overview

A. Initial Approach with kube-dns

Our initial approach was to deploy five microservices applications for evaluation, as used by Li *et al.* [6]. Our *Kubernetes* cluster successfully deployed *Bookinfo* [7] and *Online Boutique* [8] without any issues, but *Pitstop* [9], *Sock Shop* [10] faced some dependency issues and *Sitewhere* [11] faced PodDisruptionBudget issues and could not be deployed. We analyzed the code for these applications and discovered that the *Kubernetes* default kube-dns method handles the service discovery process. The kube-dns method is a DNS-based solution, enabling clients to directly query other services using regular URLs. The existing request extraction mechanism described Li *et al.* [6] sufficed in capturing the invocation relationship between microservices within the applications. However, we found that this approach of service discovery is highly dependent on the *Kubernetes* environment and limits the benefit of service discovery in generating policies. As *Kubernetes* manages the lifecycle of individual instances (pods) of these services and the network policies defined based on the service names, the use of service discovery in generating policies is not as prominent. Therefore, we decided to focus on service discovery mechanisms, such as Zookeeper and Eureka, that are not dependent on *Kubernetes*.

B. Java

Developers of microservices have the option to utilize a variety of programming languages for their projects. To optimize our investigation, we decided to concentrate on one specific language, enabling us to tailor our tool to leverage language-specific features, libraries, and frameworks. This approach would yield a more efficient and effective policy generation tool for microservices developed in the chosen language, offering enhanced performance and flexibility while only requiring the pod's image.

We selected Java as our primary focus, as it was the most widely used programming language among microservices developers globally in 2022, with a 34% popularity rate [23]. This statistic suggests that a significant portion of microservices projects are likely developed in Java. Consequently, by building a tool to generate policies for Java-based microservices, we would address the requirements of a large number of developers in this field.

Upon deciding to focus on Java, we then proceeded to explore various options for the service registry to be employed in conjunction with the Java code.

C. Limitations of Zookeeper

Zookeeper is a distributed coordination system that is widely used in large-scale distributed systems for maintaining configuration information, naming, providing distributed synchronization, and group services [24]. Although Zookeeper can be used for service discovery, it does not have a fixed method for performing this task. Service discovery is the process of locating and accessing available services in a distributed system, and the best approach depends on the specific requirements of the system.

One of the reasons why Zookeeper does not have a fixed method for service discovery is that it is designed to be flexible and extensible. Zookeeper provides a set of primitives and APIs that developers can use to build their own custom solutions on top of the core Zookeeper key-value pair storage functionality [24]. This allows developers to tailor their service discovery solutions to the specific needs of their systems.

Zookeeper is used in many distributed systems for various purposes. For instance, Hadoop uses Zookeeper to manage configuration information for its distributed file system, Apache Kafka uses Zookeeper to manage leader election for its distributed messaging system, Apache HBase uses Zookeeper to manage distributed locks for its distributed database system, and Apache SolrCloud uses Zookeeper to manage group membership in its distributed search system [25].

Another reason why Zookeeper does not have a fixed method for service discovery is that there are many different ways to implement service discovery, and no one approach is perfect for every situation. Some systems may store the Znodes containing the service registry in different patterns of Znode paths, while others may use different data formats for the service registry data in each Znode. Zookeeper can be used to implement any of these approaches because it is designed to be used to build a custom solution that is tailored to the specific needs of the system.

TABLE I
SPRING-CLOUD-ZOOKEEPER-SERVICE-DISCOVERY-DEMO

ZNode Path
/services/GreetingMicroservice/821ff4c4-9363-4572-aa71-bc893fcd2b54
/services/GreetingConsumer/83707936-c990-4ef5-8e72-211ab38671d4

To explore the potential configuration of Zookeeper, we first dumped the key-value pairs of a Zookeeper instance in Santiago Gonzalez Toral's demonstration of a Spring Cloud application implementing service discovery using Zookeeper [26]. To illustrate the arrangement of Znodes in the Zookeeper instance, we illustrated the architecture in Table I. This is an example of Spring Cloud's official implementation.

When the Spring Cloud library registers a service with ZooKeeper, it creates a Znode with a path that includes the service name located in the `Application.yml` file of each service annotated with `@EnableDiscoveryClient`, followed by a slash and a unique identifier for the instance of that service. This unique identifier is known as the "instance ID" and is generated by the Spring Cloud instance. The instance ID is a randomly generated alphanumeric string that identifies a particular instance of a service. It is used by Spring Cloud to ensure that each instance of a service has a unique registration in the ZooKeeper service registry. The format of the instance ID is `{hostname}-{random string}`.

