Automatic Policy Generation for Inter-Service Access Control of Microservices

Xing Li, *Zhejiang University;* Yan Chen, *Northwestern University;* Zhiqiang Lin, *The Ohio State University;* Xiao Wang and Jim Hao Chen, *Northwestern University*

https://www.usenix.org/conference/usenixsecurity21/presentation/li-xing

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

Automatic Policy Generation for Inter-Service Access Control of Microservices

Xing Li* *Zhejiang University*
Yan Chen *Northwestern University*
Zhiqiang Lin *The Ohio State University*
Xiao Wang *Northwestern University*
Jim Hao Chen *Northwestern University*

Abstract

Cloud applications today are often composed of many microservices. To prevent a microservice from being abused by other (compromised) microservices, inter-service access control is applied. However, the complexity of fine-grained access control policies, along with the large-scale and dynamic nature of microservices, makes the current manual configuration-based access control unsuitable. This paper presents AUTOARMOR, the first attempt to automate inter-service access control policy generation for microservices, with two fundamental techniques: (1) a static analysis-based *request extraction* mechanism that automatically obtains the invocation logic among microservices, and (2) a graph-based *policy management* mechanism that generates corresponding access control policies with on-demand policy update. Our evaluation on popular microservice applications shows that AUTOARMOR is able to generate fine-grained inter-service access control policies and update them timely based on changes in the application, with only a minor runtime overhead. By seamlessly integrating with the lifecycle of microservices, it does not require any changes to existing code and infrastructures.

1 Introduction

As an emerging software architecture, microservices have been widely used in modern cloud applications [27]. In this architecture, a large, complex application is split into multiple *microservices* according to its business boundaries. Each of them can be independently developed, deployed, and upgraded, thereby significantly improving the flexibility of software development and maintenance. However, communications among microservices are exposed through the network, which creates a potential attack surface. In particular, an adversary may attack the entire application through a compromised microservice by sending malicious requests to other microservices. Therefore, to defend against this kind of attacks, popular microservice infrastructures such as *Kubernetes* [42] and *Istio* [22] provide inter-service access control mechanisms to specify what resources a microservice can access.

However, to achieve high control flexibility, these mechanisms often employ complex policies for fine-grained authorization. Currently, these policies still rely on manual configurations from administrators, which are time-consuming and error-prone. Given the sheer scale of modern microservice applications [30], it is impractical to manually configure and maintain access control policies for thousands of microservices. Even worse, their frequent iterations require the policies to be updated accordingly in time, which can also be an "impossible mission" for manual configuration. Hence, to make a robust access control mechanism function well, automatic policy generation is essential.

Nevertheless, in distributed systems, the automatic generation of security policies is not new. Over the past few years, significant efforts have been made to achieve this goal. These studies fall into three categories based on how they acquire business logic or security intent. (1) The first is document-based approaches [3,34,51,55], which utilize natural language processing (NLP) to infer security policies from application documents. Although documents can properly reflect developers' high-level intentions, it is not trivial to extract them accurately. For example, Text2Policy [51] achieved an average recall of 89.4%. (2) The second is history-based approaches [23,35,50] that mine security policies from historical operations. However, it relies heavily on the quality of training data, which means only sufficient historical data can lead to complete security policies. For instance, P-DIFF [50] infers access control policies by monitoring access logs, but its average precision is only 89%. (3) The last category is model-based approaches [7,25], which formally model software behavior and generates security policies accordingly. Unfortunately, it is hardly agile and scalable to build and update the system model manually when accommodating frequently iterated and large-scale microservice applications.

As such, automatic policy generation for inter-service access control in microservices is still an open problem, which cannot be achieved by merely adopting the existing

*The work was performed when the first author was visiting Northwestern University.

approaches of security policy generation. To advance the state-of-the-art, we present AUTOARMOR, a practical policy generator that can automatically generate fine-grained inter-service access control policies and keep them updated over time with the application's evolution. There are two fundamental challenges when building AUTOARMOR: (1) how to obtain complete and fine-grained invocation logic, and (2) how to generate and update access control policies.

To solve the first challenge, we propose a static analysis-based *request extraction* mechanism, through which all possible invocations a microservice may initiate are extracted from its source code. Such extraction employs the usage and semantic models of inter-service communication libraries to identify the requests. After that, the detailed attributes of the invocations are collected for fine-grained access control.

To address the second challenge, we design a novel graph-based *policy management* mechanism, which considers the unique characteristics of microservices and takes over the generation, update, and removal of access control policies through a permission graph. Respecting the application evolution, AUTOARMOR is designed to be integrated into the lifecycle of microservices and does not require any modification to the current application code and infrastructures.

Contributions. Our paper makes the following contributions:

- We present AUTOARMOR, the first automatic policy generation tool for the inter-service access control of microservices, which improves the availability of current service-level authorization.
- We develop a static analysis-based *request extraction* mechanism (§4), which uses program slicing and semantic analysis to extract the inter-service invocation logic with details.
- We design a graph-based *policy management* mechanism (§5), which translates the invocation logic to fine-grained access control policies and continues to update them with the evolution of the application.
- We implement AUTOARMOR for *Kubernetes* and *Istio*, the two most widely applied microservice infrastructures, and evaluate it with 5 popular microservice applications (§6). The results show that AUTOARMOR is sound and practical in handling the policy generation for inter-service access control of microservices.

2 Background and Motivation

2.1 Microservice Architecture

The architecture of cloud applications is constantly evolving. Traditional *monolithic* software (Figure 1 (a)) is packaged and deployed as a whole containing all modules. It is efficient when the application is relatively simple. Nevertheless, growth in complexity destroys its flexibility and makes the system clumsy: modifying a single module requires retesting, repackaging, and redeploying the whole application; the accurate scaling of system bottlenecks is also unachievable.

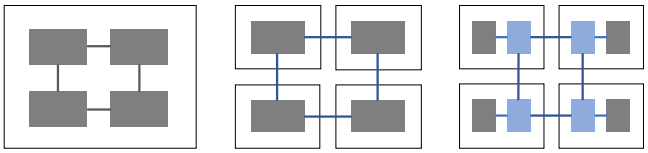

Figure 1: The evolution of cloud application architecture.

Aiming to elegantly and flexibly develop and maintain complex applications, the *microservice* architecture (Figure 1 (b)) emerged. It splits an application into several microservices[1] running on different machines (or VMs and containers). Each microservice can be independently developed, deployed, upgraded, and scaled. Through lightweight network API invocations, multiple services can be combined as *service chains* to achieve complicated functionalities. Microservices significantly improve the agility of cloud applications.

As a side effect, the invocation relationships among services become cumbersome as the service quantity grows. To solve this problem, *service mesh* takes the stage and enhances the microservice architecture as a dedicated communication infrastructure layer. It uses proxies (blue boxes in Figure 1 (c)) to manage all network traffics among microservices and transparently add features like access control, traffic management, and monitoring to inter-service communications.

To prevent service interruptions, microservices adopt a progressive upgrade method in practice. Administrators first deploy the new version of a service (also called *canary deployment*) and steer a small amount of business traffic into it for evaluation. After confirming that the new version works appropriately, all traffic will be gradually migrated to it, and the old version will then be offline. Such an update strategy is a crucial property that we need to deal with and utilize.

Currently, microservices are widely applied [27]. A survey conducted in 2018 [13] showed that 74% of respondent companies are using microservices. Based on a survey from Cloud Native Computing Foundation (CNCF) [6], *Kubernetes* [42] is the leading infrastructure, accounting for 83% of the market. As for service mesh, *Istio* [22] is generally recognized as the most popular implementation [9]. In this paper, we use them as the foundation infrastructures and work towards securing microservice applications built atop of them.

2.2 Motivation

As a cloud application architecture, microservices should give security a high priority. In this architecture, communications that were previously conducted within a monolithic application by local invocations are now exposed through the

[1] In this paper, we use *microservice* and *service* interchangeably and regard an *application* as a collection of microservices.

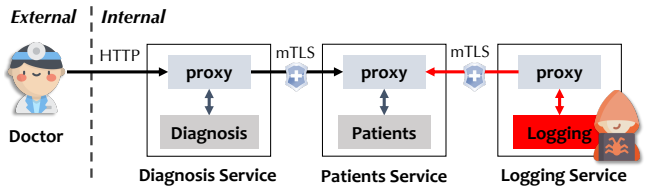

Figure 2: The architecture of a medical application. An attack is initiated from a compromised logging service.

network, which creates a potential attack surface. Although network isolation enhances security to some extent, the communication channels still need to be protected. Two methods are currently adopted to secure the inter-service communication, one is encryption, such as SSL/TLS, and the other is inter-service access control.

Inter-Service Access Control. According to a CNCF survey in 2018 [6], 73% of microservices are deployed in *containers*. However, third-party libraries may introduce exploitable vulnerabilities to containers. In particular, Tak *et al.* [41] found that more than 92% of the container images contain unpatched software vulnerabilities. Therefore, attackers can compromise a microservice by breaking into the corresponding container. Hiding behind the IP address and certificate of a compromised service, they can send malicious requests to other microservices to initiate attacks or steal data. As microservices work together to accomplish complex functionalities, their natural mutual trust makes the entire application vulnerable to a single compromised service.

Considering Figure 2 as an example, there are 3 microservices in a medical application: a *diagnosis* service, a *patients* service, and a *logging* service. Doctors can access the information stored in the *patients* service through the *diagnosis* service. However, a compromised *logging* service may directly talk to the *patients* service to obtain sensitive patient information, even with mutual-TLS enabled.

This is where inter-service access control comes in. By specifying services' permissions (i.e., the resources they can access), it can regulate the behavior of microservices and prevent such attacks. In our example, the administrator can specify that only the *diagnosis* service can access the *patients* service to defend against the attack described above.

Policy Generation Gap. Currently, popular microservice infrastructures are equipped with policy-based inter-service access control mechanisms. A policy is a list of legitimate requests, that is, a whitelist. After being issued by the administrator, policies are installed in the proxies corresponding to the related services. At runtime, proxies verify each incoming request based on the installed policies and return the authorization result for policy enforcement.

