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∗ Yan Chen Zhiqiang Lin
Zhejiang University Northwestern University The Ohio State University
Xiao Wang Jim Hao Chen
Northwestern University 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
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.

Aiming to elegantly and flexibly develop and maintain complex applications, the microservice architecture (Figure 1 (b)) emerged. It splits an application into several microservices 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

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

---

Figure 1: The evolution of cloud application architecture.
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 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.

**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.

<table>
<thead>
<tr>
<th>Application</th>
<th>Protocol</th>
<th># of Used Libraries</th>
<th># of Initiated Requests</th>
<th>Share of the Main Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bookinfo</td>
<td>HTTP</td>
<td>3 3 5</td>
<td>60%</td>
<td></td>
</tr>
<tr>
<td>Online Boutique</td>
<td>gRPC</td>
<td>2 20</td>
<td>95%</td>
<td></td>
</tr>
<tr>
<td>Sock Shop</td>
<td>HTTP</td>
<td>2 39</td>
<td>85%</td>
<td></td>
</tr>
<tr>
<td>Pitstop</td>
<td>HTTP</td>
<td>2 48</td>
<td>90%</td>
<td></td>
</tr>
<tr>
<td>Sitewhere</td>
<td>HTTP</td>
<td>1 4</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>gRPC</td>
<td>1 270</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TCP</td>
<td>5 240</td>
<td>61%</td>
<td></td>
</tr>
</tbody>
</table>

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.
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
With the above input of a microservice, we aim to extract their code forms. As we can see, their code forms are not the same, and the 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., Java), and some will be compiled into bytecode as an intermediate form (e.g., Go). 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 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.

<table>
<thead>
<tr>
<th>Languages</th>
<th>Java</th>
<th>Python</th>
<th>Go</th>
<th>JavaScript</th>
<th>Ruby</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Code</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bytecode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Binary Code</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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
In the evaluation applications, for gRPC requests, we only need to model the methods that ultimately 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, 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.
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 \( G = (N_r, N_v, E_b, E_r) \) consists of the sets of two kinds of permission nodes \((N_r, N_v)\) and two kinds of edges \((E_b, E_r)\). A service node \( a \in N_r \) 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 \( V_1, V_2, \) and \( V_3 \) of service \( A \) can all initiate a request \( r_1 \) to service \( B \), whereas \( V_2 \) may also call service \( C, D \) with request \( r_3, r_2, \) and \( V_3 \) may send \( r_4 \) 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 \( V_2 \) and \( V_3 \) 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 \( s \). If we generate a policy for every request, comparing to the strawman approach, the total number of saved policies will be:

\[
\text{policy saved} = s(vr - (vr(1 - x) + rx)) = sr(x - 1)
\]

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
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 a2. Hence, comparing two permission nodes is transformed into solving the intersection of two sets, and the time complexity is \( O(m + n) \), where \( m \) and \( n \) 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 callees 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.