TABLE II
APACHE-ZOOKEEPER-SERVICE-DISCOVERY-EXAMPLE

ZNode Path
/services/service1/node
/services/service2/node

Before we solved the issues in Toral's demonstration, we first developed our own demonstration of utilizing Zookeeper for service registration [27]. This coincidentally exposed the first major problem with using Zookeeper for service registration. There is not a standard method for the format of the service registration data within a Znode. To visualize the arrangement of Znodes in our demonstration's Zookeeper instance, we visualized the architecture in Table II. What separates this example from others is the demonstration of Zookeeper operating outside of Kubernetes' orchestration and directly routing Docker containers in the same network to each other.

Between both of these Zookeeper demonstrations, the format for the Znode path of each pod or container generally remains the same. Under the root directory is the `/services` directory, and under the `/services` directory is the directory for each service. Under the directory for the service is a node for each instance of that service, whether it is a pod in Kubernetes or a container in the network. However, in both of these scenarios, we are forgetting the case where a service is only one bare metal server.

TABLE III
ZOOKEEPER-STORING-SERVER-LOCATIONS

ZNode Path
server1/address
server2/domain
server3/address

From Uber and Twitter's own blog posts, both large enterprises implement Apache Zookeeper for service discovery to manage and monitor bare metal servers [28], [29]. Table III refers to the potential Zookeeper layout a realistic Zookeeper end-user might employ to store the key-value pairs of the IP addresses and ports or domains and ports of all of their bare metal servers; however, that is completely arbitrary since ultimately they would design the Znodes to fit their own code. Since the deployment of Znodes is so open-ended and we cannot know how every end-user deploys their Znodes since it is end-user specific, there is no universal way to query the service registry in Zookeeper.

Although the structure of Znodes is entirely end-user dependent, the content inside the Znode is also entirely end-user dependent because the end-user also decides what content is stored inside a Znode and how develops their code specifically to how they decided to store data inside the Znode.

This makes it hard to engineer an automated way to generate policies, because different applications would have different calling conventions in their code. For example, Toral's demomstration uses `@GetMapping("/")` and `@GetMapping("/GetGreeting")` while our demonstration uses `zk.exists("/services/service1")` [26], [27].

In Toral's demonstration, the Spring Cloud annotation `@EnableDiscoveryClient` uses the Spring Cloud methodology of storing data about an instance in a Znode. This Znode contains information such as the hostname, port number, metadata, and status of the service instance. The specific data stored in the Znode includes:

- `id`: The unique identifier of the service instance, which is generated by Spring Cloud.
- `name`: The name of the service, which is specified by the `spring.application.name` property in the service's configuration.
- `address`: The IP address or hostname where the service instance can be reached.
- `port`: The port number on which the service instance is listening for requests.
- `metadata`: Any additional metadata that the service instance wants to store, such as version number, environment, etc.
- `status`: The status of the service instance, which is typically "UP" if the instance is healthy and able to serve requests.

All of this information is stored as a set of key-value pairs in the Znode associated with the service instance. This data is used by other services and clients to discover and communicate with the service instance. For example, an instance ID of the GreetingConsumer service is `821ff4c4-9363-4572-aa71-bc893fcd2b54`, and the Znode would contain:

```
{
  "name": "GreetingMicroservice",
  "id":
      "821ff4c4-9363-4572-aa71-bc893fcd2b54",
  "address": "microservice",
  "port": 8080,
  "sslPort": null,
  "payload": {
    "@class":
        "org.springframework.cloud.zookeeper.di
        scovery.ZookeeperInstance",
    "id": "GreetingMicroservice:8080",
    "name": "GreetingMicroservice",
    "metadata": {
      "instance_status": "UP"
    }
  },
  "registrationTimeUTC": 1678597749551,
  "serviceType": "DYNAMIC",
  "uriSpec": {
    "parts": [
      {"value": "scheme", "variable": true},
      {"value": "://", "variable": false},
      {"value": "address", "variable": true},
      {"value": ":", "variable": false},
      {"value": "port", "variable": true}
    ]
  }
}
```

In our demonstration, the Znodes for each instance contains their IP address and port information in the format `address:port`. For example, the Znode for the instance of `client1` would contain `172.20.0.3:10000`. The method for storing and parsing data from a Znode is entirely dependent on how the end-users' code handles Znodes.

In summary, Zookeeper is a flexible and extensible distributed coordination system that can be used to implement a wide range of distributed system functionality, including service discovery. Although Zookeeper does not have a fixed method for service discovery, it provides a set of primitives and APIs that developers can use to build custom solutions that meet the specific requirements of their systems.