The access control policy is designed sophisticated for flexible authorization. As shown in Figure 3, there are three key fields in a policy: **`from`** (specifies the source of the request), **`to`** (specifies allowed operations), and **`when`** (specifies the

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: diagnosis-v1-to-patients
  namespace: default
spec:
  selector:
    matchLabels:
      app: patients
  rules:
  - from:
    - source:
        principals:["cluster.local/ns/default/sa/diagnosis"]
    to:
    - operation:
        paths: ["/patients/*"]
        methods: ["GET"]
    when:
    - key: request.headers[version]
      values: ["v1"]
```

Figure 3: An inter-service access control policy in Istio.

conditions). This policy allows the `v1` version of *diagnosis* service to access the resources under the "`/patients/`" path of *patients* service with `GET` method. By this means, *Istio* achieves fine-grained access control at the workload level.

While these mechanisms seem powerful and promising, currently they still rely on careful manual configuration, which is error-prone and inflexible. Moreover, manual configuration may be unrealistic when facing large-scale microservice applications (e.g., Twitter has $O(10^3)$ different microservices and $O(10^5)$ service instances[2] in 2016 [30]). Further, service upgrades may lead to changes in the invocation logic, and consequently affect related access control policies. Frequent iteration of microservices requires frequent policy updates, which is also challenging for manual configuration.

As such, there is a vast "**policy generation gap**" between currently adopted mechanisms and the requirements of the dynamic, large-scale microservice applications, which severely prevents them from being fully utilized in practice. This motivates us to design a tool to bridge this gap and help these mechanisms realize their full potential.

3 Overview

3.1 Threat Model, Scope, and Assumptions

Threat model. To sum up, in this paper, we consider that attackers can compromise running microservices by exploiting vulnerabilities in their containers. These attackers are capable of perceiving other services in the network as well as their exposed APIs, and also initiating arbitrary requests from the subverted services. In this manner, the adversary can perform illegal access to other microservices.

Scope. Inter-service access control mechanisms are designed to resist these attacks. Instead of developing new mechanisms, we aim to prompt microservice security by bridging the policy generation gap with an automated approach. Specifically, our goal is to automatically generate least privileged access

[2] In production, administrators use multiple identical instances of a microservice to improve performance and provide high availability.

control policies for services and to keep them up to date as the application evolves. Since this paper focuses on inter-service access control, security policies to resist attacks from end-users or against hosts or platforms are beyond our scope.

Assumptions. Our design is based on two reasonable assumptions. First, although microservices could be compromised, their source code can be benign. In other words, the programmers do not take the initiative to be malicious. Second, we assume that the source code of microservices can be obtained, which is common in practices. In §7, we discuss scenarios where the source code is unavailable or cannot be trusted.

3.2 Challenges

There are enormous challenges in designing a practical automatic policy generator for inter-service access control. We organize them into five categories and briefly describe the corresponding countermeasures based on some key insights derived from existing efforts and observations.

C1: Finding the suitable target that reflects the normal behavior of microservices. The first step in policy generation is to define normal system behavior. Many sources imply such information. Current works focus on documents, system logs, and monitoring data. The target we choose must completely and accurately reflect the expected behavior of services, which sets the approach's upper bound.

Microservices' code is the direct source of its behavior. This correspondence makes it a better representation than documents in terms of accuracy. Compared with logs, its acquisition does not require complete tests or pre-runs. Therefore, when a microservice initiates a request that is **inconsistent** with its code to other services or external networks, we regard the request as an intent violation. That is, the service is compromised, and the request is malicious. Based on this insight, we aspire to extract inter-service communication logic from the service code.

C2: Handling the inherent limitations of static analysis. It is challenging to extract the invocation logic from services' code. Microservices can communicate in various ways with different implementations. Performing comprehensive static analysis with brute force could be extremely heavy, which may cause state explosions and make the approach unfeasible. Thus, we need to find a way to reduce the search space while ensuring complete and sound results.

To this end, we can consider only the code related to network invocations. Compromised microservices affect other services through network invocations. This means that instead of the entire service code, we can focus on the program slices related to inter-service communications to narrow the analysis space. Besides, to understand the specific communication methods between microservices and obtain relevant insights, we select five popular open-source microservice applications and observe the protocols and libraries used for inter-service invocations. As shown in Table 1, the amounts of involved protocols and libraries are limited, and the invocation manner is relatively uniform, especially in the same application: for each protocol, more than half of the requests are initiated using the same library. These characteristics make it feasible to model them for accurate identification, thereby providing an opportunity to extract the inter-service invocation logic efficiently with static analysis.

Application	Protocol	# of Used Libraries	# of Initiated Requsets	Share of the Main Library
Bookinfo	HTTP	3	5	60%
	TCP	2	2	50%
Online Boutique	gRPC	2	20	95%
	TCP	1	7	100%
Sock Shop	HTTP	2	39	85%
	TCP	3	47	51%
Pitstop	HTTP	2	48	69%
	TCP	4	73	52%
Sitewhere	HTTP	1	4	100%
	gRPC	1	270	100%
	TCP	5	240	61%

Table 1: The inter-service communication protocols and libraries used in our collected microservice applications. The last column represents the proportion of requests initiated by the library that initiated the most requests to the total requests.

C3: Analyzing microservices implemented in various programming languages. Due to the loose coupling of microservices, developers can choose the most appropriate language to implement each service. As a result, an application may include services developed in multiple languages. To obtain invocations throughout the entire application, it is inevitable to support every used programming language. This is tricky since they have different syntax, libraries, and tools.

Although programming languages have different features, they can be divided into *statically typed languages* and *dynamically typed languages*. The static analysis processes are similar for languages in the same category. Besides, as aforementioned, the modeling task is relatively tractable, and these languages have powerful static analysis tools available, which is of great benefit to our work.

C4: Mining detailed attributes of inter-service invocations. Fine-grained access control relies on detailed attributes, but extracting them is not straightforward. First, the attributes are diverse since the parameters in each invocation method are distinct. We need to identify the useful ones. Second, variables may be modified multiple times from declaration to use, making it challenging to obtain accurate parameters. For example, a *URL* may be generated from a combination of numerous strings; having only part of them may be meaningless.

Although many parameters are involved in inter-service communications, we do not need to extract them all. The expressiveness of the policy language determines the expected level of granularity for attributes. That is, it defines which parameters are needed. Therefore, we only extract the attributes that can be used for access control, such as *URL* and *method*

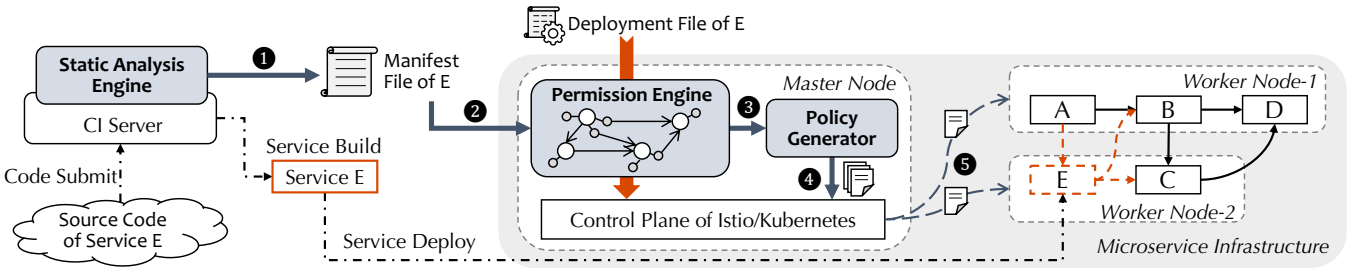

Figure 4: The architecture of AUTOARMOR and the deployment of service E.

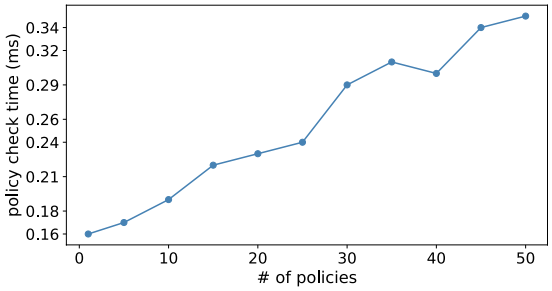

Figure 5: Average policy checking time with different number of policies installed for *details* service of *Bookinfo* app [20].

of HTTP requests. Additionally, for accuracy, we track the definitions and usage of key parameters, and construct the final attributes with string analysis.

C5: Managing access control policies. Tailoring a policy management mechanism for microservices needs to consider a series of unique characteristics. For example, their progressive upgrade method makes it common to have multiple versions of a particular service present simultaneously, and these versions may raise different inter-service dependencies. Thus it is necessary to distinguish them in the authorization. Besides, microservices are frequently released and iterated. Hence, prompt policy generation and update are indispensable, especially in large-scale scenarios.

Moreover, since network communication is slower than local invocations, performance has become a notable limitation of microservices. Policy enforcement for each request will introduce extra latency at runtime, which could further slow down large-scale applications that need to process thousands of requests per second. With this in mind, we also need to produce a policy set with an optimal runtime performance.

Inter-service access control policies are generated based on the dependencies among services, which can be naturally represented by a graph. Besides, as aforementioned, proxies match incoming requests with installed policies one by one at runtime, which implies that the number of policies may affect policy enforcement. We measured this impact in Figure 5. It shows that the policy checking time increases linearly with the number of installed policies. Therefore, redundant access control policies will degrade the entire application, and we can reduce the runtime overhead by generating an optimal policy set with minimized redundancy.

3.3 AUTOARMOR Overview

From the insights above, we have created AUTOARMOR, an inter-service access control policy generator that acquires the invocation logic among microservices and translates it to corresponding access control policies. AUTOARMOR's workflow contains two separate phases: the *request extraction* phase, and the *policy management* phase.

In the *request extraction* phase, AUTOARMOR uses a *static analysis based request extraction* to obtain the requests a microservice may initiate. It first identifies the statements that initiate network API invocations, and then uses them as starting points to perform backward taint propagation on the control flow graphs to get the program slices associated with each invocation. Finally, it extracts the relevant attributes, such as *URL* and *method*, from the slices via semantic analysis.

In the *policy management* phase, we design a *graph-based policy management* to generate policies according to the inter-service invocation logic extracted in the first phase. With the permission graph, it achieves on-demand and incremental policy updates, and minimizes redundant policies by aggregating the same permissions of different service versions.

Architecture. As shown in Figure 4, AUTOARMOR consists of an offline *Static Analysis Engine* responsible for request extraction, an online *Permission Engine* for maintaining and updating the permission graph, and an online *Policy Generator* for translating the graph into access control policies.

AUTOARMOR aims to integrate with the microservice lifecycle and infrastructures seamlessly. The lifecycle involves *continuous integration / continuous delivery* (CI/CD), an automatic workflow for service development and deployment. To naturally access the code of services, the *Static Analysis Engine* is placed on the CI server, where a series of tools run for code checking and automatic testing. Besides, located on the *master node*, the *Permission Engine* receives and parses the commands from the administrator, and then passes them to the control plane of the infrastructure. In this manner, AUTOARMOR can sense all changes of microservices.

Workflow. The dotted black line in Figure 4 depicts the standard CI/CD pipeline of service E. At first, its source code is submitted to the CI server. The *Static Analysis Engine* analyzes its code and generates a manifest file to describe the invocations that it may initiate (❶). Subsequently, service

Languages	Java	Python	Go	JavaScript	Ruby	C#
Source Code	✓	✓	✓	✓	✓	✓
Bytecode	✓	✓	✗	✗	✓	✓
Binary Code	✗	✗	✓	✗	✗	✗

Table 2: The code forms of some programming languages used in microservices.

E is built and ready for deployment. At deployment time, the *Permissions Engine* parses the manifest file to generate a permission node for *E* and inserts it into the application's permission graph (❷). With the change of the graph, the *Policy Generator* calculates the access control policies that need to be added or modified (❸). Afterward, it issues the policies to the control plane of the microservice infrastructure (❹), which further distributes them to services' proxies for subsequent policy enforcement (❺).

4 Static Analysis-Based Request Extraction

4.1 The Input of Static Analysis

A service program may have multiple code forms, which is determined by the execution process of its programming language. Some languages' source files will be interpreted and executed by the corresponding interpreter (e.g., *JavaScript*), some will be compiled into binaries (e.g., *Go*), and some will be compiled into bytecode as an intermediate form (e.g., *Java*). Table 2 lists the code forms associated with the programming languages used in our collected microservices. As we can see, their code forms are not the same, and the most suitable forms for static analysis are also diverse. Nevertheless, we use source code as the input of static analysis to illustrate the proposed method for facilitating understanding.

Apart from the source code, some external files also carry critical information related to service invocations, and we also treat them as the analysis input:

(1) The `.proto` files contain all gRPC API definitions, which can guide us to identify gRPC invocations in the service code;

(2) The deployment file contains the environment variables of the service container, which may indicate some custom fields in the invocation URLs, such as *hostname* and *port*. It also declares the metadata of the service, such as *service name* and *service version*.

Generally, only the above two external files need to be considered. Nonetheless, if there are other configuration files containing invocation-related information, it is also necessary to model and parse them for analysis.

4.2 Request Extraction

With the above input of a microservice, we aim to extract all network API invocations it may initiate and generate a manifest file to describe them.

Languages	Libraries	Methods
Java (23 svcs)	javax.ws.rs.client	SyncInvoker: get(), post(), put(), delete(), head() ,method() Invocation: invoke()
	org.springframework.web.client	RestTemplate: execute(), exchange(), getForObject(), getForEntity(), postForObject(), postForEntity(), delete(), put()
	org.apache.solr.client.solrj	SolrClient: query(), request(), ping()
JavaScript (4 svcs)	request	request(), request.get(), request.post(), request.put(), request.del()

Table 3: The modeled HTTP request libraries and methods for *Java* and *JavaScript* services in our evaluation applications.

There are several ways to extract requests statically from the source code. Simple pattern matching methods (e.g., regular expression matching) are fast, but almost inaccurate since the key variables may be defined elsewhere and modified multiple times. Tracking the definitions and usages of all variables overcomes this drawback, but it may result in state explosions or being extremely slow.

Static taint analysis is commonly utilized to track and analyze untrusted information flows for vulnerability mining. Nevertheless, recent work [29] showed that it could be applied to obtain the program slice related to a specific instruction, which contains all associated data flows. Inspired by this, we employ it to perform program slicing from statements that make network API invocations, and then extract attributes from these slices. Further, since we only focus on the attributes that can be used for access control, we involve semantic models for *semantics-aided program slicing*. That is, identifying and tainting only the key parameters in the API invocation statements to obtain the smallest but sufficient program slices. In this manner, we can reduce the state space and acculturate the analysis while ensuring the accuracy.

At a high level, our approach is designed to be general for different programming languages and invocation protocols, even if the detailed implementations are not the same. Specifically, it consists of the following steps.

Step-I: Identifying inter-service invocation statements. Our static analysis is performed on the *control flow graphs* (CFGs) describing the intra-procedural program execution and the *call graphs* (CGs) describing the inter-procedural call relationships. Therefore, we first scan the source files of the target service, and construct CFGs, CGs, and constant/variable tables accordingly to prepare for the following analysis. There are many mature tools that can assist this process for different programming languages.

Usage Models. Next, we build *usage models* for inter-service communication libraries to identify the statements that initiate network requests. As demonstrated in Table 1, the inter-service invocations in the same application are relatively uniform, and the number of libraries involved is also limited. Therefore, we can carry out targeted modeling to reduce

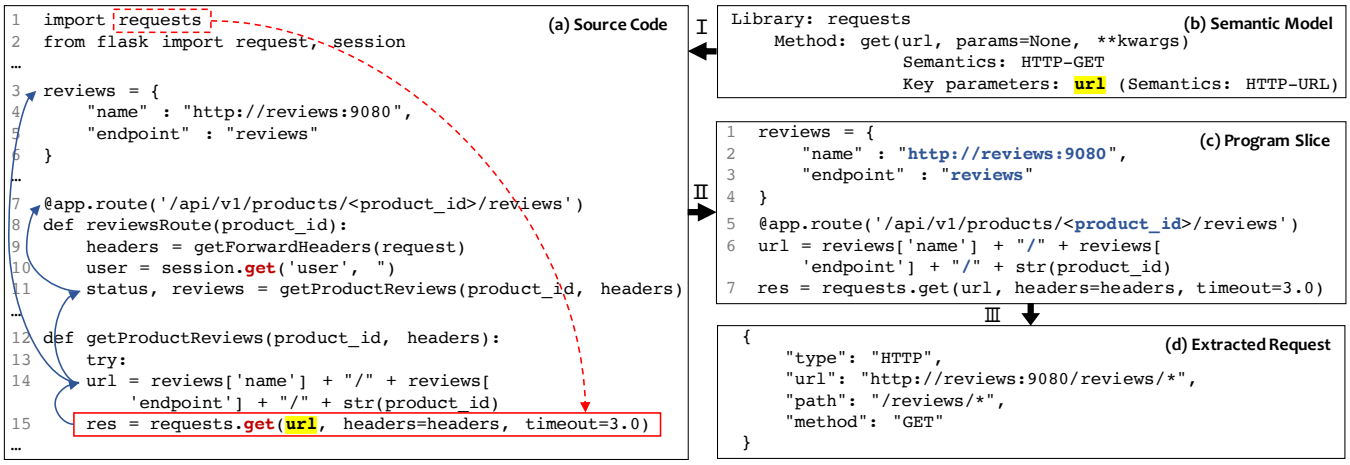

Figure 6: Request extraction for the *productpage* service of *Bookinfo* app [20]: I. using semantic (usage) models (b) to identify the network API invocation statement (line-15 in (a)), II. performing backward taint propagation to obtain the corresponding program slice (c), and III. extracting the request with its attributes from the slice (d).