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 version, we will send requests to this version and insert the unique permissions of this version as a version node into the permission graph. If the service version 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 re-perform 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.
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 answer 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)
- **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,
<table>
<thead>
<tr>
<th>Application</th>
<th>Microservice</th>
<th>Language</th>
<th>LoCs</th>
<th>Identified Requests</th>
<th>Extracted Attributes</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bookinfo</td>
<td>productpage</td>
<td>Python</td>
<td>2,061</td>
<td>HTTP: 3/3 gRPC: -</td>
<td>URL: 3/3 Method: -</td>
<td>21s</td>
</tr>
<tr>
<td></td>
<td>details</td>
<td>Ruby</td>
<td>122</td>
<td>HTTP: 1/1 gRPC: -</td>
<td>URL: 1/1 Method: 1/1</td>
<td>4s</td>
</tr>
<tr>
<td></td>
<td>reviews</td>
<td>Java</td>
<td>301</td>
<td>HTTP: 1/1 gRPC: -</td>
<td>URL: 1/1 Method: 1/1</td>
<td>27s</td>
</tr>
<tr>
<td></td>
<td>ratings</td>
<td>Java/Script</td>
<td>218</td>
<td>HTTP: 2/2 gRPC: -</td>
<td>URL: 2/2 Method: N/A</td>
<td>27s</td>
</tr>
<tr>
<td></td>
<td>frontend</td>
<td>Go</td>
<td>3,666</td>
<td>HTTP: 11/11 gRPC: -</td>
<td>URL: 11/11 Method: N/A</td>
<td>35s</td>
</tr>
<tr>
<td></td>
<td>cartservice</td>
<td>C#</td>
<td>5,941</td>
<td>HTTP: - gRPC: 7/7</td>
<td>URL: N/A Method: N/A</td>
<td>38s</td>
</tr>
<tr>
<td></td>
<td>productcatalogservice</td>
<td>Go</td>
<td>2,460</td>
<td>HTTP: - gRPC: -</td>
<td>URL: N/A Method: N/A</td>
<td>18s</td>
</tr>
<tr>
<td></td>
<td>currencyservice</td>
<td>JavaScript</td>
<td>359</td>
<td>HTTP: - gRPC: -</td>
<td>URL: N/A Method: N/A</td>
<td>25s</td>
</tr>
<tr>
<td></td>
<td>paymentsservice</td>
<td>JavaScript</td>
<td>343</td>
<td>HTTP: - gRPC: -</td>
<td>URL: N/A Method: N/A</td>
<td>26s</td>
</tr>
<tr>
<td></td>
<td>shippingsservice</td>
<td>Go</td>
<td>2,458</td>
<td>HTTP: - gRPC: -</td>
<td>URL: N/A Method: N/A</td>
<td>18s</td>
</tr>
<tr>
<td></td>
<td>emailservice</td>
<td>Python</td>
<td>2,146</td>
<td>HTTP: - gRPC: -</td>
<td>URL: N/A Method: N/A</td>
<td>20s</td>
</tr>
<tr>
<td></td>
<td>checkoutsservice</td>
<td>Go</td>
<td>2,816</td>
<td>HTTP: 8/8 gRPC: -</td>
<td>URL: 8/8 Method: N/A</td>
<td>21s</td>
</tr>
<tr>
<td></td>
<td>recommendationservice</td>
<td>Python</td>
<td>2,112</td>
<td>HTTP: - gRPC: 1/1</td>
<td>URL: 1/1 Method: N/A</td>
<td>28s</td>
</tr>
<tr>
<td></td>
<td>adsservice</td>
<td>Java</td>
<td>918</td>
<td>HTTP: - gRPC: -</td>
<td>URL: N/A Method: N/A</td>
<td>29s</td>
</tr>
<tr>
<td></td>
<td>front-end</td>
<td>JavaScript</td>
<td>9,922</td>
<td>HTTP: 33/33 gRPC: -</td>
<td>URL: 33/33 Method: N/A</td>
<td>125s</td>
</tr>
<tr>
<td></td>
<td>orders</td>
<td>Java</td>
<td>2,187</td>
<td>HTTP: 6/6 gRPC: 2/2</td>
<td>URL: 4/6 Method: 2/2</td>
<td>55s</td>
</tr>
<tr>
<td></td>
<td>payment</td>
<td>Go</td>
<td>863</td>
<td>HTTP: - gRPC: -</td>
<td>URL: N/A Method: N/A</td>
<td>11s</td>
</tr>
<tr>
<td>Sock Shop</td>
<td>user</td>
<td>Go</td>
<td>2,515</td>
<td>HTTP: 24/24 gRPC: -</td>
<td>URL: 24/24 Method: N/A</td>
<td>33s</td>
</tr>
<tr>
<td></td>
<td>catalogue</td>
<td>Go</td>
<td>1,439</td>
<td>HTTP: 8/8 gRPC: -</td>
<td>URL: 8/8 Method: N/A</td>
<td>23s</td>
</tr>
<tr>
<td></td>
<td>carts</td>
<td>Java</td>
<td>1,840</td>
<td>HTTP: 7/7 gRPC: -</td>
<td>URL: 7/7 Method: N/A</td>
<td>48s</td>
</tr>
<tr>
<td></td>
<td>shipping</td>
<td>Java</td>
<td>929</td>
<td>HTTP: 3/3 gRPC: -</td>
<td>URL: 3/3 Method: N/A</td>
<td>34s</td>
</tr>
<tr>
<td></td>
<td>queue-master</td>
<td>Java</td>
<td>926</td>
<td>HTTP: 3/3 gRPC: -</td>
<td>URL: 3/3 Method: N/A</td>
<td>31s</td>
</tr>
<tr>
<td></td>
<td>webapp</td>