D. Eureka and Springboot

Zookeeper is a open-source distributed coordination service that can be used as a service registry for microservices architectures [24]. However, its node structure can be complex to manage at scale. Znodes in Zookeeper are not standardized in terms of their paths and data, which can make it challenging to develop code for specific use cases. This can lead to inconsistencies in service discovery and management, which can impact the reliability and scalability of microservices architectures.

On the other hand, Eureka provides a standardized method of dumping the service registry, making it easy to manage service registration and discovery [30]. Eureka's REST API is designed to simplify service management by providing a simple and consistent way to interact with the service registry [31]. Table V contains the Eureka REST API calls that standardizes Eureka's key-value pair storage for service discovery. This allows developers to easily query the registry for service information, update service metadata, and remove stale services. Additionally, Eureka provides built-in support for client-side load balancing and fault tolerance, which can further improve the reliability of microservices architectures [32].

The benefits of using Eureka's REST API for service discovery and management are clear. Developers can focus on building microservices applications without worrying about the complexities of managing the service registry. Eureka provides a standardized and simple way to register and discover services, which can improve the scalability and reliability of microservices architectures. Furthermore, Eureka's integration within Spring Boot provides an even easier way to manage service registration and discovery, allowing developers to focus on building logic rather than managing infrastructure [34]. Focusing exclusively on Spring Boot microservice applications makes sense given that Spring is the world's most popular Java framework [35].

Spring Boot's integration with Eureka simplifies service registration and discovery [34]. Developers can easily register services with Eureka using annotations such as `@EnableDiscoveryClient` [35]. This annotation enables automatic service registration with Eureka, eliminating the need for manual registration of instances using HTTP POST requests to `/eureka/v2/apps/appID`. Additionally, Spring Boot provides built-in health checks and load-balancing in the Spring Cloud Netflix stack. Developers can use annotations to enable Hystrix for fault tolerance, such as `@HystrixCommand`, which can be used to ensure that only healthy services are registered with Eureka [37]. Also, developers can use annotations to configure and use Ribbon for client-side load balancing, such as `@LoadBalanced` to create a Ribbon-enabled `RestTemplate` and `@RibbonClient` to customize the Ribbon client [38]. Feign is a client-side HTTP library that can be used to simplify communication between microservices and integrates seamlessly with Eureka, allowing developers to easily call services registered with Eureka and avoiding the need to handle HTTP GET requests to `/eureka/v2/apps/appID` and `/eureka/v2/apps/appID/instanceID` [39].

In conclusion, Eureka's ease of use and standardized calling patterns allows us to perform analysis that would be relevant to any microservice application that uses it, unlike other service discovery platforms such as Zookeeper. This would increase the applicability of our project. Moreover, Eureka integrates well with Spring, the most popular Java framework, which again increases the chance that our project will be able to generate policies for many applications. Hence, Eureka is the best choice of service registry for our project.

VI. Approach

This section describes our approach to automatically generating microsegmentation policies at Layers 3 and 4 using information from the service registry and docker images, without access to the source code. At a high level, we first generate a graph representing the calling relationships between the different microservices through reverse engineering the docker images. Then, we query the service registry for the IPs and ports of each instance of each microservice. Finally, we combine the microservice call graph and the service registry information to generate a graph that represents the relationships between each IP address, and use that to generate a policy based on IP addresses and ports.

A. Binary analysis

First, we extract the docker image using tar and search for jar files. For each jar file, we make use of the CFR Java decompiler [40] to obtain a decompiled version of the microservice source code. Then, we search through the decompiled source code for @FeignClient annotations. When an interface is annotated with @FeignClient, Spring Boot generates a proxy for that interface at runtime, and allows other services to invoke methods on the proxy that then get invoked on the interface, thus allowing other services to remotely interact with the annotated service. This annotation takes 1 argument, which is the name of the service that the request that should be sent to. Hence, if an interface is annotated with @FeignClient, and the argument is Service B, and this calling method is used by Service A, then we can conclude that Service A calls Service B, and hence add this edge to our microservice call graph.

B. Service registry information extraction

Next, we query the registry in order to determine the IPs and ports of all the instances associated with each service. Following the Eureka documentation, we carry out a REST operation and use of a GET query to /eureka/apps in order to obtain all instances of all apps. The response is sent in xml format, which enables us to parse the document and record all the IPs and ports associated with each service.

C. Creation of socket graph

Here, we refer to each pair of IP addresses and their corresponding port number as a socket address. With the service call graph and the list of socket addresses associated with each service, we are able to combine the two to create a graph that shows the dependencies between socket addresses. Our generation of the socket graph follows the pseudocode shown in Figure 1.