```
import javax.ws.rs.client.*;
import javax.ws.rs.core.MediaType;
…
ClientBuilder cb = ClientBuilder.newBuilder();
…
Client client = cb.build();
WebTarget ratingsTarget = client.target(
                    ratings_service + "/" + productId);
Invocation.Builder builder = ratingsTarget.request(
                            MediaType.APPLICATION_JSON);
…
Response r = builder.get();
…
```

Method Signature:
`javax.ws.rs.client.SyncInvoker#get()Ljavax/ws/rs/core/Response;`

Figure 7: Java example of a unique method identifier in statically typed programming languages. (This code snippet is from the *review* service of *Bookinfo* app [20].)

the workload. Table 3 lists the HTTP request libraries and methods that we model for services developed in *Java* and *JavaScript* in the evaluation applications. For gRPC requests, we can parse the API definitions in the `.proto` file to figure out which statements can initiate inter-service invocations. Note that the sufficiency of usage models directly affects the completeness of request extraction and subsequent access control, which will be discussed further in §7.

In *statically typed programming languages*, such as *Java* and *Go*, a method can be uniquely identified due to its binding to the corresponding *type*. Hence, as shown in Figure 7, we only need to model the methods that ultimately initiate the requests, and then locate their uses in the code. Nevertheless, in *dynamically typed programming languages*, such as *Python* and other scripting languages, it is not easy to determine whether the method invoked in a statement is the method of our interest. The *types* of methods are checked at runtime, which makes it impossible to uniquely distinguish a method in static analysis. For example, in the code snippet in Figure 6 (a), the two `get` methods at line 10 and line 15 have completely different functionalities. Therefore, for these languages, we start from the "`import`" statements and model each step in the request initiation processes. As shown by the dotted red arrow in Figure 6 (a), we can identify the statements that actually initiate network API invocations by scanning these steps from top to bottom along the syntax tree.

Step-II: Performing semantics-aided program slicing. In this step, we first augment the usage models with semantic information to build *semantic models*. As shown in Figure 6 (b), these semantics indicate the key parameters we need to focus on during subsequent program slicing.

Program slicing. Starting from the invocation statements collected in Step-I, we taint the key parameters and perform *backward taint propagation* along their data flows to mark all relevant variables (the blue arrows in Figure 6 (a)). Namely, (1) a variable on the left-hand side of an assignment statement will taint the variables used on the right-hand side, (2) a tainted method argument will propagate the taint to the corresponding arguments in the statements that call this method, and (3) if a method's return value is tainted, the taint propagation also needs to be performed on its method body from the return statements. This process will iterate until the current variable comes from an incoming request or is assigned by a constant; that is, no more variables to propagate. In this way, we get a streamlined program slice associated with a request, which contains minimal but sufficient information for access control (Figure 6 (c)).

In the process of taint propagation, we may encounter *branches* in the reverse control flow. Generally, there are only two types of branches that will affect our program slicing: (1) there is a *conditional statement* in the code, which has multiple branches containing tainted variables, and (2) the arguments of a method are tainted, and the method has multiple callers. These *branches* indicate that various invocations may be initiated through an identical statement. Therefore, to

deal with the situations, we duplicate the program slice and perform taint analysis on each path separately.

Step-III: Extracting the details of invocations. From the program slices, we need to further extract useful attributes (Figure 6 (d)). The *path* attribute of gRPC requests are explicitly defined in the `.proto` files, and some attributes (e.g., *method* of HTTP requests) are often reflected in semantic models (e.g., Figure 6 (a)). Nevertheless, many attributes still need to be obtained from the program slices, such as *hostname*, *port* of TCP requests, and *path* of HTTP requests. These attributes are usually included in URLs and may involve a series of string processing operations. This means that they may be modified multiple times after their definitions, and we need to reconstruct them for accurate access control.

String reconstruction. Contrary to program slicing, we start from the end of taint propagation and reconstruct variables in the program slice along the forward direction of data flows. To this end, we model a series of basic string processing methods, such as '`+`' and '`append()`'. Since the program slice has been duplicated for branches in Step-II, we do not need to handle them during the reconstruction. Besides, as described in §4.1, some environment variables defined in the deployment file may be tainted due to their participation in URL generation. Thus we also employ them for string reconstruction.

The finally constructed URLs may include some fields from incoming requests, which can not be determined by static analysis (e.g., the `product_id` in Figure 6 (c)). We use wildcards to fill these undetermined fields. Note that even if prefix and suffix matching is common in access control, this uncertainty could lead to over-authorization. We will discuss this further in §6.3.

After these three steps, we generate a JSON-based manifest file to describe the requests that a microservice might initiate. The syntax of the manifest file is displayed as follows:

