MeshScope: A Bottom-Up Approach for Configuration Inspection in Service Mesh

Xing Li[†‡], Xiao Wang[‡], and Yan Chen[‡]
[†] Zhejiang University, China [‡] Northwestern University, USA

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; **Software verification and validation**.

KEYWORDS

Service Mesh, Configuration Inspection, Policy Verification

1 INTRODUCTION

The *microservice architecture* has been widely adopted in modern cloud environments. It greatly improves the flexibility of cloud applications by splitting a large and complex application into multiple *microservices*. To manage the communication among services, as an emerging microservice deployment paradigm, *service mesh* goes further. It builds a dedicated communication infrastructure layer that can transparently provide some standard features for microservices, such as load balancing, encryption, and access control. Benefiting from this, developers can focus on their applications' functionalities, and administrators can manage the inter-service communication elegantly and flexibly.

Figure 1 (a) shows a typical service mesh architecture. It deploys a *sidecar* for each microservice instance to proxy inbound and outbound traffic, thereby transparently handling the inter-service communication. To customize the behavior of these sidecars, the administrator can issue management policies to a centralized *control plane*. After receiving a policy, the control plane converts it into configurations for the related sidecars and deploys them via the *data plane* management APIs. Ultimately, all sidecars will work according to the administrator's intention.

This seems promising. Ideally, as shown in Figure 1 (b), the administrator's intentions, system configurations, and actual system behavior should be consistent. However, this is currently a challenging task for service mesh because: (1) The administrator's intention may not be configured correctly. To operate the services as desired, administrators must understand what is happening in the mesh and translate the management intentions into correct policies, which is unintuitive and error-prone. Although some tools make this process more friendly to administrators via interactive methods [3], we can still see a large number of posts daily in the community forum asking how to achieve a specific management intention [1, 2]. (2) The configured policies may not be reflected in the actual behavior of the data plane. Due to misconfigurations [7],

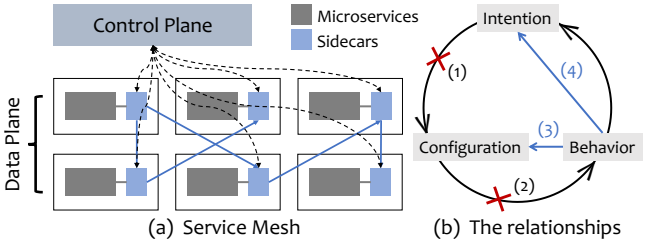


Figure 1: A typical architecture of service mesh. And the relationships among intention, configuration, and behavior.

conversion errors, or semantic differences between planes, sometimes the configured policies will not be installed correctly in the data plane. Also, configurations directly injected into the data plane will not be perceived by the control plane, resulting in unmapped settings between the planes. Besides, the installed data plane policies may not be enforced appropriately because of incorrect data plane implementation or runtime failures.

To ensure the consistencies, there are two types of existing work dedicated to configuration inspection for microservices. The first is configuration validation [3], which checks the correctness of the syntax or semantics of policies to mitigate misconfigurations. The second is configuration audit [4], which checks a series of key parameters to determine whether the configuration conforms to a set of general principles indicated in best practices. However, staying in the control plane, both of them are located in the intention-configuration process and cannot verify the enforcement of policies or describe the actual system behavior.

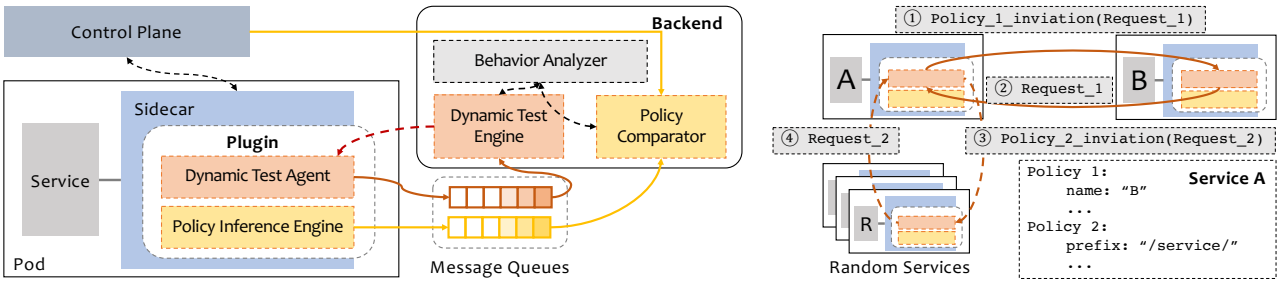
Therefore, the configuration inspection considering the data plane behavior is the missing part. To advance the state-of-the-art, we present MeshScope, a bottom-up approach that can inspect the configuration in service mesh from the perspective of system behavior (Figure 1 (b)(3)), and feedback the actual behavior to the administrator to guide the subsequent configuration (Figure 1 (b)(4)).

To this end, we address two fundamental challenges: (1) how to examine the configurations from the data plane, and (2) how to describe the actual behavior of the mesh. To solve the first challenge, we design a novel hybrid policy verification mechanism. It statically checks the differences between the policies installed in the sidecars and the policies configured in the control plane, and dynamically verifies the enforcement of the installed policies in a distributed manner. To solve the second challenge, we analyze all the inconsistencies found and describe the system behavior in terms of traffic management and security.