The socket graph now contains all the information regarding all instances of all services. If there is an edge between two nodes, then the two instances that the two nodes represent are allowed to communicate. Hence, in order to know which socket addresses an instance should be allowed to communicate with, we simply have to look at all edges from that instance.

We represent the graph using a Networkx digraph. We also created a visualization tool that displays the graph on a simple webpage, to allow the user to clearly see the relationship between various servers. The webpage displays each node, showing its IP address, port, and service name as listed from the service registry, and has edges between nodes that are allowed to communicate.

D. Implementation of policies

The socket graph provides a straightforward means of generating policies. A possible use case would be configuring a firewall to accept connections only from socket addresses that the node is connected to in the socket graph. Unfortunately, Kubernetes Network Policies are currently applied to pods based on their labels, without a way to specify that a specific IP address should only accept network traffic from a predefined list of other addresses. At present, the highest level of granularity that can be specified is that a specific application should only accept network traffic from a list of other socket addresses.

This is logical because Kubernetes, as a container orchestration platform, has abstracted away from IP addresses and port numbers, and expects users to make use of application names or labels to select pods. Kube-dns automatically resolves the addresses for the user. If a network policy were to use IP addresses to select pods, another user deploying the microservice might not be able to use the same network policy because IP addresses are dynamically assigned at startup, which defeats the purpose of container orchestration.

To address this limitation, we developed a script that modifies the label of a pod to include its IP address. This allows us to select pods by socket addresses, using the label name ”pod+IP” as a substitute. Although this seems like a roundabout and cumbersome way to generate network policies, it is necessary because of the limitations of Kubernetes and its current methods of selecting pods to apply policies to. In a real-world scenario, the implementation would likely be more straightforward. For instance, an organization employing firewalls might configure each instance’s firewall to only accept network traffic from neighboring nodes in the socket graph, utilizing tools such as iptables or Windows Firewalls.

VII. Results

We used a demo running on Kubernetes that has four microservices, and makes use of a Eureka service registry [41]. We ran our script which generated and applied Kubernetes network policies to the pods in our eureka-demo namespace, and evaluated our project as described in the sections below.

A. Evaluation via comparison to source code

After we ran our script, which only had access to the Docker images, application.yml and deployment.yml files, and Eureka registry, we compared the results against the ground truth, which we established by inspecting the source code. Through manual inspection, we were able to determine that the four microservices (accounts, products, orders, and main) have a

```
for each edge in service_graph:
let serviceA, serviceB be the vertices that the edge connects
for each socket_address_A under serviceA:
    for each socket_address_B under serviceB:
        socket_graph.add_edge(socket_address_A, socket_address_B)
```

Fig. 1. Generation of socket graph

communication pattern as shown in the diagram below. An additional service, Zuul gateway, acts as the entry point to the application.

TABLE IV
SERVICE INVOCATION RELATIONSHIPS

Service	Ingress	Egress
Main	Gateway	All Besides Gateway
Gateway	None	All Other Services
Accounts	Main and Gateway	Eureka
Orders	Main and Gateway	Eureka
Products	Main and Gateway	Eureka

Lastly, the Eureka service registry needs to be able to accept connections from all addresses, to accommodate new services starting up and attempting to register.

This demo is representative of the microservice architecture because individual services, such as products, orders, and accounts can be independently scaled and/or developed. Hence, there is no need for communication between (for example) products and orders because the main service is responsible for calling other services and putting the information together.

After establishing the ground truth from the source code, we compared it against our graph and policies, and found that it was an exact match. Our visualization tool allowed users to select instances of specific services, and it would gray-out other services which communication should not happen, while keeping the original color of services that were allowed to communicate to the user-selected service.

From Figure 2, we can see that when main is selected, our script has determined that it should have access to connections to orders, accounts, and products. However, when product is selected, our script determined that it should only have access to main, but not orders or accounts, which correlates with the ground truth established from our source code analysis.

In the appendix, we have an example of a network policy generated by our tool. The IPs and ports that are allowed by this policy correspond to main and gateway, which each have 3 instances, for a total of 6 authorized ingress addresses. As expected, the products microservice is not allowed to communicate with orders or accounts.

B. Evaluation via testing of network connections

Upon applying the network policies to the Kubernetes cluster, we proceeded to conduct a series of connectivity tests within the pods using the curl command. Firstly, we verified whether the pod was able to access the IP and port designated by the service graph. Secondly, we switched to a different port that utilized the same IP and assessed whether the connection was severed. Lastly, we evaluated if the connection was dropped when attempting to access an IP belonging to a pod that the current testing pod was not authorized to access.