```
'service': service name
'version': service version
'request': [{'type'   : 'http' | 'grpc' | 'tcp'
             'url'    : url | host name
             'path'   : only for 'http' and 'grpc'
             'method': only for 'http'
             'port'   : only for 'tcp'}]
```

5 Graph-Based Policy Management

5.1 Motivation

As described in Challenge C3, multiple service versions that exist simultaneously may have different inter-service dependencies in a microservice application. To distinguish these versions in access control, a strawman approach is specifying permissions for each version separately. However, although inter-service requests may differ in various service versions, they serve similar responsibilities because they belong to the same microservice. Hence most requests should be identical. Consequently, generating separate access control policies for each version may introduce plenty of redundancy, which will decrease the application's performance in two aspects:

(1) Additional policy checks. At runtime, a proxy needs to match the incoming requests with all installed policies individually. Thus redundant policies will involve additional policy checking time (as shown in Figure 5).

(2) Unnecessary policy installations. The installation of redundant policies is not free. Due to caching and other propagation overhead, it will take seconds before the policies are activated, which may also affect the data plane's performance.

Therefore, to defeat policy redundancy, we design a *permission graph* to depict inter-service dependencies and guide policy generation, which can be formalized as follows.

5.2 Data Structure

A *permission graph* $G = (N_s, N_v, E_b, E_r)$ consists of the sets of two kinds of *permission nodes* (N_s, N_v) and two kinds of *edges* (E_b, E_r). A *service node* $a \in N_s$ describes the permissions (i.e., the requests a microservice can initiate) common to all versions of the service, while a *version node* $a_i \in N_v$ describes the permissions specific to that version. Besides, a *belonging edge* in E_b connects a *version node* with the *service node* it belongs to, and a *request edge* in E_r indicates a possible inter-service request.

Take Figure 8 as an example. It shows an application composed of four microservices A, B, C, and D. Figure 8 (a) shows the service behavior in this application. As we can see, version $V1$, $V2$, and $V3$ of service A can all initiate a request $r1$ to service B, whereas $V2$ may also call service C, D with request $r3$, $r2$, and $V3$ may send $r4$ to service D. In this case, as shown in Figure 8 (b), we place their common permissions in service node a, and put the unique permissions of $V2$ and $V3$ in the version nodes a_2, a_3, respectively. Note that the version node a_1 still exists, but its permission is *null*.

By this means, we no longer need to maintain the permissions shared by each service version separately. This not only reduces the redundant policies and optimizes policy enforcement, but also eliminates unnecessary policy installation and realizes on-demand policy distribution. Consider an application with s services that can initiate inter-service invocations. On average, suppose each of these services has v versions, each version may send r different requests, and the share of the requests that can be initiated by all versions is x. If we generate a policy for every request, comparing to the strawman approach, the total number of saved policies will be:

$$policy_{saved} = s(vr - (vr(1-x) + rx)) = srx(v-1)$$

which can be substantial for large-scale applications.

The extraction of shared permissions could involve numerous permission node comparisons. To accelerate this process, we represent each permission node with a hash-based *skeleton tree*, whose leaves are the abstractions of the requests it

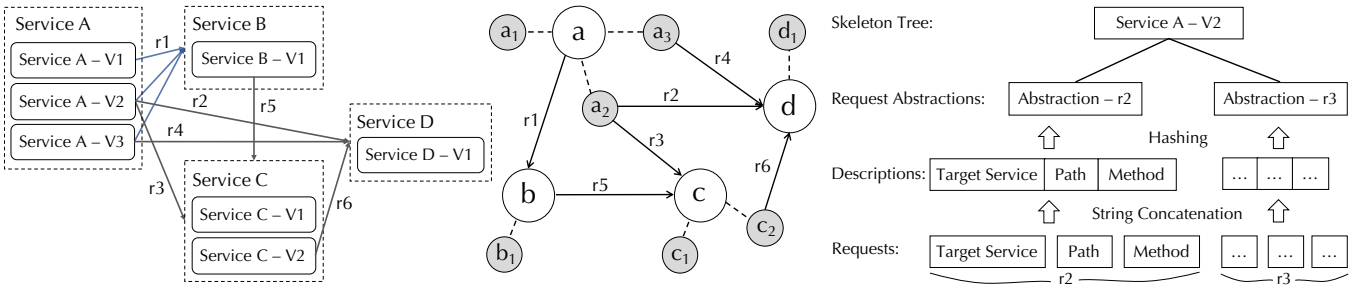

Figure 8: A demonstration of the permission graph and the skeleton tree representation of permission node a_2.

may launch. The abstraction of a request is a hash value for a string concatenation of its *target service*'s name and a series of attributes like *path* and *method*. Figure 8 (c) demonstrates the skeleton tree representation of version node a_2. Hence, comparing two permission nodes is transformed into solving the intersection of two sets, and the time complexity is $O(m+n)$, where m, n are the number of these nodes' permissions. Through this *skeleton tree comparison* method, we can quickly extract the permissions shared by all service versions and identify their differences if they are not identical.

5.3 Policy Generation

When a microservice is being deployed, AUTOARMOR generates access control policies based on its manifest file. First, the manifest file will be transformed into a permission node and added to the application's permission graph. We then translate each related *request edge* into an access control policy.

One thing to consider in this process is the deployment order of services. A permission should not be granted if the request's target service has not been deployed. Otherwise, it will result in over-authorization. Therefore, we mark the requests whose target services have not been deployed, that is, the *request edges* with a missing endpoint in the *permission graph*, and postpone the corresponding policy generation to the deployment of these callee services.

In the initial stage of an application, the administrator usually deploys a series of microservices simultaneously with a single deployment file. In this case, we abstract the *permission graph* to a *directed acyclic graph* (DAG) at service granularity, and perform *topological sorting* on it. After that, we reorganize the deployment file in reverse topological order and generate access control policies to ensure that the entire application comes up quickly and correctly.

5.4 Policy Update

The interdependencies of microservices evolve with the application behavior. This dynamic nature requires that the access control policies keep updated over time. *Service upgrade* and *service rollback* are two operations that cause microservice application changes. Owing to the progressive upgrade strategy, they can be regarded as the addition and removal of specific versions of services.

Algorithm 1: Add the version v of service s

Input : G - current permission graph
P_{sv} - permission set for the version v of service s
Output : G' - updated permission graph