<td>C#</td>
<td>40,461</td>
<td>HTTP: 16/16 gRPC: -</td>
<td>URL: 16/16 Method: N/A</td>
<td>52s</td>
</tr>
<tr>
<td></td>
<td>customermanagementapi</td>
<td>C#</td>
<td>423</td>
<td>HTTP: - gRPC: 5/5</td>
<td>URL: 5/5 Method: N/A</td>
<td>19s</td>
</tr>
<tr>
<td></td>
<td>vehiclenmanagementapi</td>
<td>C#</td>
<td>451</td>
<td>HTTP: - gRPC: 5/5</td>
<td>URL: 5/5 Method: N/A</td>
<td>18s</td>
</tr>
<tr>
<td></td>
<td>workshopmanagementapi</td>
<td>C#</td>
<td>1,563</td>
<td>HTTP: 4/4 gRPC: 20/20</td>
<td>URL: 24/24 Method: N/A</td>
<td>46s</td>
</tr>
<tr>
<td></td>
<td>workshopmanagementeventhandler</td>
<td>C#</td>
<td>685</td>
<td>HTTP: 10/10 gRPC: 14/14</td>
<td>URL: 24/24 Method: 10/14</td>
<td>30s</td>
</tr>
<tr>
<td></td>
<td>auditlogservice</td>
<td>C#</td>
<td>136</td>
<td>HTTP: 1/1 gRPC: 2/2</td>
<td>URL: 3/3 Method: 1/1</td>
<td>7s</td>
</tr>
<tr>
<td></td>
<td>notificationservice</td>
<td>C#</td>
<td>511</td>
<td>HTTP: 7/7 gRPC: 12/12</td>
<td>URL: 19/19 Method: 7/7</td>
<td>42s</td>
</tr>
<tr>
<td></td>
<td>invoicesservice</td>
<td>C#</td>
<td>641</td>
<td>HTTP: 9/9 gRPC: 14/14</td>
<td>URL: 23/23 Method: 9/9</td>
<td>45s</td>
</tr>
<tr>
<td></td>
<td>timeservice</td>
<td>C#</td>
<td>157</td>
<td>HTTP: 1/1 gRPC: 1/1</td>
<td>URL: 2/2 Method: 1/1</td>
<td>7s</td>
</tr>
<tr>
<td></td>
<td>instance-management</td>
<td>Java</td>
<td>4,069</td>
<td>HTTP: - gRPC: 35/35</td>
<td>URL: 35/35 Method: N/A</td>
<td>99s</td>
</tr>
<tr>
<td></td>
<td>event-sources</td>
<td>Java</td>
<td>6,619</td>
<td>HTTP: 1/1 gRPC: 3/3</td>
<td>URL: 3/3 Method: 130s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>inbound-processing</td>
<td>Java</td>
<td>825</td>
<td>HTTP: 2/2 gRPC: 4/4</td>
<td>URL: 6/6 Method: 4/4</td>
<td>49s</td>
</tr>
<tr>
<td></td>
<td>device-management</td>
<td>Java</td>
<td>6,381</td>
<td>HTTP: 7/44 gRPC: 74/74</td>
<td>URL: 74/74 Method: 156s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>event-management</td>
<td>Java</td>
<td>4,799</td>
<td>HTTP: 4/4 gRPC: 60/60</td>
<td>URL: 60/60 Method: 204s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>asset-management</td>
<td>Java</td>
<td>5,993</td>
<td>HTTP: 10/10 gRPC: -</td>
<td>URL: 10/10 Method: 142s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>schedule-management</td>
<td>Java</td>
<td>1,964</td>
<td>HTTP: 10/10 gRPC: -</td>
<td>URL: 10/10 Method: 77s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>batch-operations</td>
<td>Java</td>
<td>2,122</td>
<td>HTTP: 6/6 gRPC: 16/16</td>
<td>URL: 22/22 Method: 16/16</td>
<td>105s</td>
</tr>
<tr>
<td></td>
<td>device-registration</td>
<td>Java</td>
<td>1,075</td>
<td>HTTP: 4/4 gRPC: -</td>
<td>URL: 4/4 Method: 57s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>device-state</td>
<td>Java</td>
<td>1,739</td>
<td>HTTP: 1/1 gRPC: 7/7</td>
<td>URL: 8/8 Method: 61s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>event-search</td>
<td>Java</td>
<td>769</td>
<td>HTTP: 4/4 gRPC: -</td>
<td>URL: 4/4 Method: 34s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>label-generation</td>
<td>Java</td>
<td>1,379</td>
<td>HTTP: 10/10 gRPC: -</td>
<td>URL: 10/10 Method: 66s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rule-processing</td>
<td>Java</td>
<td>1,091</td>
<td>HTTP: 2/2 gRPC: 2/2</td>
<td>URL: 4/4 Method: 50s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>command-delivery</td>
<td>Java</td>
<td>3,417</td>
<td>HTTP: 6/6 gRPC: 3/3</td>
<td>URL: 9/9 Method: 123s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>streaming-media</td>
<td>Java</td>
<td>736</td>
<td>HTTP: 10/10 gRPC: -</td>
<td>URL: 10/10 Method: 49s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>outbound-connectors</td>
<td>Java</td>
<td>4,125</td>
<td>HTTP: 13/13 gRPC: 2/2</td>
<td>URL: 15/15 Method: 22</td>
<td>145s</td>
</tr>
</tbody>
</table>