Unsuccessful attempts to access a socket led to a curl command timeout. To supplement our analysis, we utilized the Cilium Hubble UI, as demonstrated in VII-B, to observe real-time network traffic and any packet drops resulting from our implemented policies. Through this process, we confirm that all three objectives were successfully achieved.

VIII. DISCUSSION

The current study has initiated the process of developing an automated implementation of network policies at layers 3 and 4. Notwithstanding, it is currently in its preliminary phase and has identified several potential limitations that could be enhanced in future iterations, as detailed below:

A. Limited scope of languages

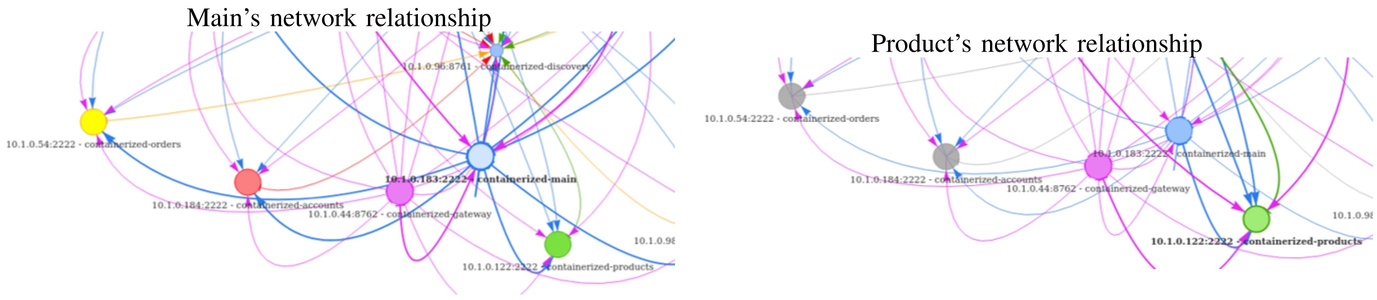
Currently, our project only works on codebases that use Java and the Spring framework. While we selected these because they were the most popular options, this is nonetheless a limitation in that our tool only works on a single language and framework. In order to be developed into a comprehensive tool that microservice developers can use to secure their systems, we would have to expand the scope to cover more languages.

It is worth noting that the overall idea behind our approach is still generalizable to other languages and frameworks, and that only the specific implementation details would be different.

B. Potentially hindered by obfuscation

Our approach relies on decompiling Java bytecode, and hence might be hindered by developers who make use of obfuscation tools and techniques. For example, annotation processing is a technique in which the Java compiler will detect annotations and invoke a processor that replaces the annotation either by generating additional source code or modifying existing code. Hence, the annotation will no longer be directly present in the compiled bytecode. Since our current approach makes use of annotations to create the service call graph, it is possible that our tool will not be able to generate an accurate call graph if techniques such as annotation processing are used.

Fig. 2. Comparing main and product services


Main's network relationship
10.1.0.96:8761 - containerized-discovery
10.1.0.54:2222 - containerized-orders
10.1.0.183:2222 - containerized-main
10.1.0.184:2222 - containerized-accounts
10.1.0.44:8762 - containerized-gateway
10.1.0.122:2222 - containerized-products
Product's network relationship
10.1.0.54:2222 - containerized-orders
10.1.0.183:2222 - containerized-main
10.1.0.184:2222 - containerized-accounts
10.1.0.44:8762 - containerized-gateway
10.1.0.122:2222 - containerized-products

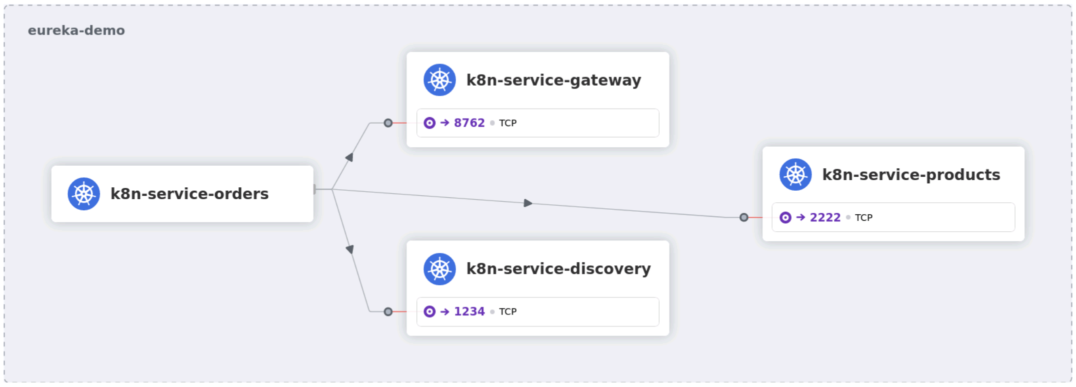


Fig. 3. Cilium Hubble UI Graph for dropped requests initiated from a pod of k8n-service-orders. The red lines indicate that requests were dropped.