```
1  n_s = get the service node of s
2  P_s = get the permission set of n_s
3  if P_s ⊂ P_sv then
4      P'_sv = P_sv − P_s
5  else
       // split the service node
6      P'_sv = P_sv − (P_s ∩ P_sv)
7      P_d = P_s − (P_s ∩ P_sv)
8      remove permissions in p_d from n_s
9      V_s = get version nodes of s
10     for n_v in V_s do
11         add permissions in P_d to n_v
12 n_sv = generate version node for v with P'_sv
13 G' = add n_sv to G
14 return G'
```

Service Addition. When deploying a new service version, we first compare its permission node with the corresponding *service node*. If its permissions are a superset of the service node, we keep the service node unchanged and insert the unique permissions of this version as a *version node* into the permission graph. If the service node contains permissions not in this version, which means this version will not send some requests that other versions will initiate, we need to split the service node. That is, put the permissions shared by other versions to those version nodes and generate a new service node. Algorithm 1 describes the entire process.

Service Removal. When a specific service version is offline, we need to remove the corresponding version node. If this version is the only version of its service, the relevant service node will be deleted at the same time. Note that removing a node also implies removing all of its outgoing edges and inactivating its incoming edges. After the removal, we reperform the permission aggregation for the remaining version nodes to ensure that the permission set of the service node is the largest subset of the permissions shared by each version. This process is demonstrated in Algorithm 2.

Algorithm 2: Remove the version v of service s

Input : G - current permission graph
s_v - the version v of service s
Output : G' - updated permission graph

n_{sv} = get the version node of s_v
G' = remove n_{sv} from G
n_s = get the service node of s
V_s = get the version nodes of s
if V_s *is* $\emptyset$ **then**
    remove n_s from G'
else
    P_c = extract common permissions of versions in V_s
    add permissions in P_c to n_s
return G'

Components	Target	Languages	Base	Modified LoCs
Static Analysis Engine	Java	Java	SonarJava	~4000
	JavaScript	Java	SonarJS	~3000
	Python	Java	SonarPython	~2000
	C#	C#	Roslyn	~3000
	Go	Go	Go Tools	~2000
	Ruby	Ruby	Parser	~1000
Permission Engine		Python	-	~1000
Policy Generator		Python	-	~500

Table 4: The implementation detail of AUTOARMOR.

Complexity. Let n be the number of permissions of the service version to be processed, and m be the number of versions this microservice has. Assuming that the proportion of permissions shared by these versions is a constant x, the best and worst time complexity to deal with the addition of this version is $O(n)$ and $O(mn)$, respectively. As for the removal of this version, due to the permission re-aggregation, the corresponding complexity is $O(mn)$.

6 Evaluation

We have implemented a prototype of AUTOARMOR on *Kubernetes* [42] and *Istio* [22], two of the most popular microservice infrastructures at present. The prototype is implemented in three components, with ~16,500 LoCs in total. A detailed breakdown is shown in Table 4. To adapt to our evaluation applications, the *Static Analysis Engine* is implemented on the basis of a series of existing static analysis tools, including *SonarJava* [37], *SonarJS* [38], *SonarPython* [39], *Roslyn* [31], *Go Tools* [17], and *Parser* [49].

In this section, we present our evaluation results by testing AUTOARMOR's functionality and performance, as shown in Table 5. Specifically, our evaluation seeks to answers the following six questions:

- **Q1:** Can AUTOARMOR extract the invocation logic among microservices? (§6.2, E1: Effectiveness)
- **Q2:** How much time does it take to complete the offline program analysis? (§6.2, E2: Analysis Time)
- **Q3:** Can the generated access control policies enhance application security? (§6.3, E3: Security Evaluation)

Evaluation	Request Extraction	Policy Management
Functionality	E1: Effectiveness	E3: Security Evaluation
Performance	E2: Analysis Time	E4: Efficiency E5: Scalability E6: End-to-End Performance

Table 5: Our experiment design.

Name	# of Svcs	LoCs	Type	Multi-Lang	★ on GitHub
Bookinfo [20]	6	2,702	Demo	✓	24.7k
Online Boutique [18]	11	23,219	Demo	✓	8.8k
Sock Shop [48]	13	20,150	Demo	✓	2.5k
Pitstop [14]	13	45,028	Demo	✗	630
Sitewhere [36]	21	53,751	Industrial	✗	717

Table 6: The microservice applications used in our evaluation.

- **Q4:** Is it efficient to generate, manage, and update the access control policies? (§6.3, E4: Efficiency)
- **Q5:** Can it be applied to large-scale microservice applications? (§6.3, E5: Scalability)
- **Q6:** Can it optimize the performance of policy enforcement at runtime? (§6.3, E6: End-to-End Performance)

6.1 Evaluation Environment

As shown in Table 6, our prototype was evaluated with five popular open-source microservice applications. The first four are demonstration applications for teaching purposes, and the fifth is an industrial application: *Bookinfo* [20] is a simple example application of *Istio*, *Online Boutique* [18] and *Sock Shop* [48] are relatively complex e-commerce website applications, *Pitstop* [14] is a garage management system, and lastly, *Sitewhere* [36] is an industrial IoT application enablement platform. These applications contain 64 unique services, and all of them are deployed in containers. Since these applications are designed to accomplish different tasks, they have no service overlap. Assessing how services applied in multiple applications affect the policy generation is future work.

The microservice infrastructure used in our experiments is a 3-node Kubernetes cluster (v.1.18.6) with Istio (v1.6.8). Each node is equipped with eight 2.30-GHz Intel(R) Core(TM) CPUs (i5-8259U) and 32 GB of RAM.

6.2 Request Extraction Evaluation

E1: Effectiveness. Among the 64 unique microservices out of our 5 applications, 48 are *business* services, and the rest are *infrastructure* services that do not contain business code and only provide functions such as data storage and message queues. Therefore, our evaluation of the request extraction mechanism only considers business services.

The effectiveness of request extraction is indicated by two metrics: (1) whether it can identify inter-service invocations in the code, and (2) whether it can extract the detailed attributes of these invocations. To measure them, we first model the inter-service invocation libraries used in each application,

Application	Microservice	Language	LoCs	Identified Requests			Extracted Attributes			Time
				HTTP	gRPC	TCP	URL	Method	Port	
Bookinfo	productpage	Python	2,061	3/3	-	-	3/3	3/3	N/A	21s
	details	Ruby	122	1/1	-	-	1/1	1/1	N/A	4s
	reviews	Java	301	1/1	-	-	1/1	1/1	N/A	27s
	ratings	JavaScript	218	-	-	2/2	2/2	N/A	2/2	27s
Online Boutique	frontend	Go	3,666	-	11/11	-	11/11	N/A	N/A	35s
	cartservice	C#	5,941	-	-	7/7	7/7	N/A	7/7	38s
	productcatalogservice	Go	2,460	-	-	-	N/A	N/A	N/A	18s
	currencyservice	JavaScript	359	-	-	-	N/A	N/A	N/A	25s
	paymentservice	JavaScript	343	-	-	-	N/A	N/A	N/A	26s
	shippingservice	Go	2,458	-	-	-	N/A	N/A	N/A	18s
	emailservice	Python	2,146	-	-	-	N/A	N/A	N/A	20s
	checkoutservice	Go	2,816	-	8/8	-	8/8	N/A	N/A	21s
	recommendationservice	Python	2,112	-	1/1	-	1/1	N/A	N/A	28s
	adservice	Java	918	-	-	-	N/A	N/A	N/A	29s
Sock Shop	front-end	JavaScript	9,922	33/33	-	-	33/33	33/33	N/A	125s
	orders	Java	2,187	6/6	-	2/2	4/8	6/6	2/2	55s
	payment	Go	863	-	-	-	N/A	N/A	N/A	11s
	user	Go	2,515	-	-	24/24	24/24	N/A	24/24	33s
	catalogue	Go	1,439	-	-	8/8	8/8	N/A	8/8	23s
	carts	Java	1,840	-	-	7/7	7/7	N/A	7/7	48s
	shipping	Java	929	-	-	3/3	3/3	N/A	3/3	34s
	queue-master	Java	926	-	-	3/3	3/3	N/A	3/3	31s
Pitstop	webapp	C#	40,461	16/16	-	-	16/16	16/16	N/A	52s
	customermanagementapi	C#	423	-	-	5/5	5/5	N/A	5/5	19s
	vehiclemanagementapi	C#	451	-	-	5/5	5/5	N/A	5/5	18s
	workshopmanagementapi	C#	1,563	4/4	-	20/20	24/24	4/4	20/20	46s
	workshopmanagementeventhandler	C#	685	10/10	-	14/14	24/24	10/10	14/14	30s
	auditlogservice	C#	136	1/1	-	2/2	3/3	1/1	2/2	7s
	notificationservice	C#	511	7/7	-	12/12	19/19	7/7	12/12	42s
	invoiceservice	C#	641	9/9	-	14/14	23/23	9/9	14/14	45s
	timeservice	C#	157	1/1	-	1/1	2/2	1/1	1/1	7s
Sitewhere	web-rest	Java	6,648	-	215/215	-	215/215	N/A	N/A	242s
	instance-management	Java	4,069	-	-	35/35	35/35	N/A	35/35	99s
	event-sources	Java	6,619	-	1/1	3/3	4/4	N/A	3/3	130s
	inbound-processing	Java	825	-	2/2	4/4	6/6	N/A	4/4	49s
	device-management	Java	6,381	-	-	74/74	74/74	N/A	74/74	156s
	event-management	Java	4,799	-	4/4	60/60	64/64	N/A	60/60	204s
	asset-management	Java	5,993	-	-	10/10	10/10	N/A	10/10	142s
	schedule-management	Java	1,964	-	-	10/10	10/10	N/A	10/10	77s
	batch-operations	Java	2,122	-	6/6	16/16	22/22	N/A	16/16	105s
	device-registration	Java	1,075	-	10/10	4/4	14/14	N/A	4/4	57s
	device-state	Java	1,739	-	1/1	7/7	8/8	N/A	7/7	61s
	event-search	Java	769	4/4	-	-	4/4	4/4	N/A	34s
	label-generation	Java	1,379	-	10/10	-	10/10	N/A	N/A	66s
	rule-processing	Java	1,091	-	2/2	2/2	4/4	N/A	2/2	50s
	command-delivery	Java	3,417	-	6/6	3/3	9/9	N/A	3/3	123s
	streaming-media	Java	736	-	-	10/10	10/10	N/A	10/10	49s
	outbound-connectors	Java	4,125	-	13/13	2/2	15/15	N/A	2/2	145s
Total	48 unique services	6 languages	-	96/96	290/290	369/369	751/755	96/96	369/369	-

Table 7: The request extraction evaluation for business services in our evaluation applications. In this table, `A/B` means that there are `B` requests or attributes in the code, and `A` of them are successfully identified or extracted. (E1, E2)

mark the code folders and configuration files, and then use the *Static Analysis Engine* to analyze all services and record the identified requests and extracted attributes. Finally, we compare the analysis results with the ground truth obtained by manual analysis for verification.

Table 7 shows the result of this experiment. AUTOARMOR identified 96 HTTP requests, 290 gRPC requests, and 369 TCP requests from the code of these services, which achieved 100% coverage in request identification. In terms of attribute extraction, AUTOARMOR achieved a 100% extraction rate for the *method* and *port* attributes whose extractions are more dependent on the usage models and configuration files. Regarding URL extraction that relies more on program slicing, the extraction rate is 99.5%. 0.5% (4/755) of the invocations' URLs could not be obtained. This is because *Sock Shop*'s *orders* service directly accesses the URLs in its received requests. These URLs do not exist in its code and cannot be extracted. This incompleteness indicates some limitations of AUTOARMOR. In the following, we rigorously discuss the false negatives and false positives, and define the precise conditions under which a request can be properly extracted.

False Negative. Overall, three situations may lead to false negatives in request extraction, thereby invalidating the subsequent access control:

(1) Attributes from the input. Such attributes cannot be obtained by static analysis, as they do not exist in the code or configuration files. Nevertheless, reckless access to the URLs in incoming requests increases the risk of injection attacks. Introducing input validation and sanitization or dynamic analysis could be helpful in this situation.

(2) Incomplete library modeling. The identification of requests relies on the usage and semantic models of inter-service communication libraries. Invocations initiated via unmodeled libraries will not be covered, leading to incomplete request identification. Therefore, AUTOARMOR requires efforts to model the used invocation libraries.