Over the past few years, network troubleshooting based on active probe testing has been studied in the context of packet-switched networks [6, 8]. However, the complexity of service mesh policies introduces new challenges to both the generation of test workloads and the verification of system behavior. First, service mesh employs

SIGCOMM '20 Demos and Posters, August 10–14, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8048-5/20/08.
https://doi.org/10.1145/3405837.3411370



(a) MeshScope Architecture (b) Dynamic Verification for Inbound Policies

Figure 2: The architecture of MeshScope and an example of the dynamic policy verification.

various types of policies and rich matching conditions (e.g., service identity, API, path, method, version, etc.) to achieve flexible management. The combination of these conditions and the interaction between policies make it difficult to generate high-quality detection requests. Second, various policy actions, such as load balancing, authorization, and weighted routing, can lead to challenges in deducing and verifying system behavior. Nevertheless, the sidecar's computing power is much stronger than that of the switch so that we can achieve efficient distributed testing and meet the performance requirements in practice.

2 SYSTEM DESIGN

Figure 2 demonstrates the architecture of MeshScope. It comprises three major components: the *plugins* embedded in the sidecar of each microservice instance, a *backend* responsible for managing the tests and analyzing the system behavior, and a series of *message queues* for caching test results.

2.1 Policy Verification

Our hybrid policy verification mechanism includes a continuous static verification and an on-demand dynamic verification.

The static policy verification is mainly used to detect the consistency issues between the control plane and the data plane continuously. As shown in the yellow flow of Figure 2 (a), the *Policy Inference Engine* extracts the configuration installed on the sidecar, such as network filter chains, routing rules, etc., deduces the policies in control plane in reverse, and then passes them to the backend. Subsequently, the *Policy Comparator* at the backend compares the inferred policies with the ones obtained from the control plane to find potential inconsistencies.

The dynamic policy verification is mainly responsible for verifying the enforcement of data plane configurations. We design a set of verification methods for different types of policies. For example, regarding an inbound policy like authorization policy, we perform the probe testing based on an invitation-request mode. Specifically, to receive a desired *probe request* for verification, the *Dynamic Test Agent* generates an *invitation* contains the desired request, and sends it to the expected sender. After receiving the invitation, the sender initiates the specified request to the receiver, thus completing a round of dynamic verification. As shown in Figure 2 (b), to verifying `Policy 1` for service `A`, its *Dynamic Test Agent* sends an invitation to the expected sender (`B`). Afterwards, the sidecar of `B` sends `A` the request it needs to check whether `Policy 1` has been enforced. For the policies with no specific desired sender, such as `Policy 2`, the receiver sends invitations to random services. All sidecars perform this process distributively, but are linked together according to the business logic implied in the policies. Finally, agents send the failed policies to the backend for subsequent analysis.

Typically each installed policy requires one probe request. Nevertheless, for some configurations such as *weighted routing* and *rate limiting*, we may need to generate multiple requests for a policy to check their status. Besides, some types of policies are orthogonal, such as *access control* and *load balancing*, thus we can take advantage of this and verify multiple policies with one probe request.

2.2 Behavior Analysis

With the inconsistencies between system behavior and configurations, we further aim to analyze them and provide a mesh behavior view to guide the administrator's configuration.

First, we can identify the root causes of the collected anomalies. For example, a stable inconsistency between the data plane configuration and the actual behavior implies an incorrect data plane implementation. Second, we can provide preliminary repair suggestions based on some insights. For example, since different instances of the same microservice usually have the same configuration, we can model and calculate the *distance* between services to infer correct configurations. For inconsistencies caused by runtime failures, we can try to resolve them by restarting the service instances.

In service mesh, management intentions may change continuously with evolved business logic and current operating state, which are difficult to predict in advance. Therefore, instead of dealing with the intention, we are committed to presenting the actual system behavior for the administrators. There are two main kinds of policies in service mesh: traffic management and security. For the former, we mainly show whether the mesh's behavior is consistent with the configuration. In the security aspect, we list all possible operations in the current configuration space.

3 ONGOING AND FUTURE WORK

Currently, we are working on the implementation of the proposed policy verification mechanism. We intend to employ popular microservices applications, such as Online Boutique [5], and policy sets to reasonably evaluate MeshScope. In future work, we aim to utilize emerging technologies to investigate the identified inconsistencies, and automate the diagnosis and repair of misconfigurations.

ACKNOWLEDGMENT

This work is supported by National Key R&D Program of China (2017YFB0801703) and the Key Research and Development Program of Zhejiang Province (2018C01088).

REFERENCES

[1] 2020. Authentication Policy Origins JWT - Internal vs Public Access - Discuss Istio. (2020). https://bit.ly/3eP3KyQ Accessed on 2020-6-20.

[2] 2020. JWT authentication on specific method - Discuss Istio. (2020). https://bit.ly/3jttHrj Accessed on 2020-6-20.

[3] 2020. Kiali. (2020). https://kiali.io/ Accessed on 2020-6-20.

[4] 2020. kube-bench. (2020). https://bit.ly/2YOd2pX Accessed on 2020-6-20.

[5] 2020. Online Boutique. (2020). https://bit.ly/3ilMeVI Accessed on 2020-6-20.

[6] Kai Bu, Xitao Wen, Bo Yang, Yan Chen, Li Erran Li, and Xiaolin Chen. 2016. Is every flow on the right track?: Inspect SDN forwarding with RuleScope. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.

[7] Trail of Bits and Atredis Partners. 2019. Kubernetes Security Audit. (2019). https://bit.ly/2BUWqUw Accessed on 2020-6-20.

[8] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. 241–252.