C. Reliant on the integrity of service registry

Our current approach relies on the fact that the service registry accurately reflects the socket addresses associated with each service. However, this relies on each instance accurately registering itself under the right service, which is an assumption that might not hold if an attacker compromises a pod. For example, the attacker could theoretically deregister themselves from the service registry, and then re-register as a different microservice, possibly one that is authorized to connect to more services. Future works should consider how this limitation can be overcome - one preliminary idea would be to have an authentication-based system such that the service registry requires a pod to authenticate its identity before it is allowed to register under a specific service. Pods can be initialized with an authentication key corresponding to their service, but not other services, which would prevent an attacker from changing the identity of a pod in the service registry.

IX. Future Directions

A. Enhancing Security of Service Registries

Throughout the course of our study, we gained valuable insights into the crucial role that service registries play in microservice architectures. They facilitate the dynamic scaling of microservices by managing new instances and instance shutdowns, thus ensuring the smooth operation of services. However, this also makes them attractive targets for malicious entities.

As discussed earlier, attackers could potentially exploit service registries by falsely registering compromised machines or virtual machines, leading to undesired communication privileges. Additionally, direct attacks on service registries may have even more severe consequences. For instance, attackers with control over the registry could manipulate the socket graph, generating inaccurate policies and exposing additional vulnerabilities. Therefore, we propose that future research should prioritize the development of strategies to enhance service registry security.

B. Automated policy enforcement

In our study, we successfully generated a graph depicting the dependencies between specific socket addresses. Due to time limitations, our project was restricted to policy generation for a single platform, Kubernetes. We recommend that future research efforts focus on developing a versatile tool capable of supporting diverse platforms, such as Docker Swarm.

Moreover, expanding the scope of this research to include hybrid applications, which operate both in the cloud and on container orchestration platforms, could further enhance the applicability and effectiveness of automated policy enforcement in microservice ecosystems.

Acknowledgment

The authors would like to thank Andong Chen from Zhejiang University and Zhaoxuan Jin from the Southern University of Science and Technology for their collaboration and

support. Their mentorship and contributions to this project have been invaluable.

References

[1] H. Vural, M. Koyuncu, and S. Guney, “A Systematic Literature Review on Microservices,” Computational Science and Its Applications – ICCSA 2017, pp. 203–217, 2017, doi: https://doi.org/10.1007/978-3-319-62407-5_14.

[2] N. Dragoni et al., “Microservices: Yesterday, Today, and Tomorrow,” Present and Ulterior Software Engineering, pp. 195–216, 2017, doi: https://doi.org/10.1007/978-3-319-67425-4_12.

[3] 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, Sep. 2021, doi: https://doi.org/10.1109/msec.2021.3094726.

[4] C. Yoon et al., “Flow Wars: Systemizing the Attack Surface and Defenses in Software-Defined Networks,” IEEE/ACM Transactions on Networking, vol. 25, no. 6, pp. 3514–3530, Dec. 2017, doi: https://doi.org/10.1109/tnet.2017.2748159.

[5] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran and S. Shin, ”BASTION: A security enforcement network stack for container networks”, Proc. USENIX Annu. Techn. Conf. (ATC 2020), pp. 81-95, 2020, https://www.usenix.org/conference/atc20/presentation/nam (accessed Mar. 12, 2023).

[6] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen, Automatic Policy Generation for Inter-Service Access Control of Microservices, in the Proc. of USENIX Security 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/li-xing (accessed Mar. 12, 2023).

[7] Istio. Istio / bookinfo application. 2019, http://bit.ly/3dzSsBv (accessed Mar. 12, 2023).

[8] Google Cloud Platform. Online boutique: Cloud-native microservices demo application, 2019, http://bit.ly/online-boutique (accessed Mar. 12, 2023).

[9] EdwinVW. Pitstop: Garage management application, 2020 https://github.com/EdwinVW/pitstop/ (accessed Mar. 12, 2023).

[10] Weaveworks. Microservices demo: Sock shop, 2017 https://microservicesdemo.github.io/ (accessed Mar. 12, 2023).

[11] Sitewhere. Sitewhere: Open source internet of things platform, 2023 https://sitewhere.io/, 2020 (accessed Mar. 12, 2023).