(3) Sophisticated parameter processing. As a proof-of-concept prototype, AUTOARMOR currently does not support some advanced features. For example, *aliasing* reflects changes to one variable to another. This may result in incomplete program slices since we only perform one-way taint propagation. These advanced features may lead to further research challenges. Since we did not witness such a case in the collected applications and are not committed to prompting current static analysis techniques, we believe this is not a fundamental limitation. Support for the advanced features of different programming languages is future work.

False Positive. Due to the conservative extraction strategy, our request extraction is more prone to false negatives than false positives. However, deserted or dead code could indeed cause false positives, even if the modeling is correct. Removing such code in time or enhancing AUTOARMOR with *invalid code analysis* could alleviate this problem.

Conditions for Correct Extraction. Based on the results of E1 and our implementation, we summarize the sufficient conditions for correct request extraction as follows:

(1) All attributes are contained in the code or supported configuration files. These materials are the target of static analysis. Therefore, if some attributes are not included, they must not be obtained via static analysis. For the *path* attribute, since some fields may come from input requests, partial inclusion is also acceptable. However, a completely missing request URL will cause the callee service's hostname not to be identified. This is also the reason why the URL extraction of the four requests in E1 failed. Our experiment result shows that most of the requests can meet this condition.

(2) CFGs and CGs can be constructed from the service code. They are data structures on which static analysis relies. Both request identification and program slicing are performed on them. Incomplete call graphs will cause the interruption of taint propagation and affect the integrity of program slices. Thanks to the existing static analysis tools, both CFGs and CGs can be built smoothly in our experiment.

(3) The involved inter-service invocation libraries are completely modeled. Usage models are the core for request identification. Although the communication methods used in each application may be different, as shown in Table 1, the invocation methods are fairly uniform in a single application, and the number of used libraries is also limited. Thus, complete modeling of them is feasible. Besides, some applications encapsulate dedicated libraries to initiate inter-service invocations (e.g., *Sitewhere*). Therefore, it is inevitable to model them to identify the corresponding invocations.

(4) The URL constructions do not involve complex string operations. We extract the URL mainly to obtain the callee service's service name (*hostname*) and the *path* attribute. gRPC requests' paths are defined in the `.proto` file as part of the API definition, so it will not be modified in the code. Moreover, the hostnames and ports of TCP requests are usually defined in the deployment or configuration files. For an HTTP request, unlike ordinary strings, the URL has special semantics. Therefore, it usually does not involve complicated string operations, especially in small programs such as microservices. For example, although we considered the possibility of complex string operations, we did not encounter intricate URL processing in the evaluation, such as modifying in loops or using aliasing. Nonetheless, if there are complex operations on URLs in the target application, a series of work [45,47,56] dedicated to string analysis can provide support. Exploring too profoundly in this aspect is beyond the scope of this paper.

E2: Analysis Time. As shown in Table 7, AUTOARMOR takes 57s to extract the requests from a microservice's code on average. This result varies according to the amount of code and the requests initiated by each service. Since we only focus on the key parameters when starting program slicing, and the number of variables involved in constructing the useful attributes (e.g., *URL* and *Method*) is usually limited, the state space is significantly narrowed. Moreover, microservices reduce a single service's internal complexity, which further facilitates the efficient execution of static analysis. Overall, we consider the analysis time acceptable for an offline process performed before service building.

6.3 Policy Management Evaluation

E3: Security Evaluation. This experiment is to verify the security improvements AUTOARMOR brings to microservice applications and the correctness of the generated policies. To this end, according to the threat model defined §3.1, we generate three different types of unauthorized requests as simulated attacks (*A1–A3*) to challenge three of our evaluation applications with and without AUTOARMOR installed.

A1: Accessing Unauthorized Microservices. This attack refers that a microservice initiates a request to an endpoint of a service that it has no dependencies on. We assume that any service in the application can be compromised. Thus, to generate such unauthorized requests, we start from each service and build a request pointing to all endpoints of all services that the current service should not access.

Application		BookInfo		O-Boutique		Sock Shop	
w/ AUTOARMOR Policies		✗	✓	✗	✓	✗	✓
A1	# of Requests	140 (78%)		316 (79%)		2,092 (85%)	
	Blocked Requests	0	140	0	316	0	2,092
A2	# of Requests	24 (13%)		42 (11%)		290 (12%)	
	Blocked Requests	0	24	0	42	0	290
A3	# of Requests	16 (9%)		40 (10%)		78 (3%)	
	Blocked Requests	0	16	0	40	0	78
Total:	# of Requests	180 (100%)		398 (100%)		2460 (100%)	
	Blocked Requests	0	180	0	398	0	2,460

Table 8: Security evaluation of applications with and without policies generated by AUTOARMOR. (E3)

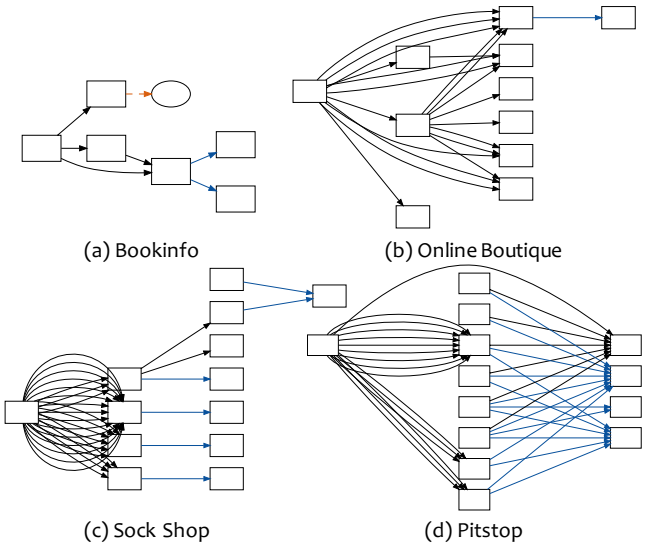

Figure 9: The extracted system behavior of the evaluation applications. (The TCP requests are marked in blue, and the outside requests are marked with dashed orange lines; texts and *Sitewhere* are not shown for space reasons.)(E3)

A2: Accessing Unauthorized Resources. This attack refers that a microservice that can access an endpoint of another service tries to access other endpoints or resources of the target service. We generate such requests by changing the request target to unauthorized endpoints or modifying the legal requests' paths or ports.

A3: Performing unauthorized operations. This attack refers that a microservice uses the wrong methods to access RESTful resources. Such requests can be generated via modifying the methods of legal HTTP requests.

After the generation, we login to each service's container and send the related requests on behalf of them. To record the experiment result, we use *Prometheus* [43] to monitor the processing of the requests. The HTTP status code `403` indicates that the request is blocked. As shown in Table 8, the generated policies can successfully block all three types of attacks. Besides, in E6, we use the load generators provided by these applications to inject external requests to them and find that none of the triggered internal requests were blocked by the policies. This further indicates the correctness of the generated policies: no normal requests are blocked, while unauthorized requests are all blocked.

Request Type	Distribution of Wildcards in URLs	Service-Level Over-Authorization	Affected Attributes
HTTP	30/96 (31%)	0 (0%)	path (100%)
gRPC	0/290 (0%)	N/A	N/A
TCP	0/369 (0%)	N/A	N/A

Table 9: The distribution and impact of wildcards. (E3)

Based on the extracted inter-service invocation logic, we can also plot the actual system behavior models for microservice applications (e.g., Figure 9). In this sense, AUTOARMOR can also help administrators better understand the actual operation logic of microservice applications and timely identify system behaviors that are not consistent with the design.

Wildcards in URLs. As mentioned in §4.2, some variables from the input may introduce wildcards into extracted URLs, leading to a certain degree of over-authorization. We also assessed the distribution and impact of this phenomenon.

GRPC requests will not introduce wildcards due to their fixed paths in the API definitions. TCP requests' authorization granularity is at the port level. Since port information is stable and usually defined in the deployment file, undetermined fields will also not be involved. As shown in Table 9, all the introduced wildcards belong to the URLs of HTTP requests, accounting for 31% of them. Nevertheless, we found that these wildcards are all located in the path attribute and are used to identify a specific object in a collection (e.g., *session_id* and *product_id*). None of them can cause service-level over-authorization. That is, no microservice will be able to access unauthorized targets due to wildcards.

Overapproximation of file system policies may cause security problems. However, it is practical to use wildcards for inter-service access control, especially when configuring manually. Otherwise, it may lead to too many policies. Therefore, we believe that wildcards are acceptable here, and further reduction of the brought over-authorization will be future work. Administrators can also supplement custom policies.

E4. Efficiency. To evaluate the efficiency of our graph-based policy management mechanism, we employ a series of standard events in microservices' lifecycle and measure the processing time in dealing with policy generation, update, and removal tasks. Specifically, with the evaluation applications, we performed the following three steps:

(1) Deployment (DEP): Deploy the evaluation applications as `version 1` services into the infrastructure.
(2) Re-Deployment (Re-DEP): Deploy these applications again and mark the newly deployed services as `version 2`. Note that the requests that can be initiated by `version 1` and `version 2` services are identical.
(3) Removal (REM): Take offline the `version 1` services deployed in the first step.

Through the above three steps, we complete the standard procedure of microservice deployment and a version upgrade.

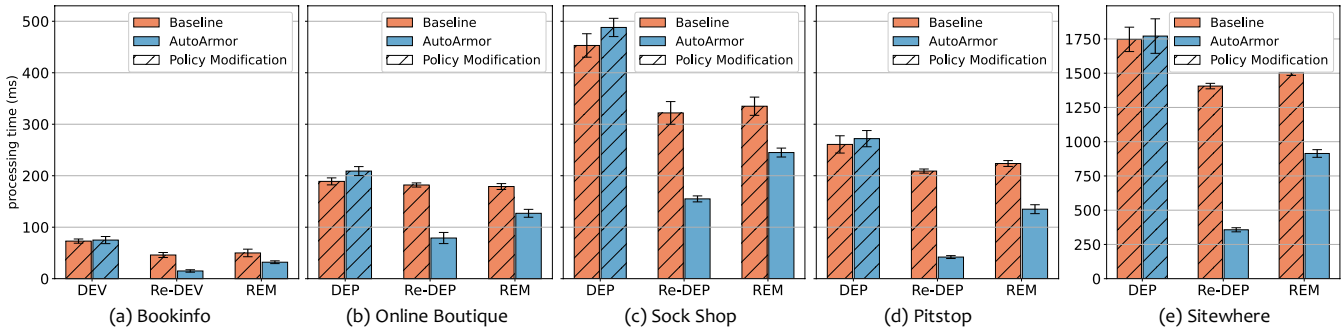

Figure 10: AUTOARMOR's policy management performance for the evaluation applications. (E4)

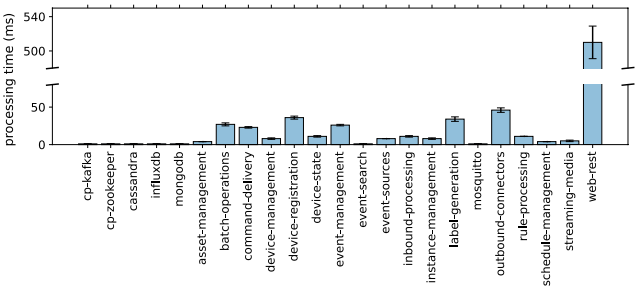

Figure 11: Distribution of processing time for generating policies for *Sitewhere* services. (E4)

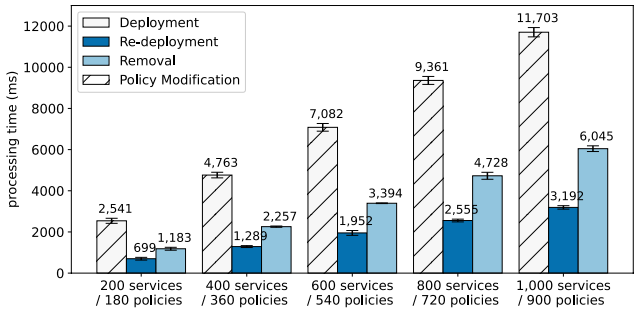

Figure 12: AUTOARMOR's policy management performance for large-scale applications. (E5)

Meanwhile, the tasks of policy generation, update, and removal are created accordingly.

To measure the permission graph's effect in the policy generation phase, we implement "generating policies for all service versions separately" as the baseline method and compare its performance with AUTOARMOR under the same situation. Besides, as mentioned in §5.1, it takes a few seconds for access control policies to propagate and install after being issued, which implies potential overhead. Therefore, we also mark the processes that will trigger policy installation. The experiment result is shown in Figure 10.

As we can see, in the initial deployment, AUTOARMOR is slightly slower than the baseline method due to the permission graph's construction. However, at Re-DEP and REM steps, AUTOARMOR found that the newly deployed/removed services have the same permissions and can be fully covered by the corresponding *service node*. So it would not generate new or modify existing policies. This brings a notable performance improvement and eliminates subsequent policy installations. Moreover, reducing redundant policies can also accelerate runtime policy enforcement, which will be evaluated in E6. In Figure 11, we recorded the time distribution of generating policies for *Sitewhere* services at DEP step. The processing time shows a strong positive correlation with the number of invocations initiated by the service.