| Total 48 unique services | 6 languages | HTTP: 100% gRPC: 100% TCP: 100% | URL: N/A Method: N/A Time: 75s Total: 390389 |

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:
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, 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.
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.

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.

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

<table>
<thead>
<tr>
<th>Application</th>
<th>BookInfo</th>
<th>O-Boutique</th>
<th>Sock Shop</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/ AUTOARMOR Policies</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A1</td>
<td># of Requests: 140 (78%)</td>
<td>316 (79%)</td>
<td>2,092 (85%)</td>
</tr>
<tr>
<td>Blocked Requests: 0</td>
<td>140</td>
<td>0</td>
<td>316</td>
</tr>
<tr>
<td>A2</td>
<td># of Requests: 24 (13%)</td>
<td>42 (11%)</td>
<td>290 (12%)</td>
</tr>
<tr>
<td>Blocked Requests: 0</td>
<td>24</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>A3</td>
<td># of Requests: 14 (9%)</td>
<td>40 (10%)</td>
<td>78 (3%)</td>
</tr>
<tr>
<td>Blocked Requests: 0</td>
<td>16</td>
<td>0</td>
<td>40</td>
</tr>
<tr>
<td>Total</td>
<td># of Requests: 180 (100%)</td>
<td>398 (100%)</td>
<td>2,460 (100%)</td>
</tr>
<tr>
<td>Blocked Requests: 0</td>
<td>180</td>
<td>0</td>
<td>398</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Distribution of Wildcards in URLs</th>
<th>Service-Level Over-Authorization</th>
<th>Affected Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>30/96 (31%)</td>
<td>0 (0%)</td>
<td>path (100%)</td>
</tr>
<tr>
<td>gRPC</td>
<td>0/290 (0%)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>TCP</td>
<td>0/369 (0%)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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

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

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