[12] Manar Alohaly, Hassan Takabi, and Eduardo Blanco. A deep learning approach for extracting attributes of abac policies. In Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies, pages 137–148. ACM, 2018

[13] Alaeddine Saadaoui and Stephen L. Scott. Web services policy generation based on SLA requirements. In 3rd IEEE International Conference on Collaboration and Internet Computing, CIC, pages 146–154, 2017.

[14] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, page 12. ACM, 2012.

[15] Le Yu, Tao Zhang, Xiapu Luo, Lei Xue, and Henry Chang. Toward automatically generating privacy policy for android apps. IEEE Trans. Information Forensics and Security, 12(4):865–880, 2017.

[16] Leila Karimi and James Joshi. An unsupervised learning based approach for mining attribute based access control policies. In 2018 IEEE International Conference on Big Data (Big Data). IEEE, 2018.

[17] Taghrid Samak and Ehab Al-Shaer. Synthetic security policy generation via network traffic clustering. In Proceedings of the 3rd ACM Workshop on Security and Artificial Intelligence, AISec 2010, Chicago, Illinois, USA, October 8, 2010, pages 45–53, 2010.

[18] Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, and Tianwei Sheng. Towards continuous access control validation and forensics. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 113–129, 2019.

[19] Wu Bei, Xingyuan Chen, Yongliang Wang, Dai Xiangdong, and Peng Jun. Network system model-based multi-level policy generation and representation. In International Conference on Computer Science and Software Engineering, CSSE, pages 283–287, 2008.

[20] Ulrich Lang. Openpmf scaas: Authorization as a service for cloud & soa applications. In 2010 IEEE Second International Conference on Cloud Computing Technology and Science. IEEE, 2010.

[21] David S Linthicum. Practical use of microservices in moving workloads to the cloud. IEEE Cloud Computing, 3(5):6–9, 2016

[22] S. Curtis, “Using Network Policy with Istio,” Istio, Aug. 10, 2017. https://istio.io/latest/blog/2017/0.1-using-network-policy/ (accessed Mar. 14, 2023).

[23] J. Sava, “Microservice developers programming languages 2022,” Statista, Mar. 03, 2023. https://www.statista.com/statistics/1273806/microservice-developers-programming-language/ (accessed Mar. 14, 2023).

[24] Apache Software Foundation, “ZooKeeper: Because Coordinating Distributed Systems is a Zoo,” zookeeper.apache.org, Jan. 30, 2023. https://zookeeper.apache.org/doc/current/zookeeperOver.html

[25] Apache Software Foundation, “ZooKeeper: Because Coordinating Distributed Systems is a Zoo,” zookeeper.apache.org, Jan. 30, 2023. https://zookeeper.apache.org/doc/current/zookeeperUseCases.html

[26] S. G. Toral, “spring-cloud-zookeeper-service-discovery-demo,” GitHub, Mar. 14, 2023. https://github.com/santteegt/spring-cloud-zookeeper-service-discovery-demo (accessed Mar. 14, 2023).

[27] C. Kotwasinski, “ApacheZookeeperServiceDiscoveryExample,” GitHub, Feb. 06, 2023. https://github.com/conorKotwasinski/ApacheZookeeperServiceDiscovery Example (accessed Mar. 14, 2023).

[28] M. Bansal, B. Yang, M. Bhosale, and K. Jiang, “Uber’s Highly Scalable and Distributed Shuffle as a Service,” Uber Blog, Jul. 07, 2022. https://www.uber.com/blog/ubers-highly-scalable-and-distributed-shuffle-as-a-service/ (accessed Mar. 14, 2023).

[29] M. Han, “ZooKeeper at Twitter,” blog.twitter.com, Oct. 11, 2018. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2018/zoo keeper-at-twitter (accessed Mar. 14, 2023).

[30] Spring Cloud, “spring-cloud/spring-cloud-netflix,” GitHub, Mar. 13, 2023. https://github.com/spring-cloud/spring-cloud-netflix (accessed Mar. 14, 2023).

[31] Netflix, “Eureka REST operations,” GitHub, Nov. 24, 2021. https://github.com/Netflix/eureka/wiki/Eureka-REST-operations (accessed Mar. 14, 2023).

[32] Spring Cloud, “Spring Cloud Netflix,” cloud.spring.io. https://cloud.spring.io/spring-cloud-netflix/reference/html/ (accessed Mar. 14, 2023).

[33] Kubernetes Network Policies, kubernetes.io. https://kubernetes.io/docs/concepts/services-networking/network-policies/ (accessed Mar. 14, 2023).