Compared to the average deployment time of microservices (40s to 1min per service [26]), the overhead introduced by AUTOARMOR is almost negligible. In this case, even considering the propagation and installation costs, the policies can be fully deployed before a service is ready so that all requests related to this service will be protected.

E5: Scalability. To evaluate whether AUTOARMOR can be applied to large-scale microservice applications, we employ the *Istio load tests mesh* [21], a tool used by Istio for performance and scalability testing. It can generate microservice applications with a large number of services and inject heavy load into them. Due to the limited capacity of the server, we simulate AUTOARMOR's processing for applications of different scales. The experimental method is consistent with the experiments on the evaluation applications (E4). That is, deploying the application, disposing a new version, and removing the original one. The results are shown in Figure 12.

As shown, AUTOARMOR's processing time increases linearly with the scale of microservice applications. Nonetheless, it can complete all policy generation in 12s for an extensive application with 1,000 unique services. This is acceptable in practice considering the service deployment time and indicates its potential for large-scale microservices.

E6: End-to-End Performance. To evaluate the permission graph's effect on runtime policy enforcement, we utilize the end-to-end latency of the evaluation applications as the metric to provide a clear view of the latency reduction for employing AUTOARMOR optimized policies.

In the experiment setup stage, we selected some microservices that have large workloads from the evaluation applica-

Application	External Requests			Average End-to-End Latency				
	URL	Method	Quantity	w/o Policies	w/ Baseline Policies		w/ AutoArmor Policies	
Bookinfo	http://<bookinfo-url>/productpage	GET	100000	319ms	323ms	▲ 4ms(1.25%)	321ms	▼ 2ms (0.62%)
Online Boutique	http://<boutique-url>/	GET	4,400	82ms	86ms	▲ 4ms (4.88%)	84ms	▼ 2ms (2.33%)
	http://<boutique-url>/cart	GET	13,000	80ms	91ms	▲ 11ms (13.75%)	86ms	▼ 5ms (5.81%)
	http://<boutique-url>/cart	POST	13,000	139ms	158ms	▲ 19ms (13.67%)	144ms	▼ 14ms (8.86%)
	http://<boutique-url>/cart/checkout	POST	4,400	112ms	129ms	▲ 17ms (15.18%)	121ms	▼ 8ms (6.20%)
	http://<boutique-url>/product/*	GET	56,000	76ms	85ms	▲ 9ms (11.84%)	82ms	▼ 3ms (3.53%)
	http://<boutique-url>/setCurrency	POST	9,000	91ms	93ms	▲ 2ms (2.20%)	94ms	▲ 1ms (1.08%)
Sock Shop	http://<sockshop-url>/	GET	11,000	95ms	104ms	▲ 9ms (9.47%)	98ms	▼ 6ms (5.77%)
	http://<sockshop-url>/basket.html	GET	11,000	101ms	111ms	▲ 10ms (9.90%)	105ms	▼ 6ms (5.41%)
	http://<sockshop-url>/cart	DELETE	11,000	190ms	204ms	▲ 14ms (7.37%)	197ms	▼ 7ms (3.43%)
	http://<sockshop-url>/cart	POST	10,000	364ms	436ms	▲ 72ms (19.78%)	401ms	▼ 35ms (8.03%)
	http://<sockshop-url>/catalogue	GET	11,000	168ms	177ms	▲ 9ms (5.36%)	169ms	▼ 8ms (4.52%)
	http://<sockshop-url>/category.html	GET	11,000	96ms	105ms	▲ 9ms (9.38%)	98ms	▼ 7ms (6.67%)
	http://<sockshop-url>/detail.html?id=*	GET	11,000	95ms	105ms	▲ 10ms (10.53%)	98ms	▼ 7ms (6.67%)
	http://<sockshop-url>/login	GET	11,000	350ms	373ms	▲ 23ms (6.57%)	367ms	▼ 6ms (1.61%)
	http://<sockshop-url>/orders	POST	9,500	392ms	476ms	▲ 84ms (21.42%)	468ms	▼ 8ms (1.68%)

Table 10: The end-to-end latency of the external requests generated by each evaluation application's load generator. (E6)

Application	Service with Multi-Versions	# of Versions	# of Generated Policies		
			Baseline	AutoArmor	
Bookinfo	reviews	3	8	6	▼ 2 (25%)
Online Boutique	frontend	3	78	21	▼ 57 (73%)
	checkoutservice	5			
	cartservice	4			
Sock Shop	orders	5	57	34	▼ 23 (40%)
	carts	2			
	catalogue	3			

Table 11: The service deployed with multiple versions used in E6 and the number of generated policies. (E6)

tions and deployed multiple versions[3] for them to simulate the real-world production environment. Subsequently, we generate access control policies for all services with the baseline method and AUTOARMOR, respectively. Table 11 shows the specific deployment configuration and the number of generated policies for each experiment.

After that, to generate external requests for end-to-end latency measurement, we utilize the workload generation tools provided by *Online Boutique* and *Sock Shop*, and develop a similar load generator for *Bookinfo* using Locust [8]. We recorded the end-to-end latency of the three applications with policies generated by the baseline method and AUTOARMOR, respectively. In addition, as a control group, we also measured each application's performance without any access control policy installed. The final result is shown in Table 10, where the red and green triangles represent the data changes in the column relative to the previous column.

As we can see, the overall latency shows an increasing trend after applying the baseline policies, and a decreasing trend when the AUTOARMOR policies replace the baseline policies. This result is consistent with the change of the policy quantity in Table 11 and our observation in Figure 5. Nonetheless, the table indicates one case against the general trend. This is because despite the policy enforcement has some influence, the dominating factors that affect the end-to-end performance are still the applications themselves. Besides, AUTOARMOR did not eliminate all inter-service access control policies. To sum up, this experiment shows that through a streamlined policy set, AUTOARMOR can speed up the runtime policy checking, thereby achieving a better end-to-end performance.

[3]The requests that different versions can initiate are identical.

7 Discussion

AUTOARMOR has made the first step towards automatic policy generation for inter-service access control of microservices. However, it is still preliminary and has several limitations for future improvement, which we discuss below.

Source Access of the Microservice Code. In this paper, we assume that administrators can get the source code of microservices. However, there may be situations where the source code is not available, such as the microservices purchased from third parties. These services are generally middlewares or databases, which usually do not initiate requests. Nevertheless, if a service does need to invoke other microservices, it's necessary to configure the appropriate access control policies for it. To this end, there are several options: (1) require the corresponding manifest file from the service provider and review it; (2) manually configure the access control policy for it; (3) employ reverse engineering or static analysis for binary code. Our future work will seek methods to develop binary analysis approaches to achieve more general business logic extraction.

Trustworthiness of the Microservice Code. We also assume that the code of microservices is benign. Although the mature code review mechanisms guarantee this to some extent, as artifacts, bugs in the code are inevitable. Multi-company collaborations or third-party services can also lead to untrusted code. In these cases, the administrators can be involved in the review of manifest files. This is an offline process and therefore does not increase the runtime overhead. In

addition, the administrators can periodically check the system behavior model (e.g., Figure 9) to detect possible anomalies.

Incompleteness of Request Extraction. Although AUTOARMOR is committed to extracting the complete inter-service invocation logic, as discussed in E1, it still has false positives or false negatives. Since false negatives lead to false denials, every effort must be made to reduce them. Therefore, in practice, developers should try to program in a unified style to facilitate static analysis, or even use annotations to ensure the completeness. Combining it with dynamic methods can also benefit the soundness. After deploying AUTOARMOR for a new application, the administrator can first use dynamic testing or manual inspection to evaluate its results, and make corresponding adjustments (e.g., add missing usage models or define supplementary policies). After that, as a member of the CI/CD automation pipeline, it can truly realize its potential.

Granularity of Access Control. At present, AUTOARMOR is dedicated to generating policies to indicate whether microservice can access resources. It attempts to achieve fine-grained authorization with detailed attributes. However, there are always ongoing efforts for finer granularity of access control. To be integrated with existing infrastructures, AUTOARMOR follows current policy-based mechanisms, but is also limited by the expression ability of policies.

Attacks That May Still Occur. AUTOARMOR uses wildcards to represent undetermined fields in request paths, which may cause over-authorization. Nevertheless, this kind of over-authorization may be unavoidable, and it is also a common practice in the real-world. Input validation and sanitization may alleviate such problems. Besides, administrators can deploy their own access control policies based on other insights, which is also a powerful supplement to the AUTOARMOR policy set. Moreover, the adversary can still launch *mimicry attacks* [46], that is, imitating normal system behavior, and trying to cause damages in the permission space and avoid being blocked or detected. Access control is incompetent against such attacks. They may require other security mechanisms, such as *Intrusion Detection Systems* (IDS) and rate limiting.

8 Related Work

Service Dependency Extraction. To obtain the dependencies among microservices at a fine-grained level, many different methods have been adopted in the present works. They can be divided into two categories, namely *intrusive work* and *non-intrusive work*. Manual annotation [57] and code injection (e.g., [12, 15, 40]) are two common kinds of intrusive work. Although these methods can guarantee the full coverage of service dependencies, the modification of the source code of applications is required. On the contrary, non-intrusive methods obtain the service dependencies through network packet/flow inference (e.g., [2,4,5,52]) or log mining (e.g., [1,28,50,54]), which do not intrude into source code but cannot guarantee that all APIs will be covered. AUTOARMOR can be classified as non-intrusive work, but it does not require actual program execution due to static analysis.

Automatic Security Policy Generation for Distributed Systems. Currently, these methods can be divided into three categories. The *document-based approaches* (e.g., [3,34,51, 55]) usually generate security policies by applying NLP methods to analyze the security requirements and syntax description. Although the documents could better express the developer's intentions, they are not always there. Meanwhile, these approaches are usually coarse-grained and incomplete due to the limitation of NLP [51]. The *history-based approaches* (e.g., [23, 35, 50]) utilize the collected traces or historical data to infer the rule criteria and policy structure from the traffic. Thus, their effectiveness depends on the granularity and completeness of the traces, which is hard to be guaranteed. Besides, they require applications to run in advance to collect data, which may cause attack windows. The last category is *model-based approaches* (e.g., [7,25]), which manually build models to understand the security requirement of the system, then generate security policies accordingly. However, the modeling process is time-consuming and error-prone, making them not suitable for flexible microservice applications. Seamlessly integrated with the microservice lifecycle and infrastructure, AUTOARMOR can build accurate real-time system behavior models, thus conquering their weaknesses.

Methods Using Static Analysis to Generate Policies. Using static analysis for policy generation is not a recent innovation. In the host security field, many studies [10,24,32] have tried to obtain privileged behaviors required by programs, such as file access or system calls, through static analysis. They then generate different types of access control policies accordingly to reduce security risks. A recent study [16] applied this idea to containers. It utilized code analysis to extract the system calls required by containerized applications and generated Seccomp policies to narrow the attack surface. Unlike these efforts, AUTOARMOR is active at the application layer of the modern cloud environment; it extracts the horizontal invocations between services and infers the corresponding permissions, thereby solving the policy generation gap for inter-service access control of microservices.

Policy Dynamic Update. When the system status changes (e.g., service deployment/removal or security demands are modified), the security policies need to be updated to adapt to the new status. The general procedure for updating policies starts with analyzing and verifying changes from the intents, and consequently constructing a reference model to obtain the minimal update (e.g., [19, 44]). However, these works can only handle high-level update requests from administrators and cannot respond to the system changes. Besides, it can be computationally expensive to obtain minimal updates through formal methods. In contrast to them, AUTOARMOR

is committed to starting with the system changes and strives to respond to them quickly and reasonably.

Graph-Based Policy Management. Abstracting policies to graphs is the state-of-the-art of automatic policy management. Various graph structures can be constructed based on different intentions and policy definitions. Xu *et al.* [53] construct a directed graph to model the information flows in the system for integrity. Prakash *et al.* [33] leverage a network communication graph to detect and resolve network policy conflicts as well as to model and compose service chaining policies. Chen *et al.* [11] build a permission event graph that connects permissions events and handlers to specify and enforce policies automatically. Different from the structures used in these works, we design a novel permission graph that ingeniously combines the characteristics of microservices. On this basis, we reduce the redundant policies and realize the agile generation and update for inter-service access control policies.

9 Conclusion

We have presented AUTOARMOR, the first automatic policy generation tool for inter-service access control of microservices. It includes two fundamental techniques: (1) a static analysis-based request extraction mechanism that can extract the inter-service invocation logic of the application, and (2) a graph-based policy management mechanism that can swiftly generate fine-grained access control policies and keep them up-to-date over time. We have implemented a prototype for *Kubernetes* and *Istio*, and tested it with five popular microservice applications. Our experimental results show that AUTOARMOR can effectively bridge the policy generation gap of the inter-service access control for microservices with only a minor overhead.

Acknowledgments

We thank Xue Leng for the assistance and many constructive discussions. We also thank the anonymous reviewers and our shepherd, Trent Jaeger, for their insightful comments.

References

[1] Manoj K. Agarwal, Manish Gupta, Gautam Kar, Anindya Neogi, and Anca Sailer. Mining activity data for dynamic dependency discovery in e-business systems. *IEEE Transactions on Network & Service Management*, 1(2):49–58, 2004.

[2] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP*, 2003.

[3] 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.

[4] Paramvir Bahl, Ranveer Chandra, Albert G. Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 13–24, 2007.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[6] Kaitlyn Barnard. Cncf survey: Use of cloud native technologies in production has grown over 200%. http://bit.ly/cncf-survey, 2018. Accessed on 2020-01-20.

[7] 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.

[8] Carl Byström, Jonatan Heyman, Joakim Hamrén, and Hugo Heyman. Locust - a modern load testing framework. https://locust.io/, 2019. Accessed on 2020-01-20.

[9] Brad Casemore and Mehra Rohit. Vendors stake out positions in emerging istio service mesh landscape. http://bit.ly/idc05, 2018. Accessed on 2020-01-20.

[10] Paolina Centonze, Robert J Flynn, and Marco Pistoia. Combining static and dynamic analysis for automatic identification of precise access-control policies. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 292–303. IEEE, 2007.

[11] Kevin Zhijie Chen, Noah M. Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R. Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[12] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *2002 International Conference on Dependable Systems and Networks (DSN)*, pages 595–604, 2002.

[13] Dimensional Research. Global microservices trends report. http://bit.ly/lightstep, 2018. Accessed on 2020-01-20.

[14] EdwinVW. Pitstop: Garage management application. https://github.com/EdwinVW/pitstop/, 2020. Accessed on 2020-01-20.

[15] Ulfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: Extensible distributed tracing from kernels to clusters. *Acm Transactions on Computer Systems*, 30(4):1–35, 2012.

[16] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[17] Google. Go tools: tools that support the go programming language. https://github.com/golang/tools, 2019. Accessed on 2020-01-20.

[18] Google Cloud Platform. Online boutique: Cloud-native microservices demo application. http://bit.ly/online-boutique, 2019. Accessed on 2020-01-20.

[19] Jinwei Hu, Yan Zhang, and Ruixuan Li. Towards automatic update of access control policy. In *Uncovering the Secrets of System Administration: Proceedings of the 24th Large Installation System Administration Conference, LISA*, 2010.

[20] Istio. Istio / bookinfo application. http://bit.ly/3dzSsBv, 2019. Accessed on 2020-01-20.

[21] Istio. Istio / Performance and Scalability. http://bit.ly/load657, 2019. Accessed on 2020-01-20.

[22] Istio. Istio: Connect, secure, control, and observe services. https://istio.io/, 2019. Accessed on 2020-01-20.

[23] 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.

[24] Sven Lachmund. Auto-generating access control policies for applications by static analysis with user input recognition. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, pages 8–14, 2010.

[25] 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.

[26] Chan-Yi Lin, Ting-An Yeh, and Jerry Chou. DRAGON: A dynamic scheduling and scaling controller for managing distributed deep learning jobs in kubernetes cluster. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science, CLOSER*, pages 569–577, 2019.

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

[28] Shang Pin Ma, Chen Yuan Fan, Yen Chuang, Wen Tin Lee, Shin Jie Lee, and Nien Lin Hsueh. Using service dependency graph to analyze and test microservices. In *IEEE Computer Software & Applications Conference*, 2018.

[29] Abner Mendoza and Guofei Gu. Mobile application web api reconnaissance: Web-to-mobile inconsistencies & vulnerabilities. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 756–769. IEEE, 2018.

[30] Benedict Michael and Charanya Vinu. How we built a metering and chargeback system to incentivize higher resource utilization of twitter infrastructure. http://bit.ly/3aETlqs, 2017. Accessed on 2020-01-20.

[31] MicroSoft. The Roslyn .NET compiler. http://bit.ly/roslyn1, 2019. Accessed on 2020-01-20.

[32] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–26, 2020.

[33] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: using graphs to express and automatically reconcile network policies. In Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*. ACM, 2015.

[34] 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.

[35] 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.

[36] Sitewhere. Sitewhere: Open source internet of things platform. https://sitewhere.io/, 2020. Accessed on 2020-01-20.

[37] SonarSource. Sonarjava | code quality and code security for java | sonarsource. http://bit.ly/39yopVF, 2019. Accessed on 2020-01-20.

[38] SonarSource. Sonarjs | code quality and code security for javascript | sonarsource. http://bit.ly/3dzQMYC, 2019. Accessed on 2020-01-20.

[39] SonarSource. Sonarpython | code quality and code security for python | sonarsource. http://bit.ly/3dA563q, 2019. Accessed on 2020-01-20.

[40] Byung Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, and Rong N. Chang. vpath: Precise discovery of request processing paths from blackbox observations of thread and network activities. In *Conference on Usenix Technical Conference*, 2010.

[41] Byungchul Tak, Hyekyung Kim, Sahil Suneja, Canturk Isci, and Prabhakar Kudva. Security analysis of container images using cloud analytics framework. In *International Conference on Web Services*, pages 116–133. Springer, 2018.

[42] The Linux Foundation. Production-Grade Container Orchestration - Kubernetes. https://kubernetes.io/, 2019. Accessed on 2020-01-20.

[43] The Linux Foundation. Prometheus - monitoring system & time series database. https://prometheus.io/, 2019. Accessed on 2020-01-20.

[44] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 214–226. 2019.

[45] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1232–1243, 2014.

[46] David Wagner and R Dean. Intrusion detection via static analysis. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 156–168. IEEE, 2000.

[47] Qi Wang, Jingyu Zhou, Yuting Chen, Yizhou Zhang, and Jianjun Zhao. Extracting urls from javascript via program analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 627–630, 2013.

[48] Weaveworks. Microservices demo: Sock shop. https://microservices-demo.github.io/, 2017. Accessed on 2020-01-20.

[49] Whitequark. Parser - a production-ready ruby parser. http://bit.ly/37yLOYf, 2019. Accessed on 2020-01-20.

[50] 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.

[51] 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.

[52] Chen Xu, Zhang Ming, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Usenix Symposium on Operating Systems Design & Implementation*, 2008.

[53] Wenjuan Xu, Xinwen Zhang, and Gail-Joon Ahn. Towards system integrity protection with graph-based policy analysis. In Ehud Gudes and Jaideep Vaidya, editors, *Data and Applications Security XXIII, 23rd Annual IFIP WG 11.3 Working Conference, Montreal, Canada, July 12-15, 2009. Proceedings*, volume 5645 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2009.

[54] Jianwei Yin, Xinkui Zhao, Tang Yan, Zhi Chen, Zuoning Chen, and Zhaohui Wu. Cloudscout: A non-intrusive approach to service dependency discovery. *IEEE Transactions on Parallel & Distributed Systems*, 28(5):1271–1284, 2017.

[55] 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.

[56] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[57] Guobing Zou, Yang Xiang, Pengwei Wang, Shengye Pang, Honghao Gao, Sen Niu, and Yanglan Gan. Extracting business execution processes of api services for mashup creation. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2018.