[34] Spring, “Spring Projects,” Spring.io, 2019. https://spring.io/projects/spring-boot

[35] S. Maple and A. Binstock, “JVM Ecosystem report 2018 - About your Platform and Application — Snyk,” snyk.io, Oct. 17, 2018. https://snyk.io/blog/jvm-ecosystem-report-2018-platform-application/

[36] Spring Cloud, “2. Spring Cloud Commons: Common Abstractions,” cloud.spring.io. https://cloud.spring.io/spring-cloud-commons/multi/multi__spring_cloud_commons_common_abstractions .html (accessed Mar. 14, 2023).

[37] Spring Cloud, “3. Circuit Breaker: Hystrix Clients,” cloud.spring.io. https://cloud.spring.io/spring-cloud-netflix/multi/multi__circuit_breaker_hystrix_clients.html (accessed Mar. 14, 2023).

[38] Spring Cloud, “6. Client Side Load Balancer: Ribbon,” cloud.spring.io. https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-ribbon.html (accessed Mar. 14, 2023).

[39] Spring, “Spring Cloud OpenFeign,” docs.spring.io. https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/ (accessed Mar. 14, 2023).

[40] L. Benfield, “CFR - yet another java decompiler.,” www.benf.org. https://www.benf.org/other/cfr/ (accessed Mar. 14, 2023).

[41] T. Diler, “SPRING BOOT MICROCSERVICE USING SPRING CLOUD, EUREKA, RIBBON, ZUUL, ZIPKIN, SLEUTH,” GitHub, Mar. 08, 2023. https://github.com/tanerdiler/spring-boot-microservice-eureka-zuul-docker-gateway-kubernetes (accessed Mar. 15, 2023).

Appendix

TABLE V
Eureka REST Operations

Operation	**HTTP Action**	**Description**
Register new application instance	POST /eureka/v2/apps/appID	Input: JSON/XML payload HTTP Code: 204 on success
De-register application instance	DELETE /eureka/v2/apps/appID/instanceID	HTTP Code: 200 on success
Send application instance heartbeat	PUT /eureka/v2/apps/appID/instanceID	HTTP Code: * 200 on success * 404 if instanceID doesn't exist .
Query for all instances	GET /eureka/v2/apps	HTTP Code: 200 on success Output: JSON/XML
Query for all appID instances	GET /eureka/v2/apps/appID	HTTP Code: 200 on success Output: JSON/XML
Query for a specific appID/instanceID	GET /eureka/v2/apps/appID/instanceID	HTTP Code: 200 on success Output: JSON/XML
Query for a specific instanceID	GET /eureka/v2/instances/instanceID	HTTP Code: 200 on success Output: JSON/XML
Take instance out of service	PUT /eureka/v2/apps/appID/instanceID/status?value=OUT_OF_SERVICE	HTTP Code: * 200 on success * 500 on failure
Move instance back into service (remove override)	DELETE /eureka/v2/apps/appID/instanceID/status?value=UP	HTTP Code: * 200 on success * 500 on failure
Update metadata	PUT /eureka/v2/apps/appID/instanceID/metadata?key=value	HTTP Code: * 200 on success * 500 on failure
Query for all instances under a particular vip address	GET /eureka/v2/vips/vipAddress	* HTTP Code: 200 on success Output: JSON/XML * 404 if the vipAddress does not exist.
Query for all instances under a particular secure vip address	GET /eureka/v2/svips/svipAddress	* HTTP Code: 200 on success Output: JSON/XML * 404 if the svipAddress does not exist.

```
apiVersion networking.k8s.io/v1
kind NetworkPolicy
metadata
 name 10.1.0.2312222containerizedproductspolicy
 namespace eurekademo
spec
 egress
 ports
  port 8761
  to
  podSelector
    matchLabels
     ip 10.1.0.96
 ports
  port 53
   protocol UDP
  to
  namespaceSelector
    matchLabels
     name kubesystem
   podSelector
    matchLabels
     k8sapp kubedns
 to
  ipBlock
    cidr 10.152.183.0/24
 ingress
 from
  podSelector
    matchLabels
     ip 10.1.0.222
 from
  podSelector
    matchLabels
     ip 10.1.0.69
 from
  podSelector
    matchLabels
     ip 10.1.0.183
 from
  podSelector
    matchLabels
     ip 10.1.0.166
 from
  podSelector
    matchLabels
     ip 10.1.0.246
 from
  podSelector
    matchLabels
     ip 10.1.0.44
 podSelector
  matchLabels
   ip 10.1.0.231
 policyTypes
 Ingress
 Egress
```

Fig. 4. An example of generated Kubernetes Network Policy