MeshTest: End-to-End Testing for Service Mesh Traffic Management

Naiqian Zheng Tianshuo Qiao Xuanzhe Liu Xin Jin
School of Computer Science, Peking University

Abstract

We present MeshTest, the first end-to-end testing framework for traffic management of service mesh. The key idea of MeshTest is to automatically generate input configurations with end-to-end semantics, and then create real test request suites on each input. There are two technical challenges. First, the input space of service mesh configurations is large and complex. The input configurations should be carefully orchestrated to form end-to-end service flow paths. Second, the abstract output network behavior cannot be directly checked for correctness, and we need to generate a set of real requests that are capable of checking possible behaviors. To address these challenges, we model the service flows of traffic management in service mesh, and propose a novel *Service Flow Exploration* technique to enumerate all possible configuration resources and interactions between them in the input configuration. We design and implement MeshTest, which contains an automatic input configuration generator based on *Service Flow Exploration* and a *Service Mesh Oracle* which leverages formal methods to generate test request suites. MeshTest has found 23 *new* bugs (19 confirmed and 10 fixed) in two popular service mesh systems, Istio and Linkerd.

1 Introduction

Cloud computing systems are growing with larger scale and more complex functionalities. Many cloud computing systems choose to be architected in a microservice-alike [1] manner, with each collection of microservices performing some discrete business functions. Service mesh systems are designed to process network requests and manage the communication between microservices. The widely-used open-source and commercial service mesh systems, such as Istio [2], Linkerd [3], Canal Mesh [4] and AWS App Mesh [5], provide rich traffic management functionalities for service traffic. Traffic management decides how service requests are transmitted through the service mesh system, and establishes the foundation for other functionalities. It provides declarative API for developers to define service routing, load balancing, A/B testing [6], and access control with just a few lines of configurations. Service meshes have been widely-used in industry [7], and there are also academic proposals such as application-defined networks [8, 9] to improve performance and simplify management of service meshes.

The rapid evolution and increasingly popular use of service mesh systems have raised higher demands on their reliability. Reliable traffic management is critical in service meshes, as it

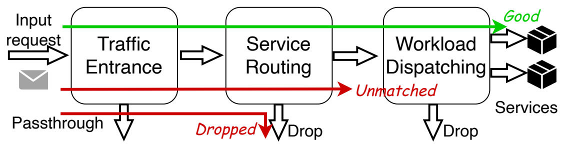


Figure 1: Network request processing stages in service mesh traffic management.

directly controls the service traffic and thus affects the correctness and performance of the entire application. The complexity in both implementation and functionality of service mesh traffic management makes it difficult to systematically test or verify. For example, Istio [2], one of the most widely-used service mesh systems, contains over 1,000 components and 3×10^5 lines of code. It provides more than 10 custom resources with extensive configuration options to define traffic management policies. Each custom resource is able to define a specific network function, and an actual service mesh configuration contains multiple custom resources. These resources are orchestrated to describe full end-to-end network functions, including traffic entrance, service routing, and workload dispatching (shown in Figure 1). The interactions between resources create more complex functionalities, such as priority competition on resources with the same matching conditions.

Figure 2 shows a real bug that our technique detects in Istio [10]. When two `service entries` (a custom resource in Istio to define service entrance rules) are defined with the same hostname but different ports and workloads, one of them will be unreachable. This bug is caused by the incorrect implementation in service entrance, which fails to merge the two `service entries` correctly and makes the service listening on port 9080 unreachable. Similar to this bug, traffic management bugs lead to severe issues, such as service unavailability and incorrect service routing, which subsequently degrade the entire application. These bugs escape from existing testing suites because existing service mesh systems rely most on unit tests which can not check traffic management correctness end to end. Some systems include a few end-to-end tests, but they only check basic functionalities and do not cover complex resource interactions.

Creating end-to-end tests for traffic management of service mesh is not trivial, since the test input should be carefully orchestrated to form service flow paths from the sources to the destinations. Figure 1 shows three stages of request processing in the traffic management of service mesh systems. Each stage receives and forwards requests based on the fields and

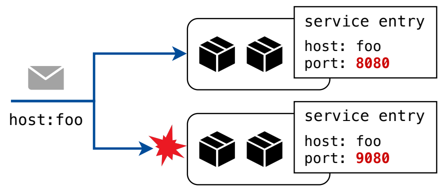


Figure 2: A bug [10] discovered by MeshTest. When two `service entries` are defined on the same host but different ports and workloads, one of them will be unreachable.

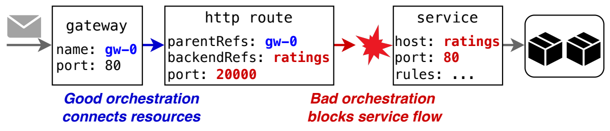


Figure 3: An example configuration snippet of Linkerd [3]. It demonstrates that resource orchestration is necessary for end-to-end test cases; otherwise, it would break the service flow and make the cases not end-to-end.

metadata of the request (*e.g.*, hostname, destination). An end-to-end test case should build up service flow paths through these stages, while a basic test input may drop the request in some stages or fail to hand over between stages. To orchestrate resources and forward the service flow from one resource to another, developers need to set up connectors fields in the entry and exit of each resource to specify what kind of service flow to accept and where to forward. Figure 3 shows a good and a bad example of service flow forwarding between different resources based on a configuration snippet of Linkerd [3]. In the good case (highlighted in blue), the `gateway` hands over the service request to the `http route` via the matching connector fields. In the bad case (highlighted in red), the `http route` fails to forward the request to the service due to a mismatch in port, and blocks the service flow, causing the test case to be not end-to-end. Notably, the real end-to-end test cases can be much more complex than this example, as the types of resources and their connections are more diverse, allowing service flow splitting and merging in various ways.

We seek for a systematic approach to automatically perform end-to-end testing for traffic management of service mesh. Existing techniques like symbolic execution [11] and fuzzing [12] do not model service flows, and cannot be directly used to generate end-to-end input configurations for service mesh systems because of state explosion and low validation pass rates. Besides, the automatic testing tool requires an automatic testing oracle to check the correctness on each input configuration, which existing tools cannot achieve.

Our solution. In this paper, we present MeshTest, the first automatic end-to-end testing framework for traffic management of service mesh. MeshTest automatically generates end-to-end input configurations based on an explicit model on service flows and a novel *Service Flow Exploration* technique. MeshTest's *Service Mesh Oracle* checks whether the service mesh correctly implements functionalities specified in the input configuration by sending real network requests. MeshTest is highly usable. It does not require hypotheses about vulnerable regions in the code nor implementation details. MeshTest is not tied to any specific service mesh system, and can be retrofitted among different systems by its generic API.

MeshTest is a service flow centric approach. It gradually models service flow at a finer granularity on a smaller scale in four stages: service flow exploration, service flow filling, fine-grained service flow modeling and service flow execution. The first two stages compose the input generator, which automatically creates end-to-end input configurations. The last two stages compose the *Service Mesh Oracle*, which automatically generates real network requests to check whether the service mesh system behaves correctly under the input configuration. These stages are designed to be loose-coupled so that they can work individually and can be extended to other tools.

The key insight of MeshTest's input generator is that in an end-to-end input configuration, the resources are orchestrated by end-to-end service flows. Specifically, MeshTest models the input space of end-to-end configurations as *Service Flow Skeletons* (representing interactions among configuration resources) and *Service Flow Bodies* (representing the detailed rules inside each configuration resource). The MeshTest input generator firstly conducts a domain-specific *Service Flow Exploration* technique to create service flow skeletons (§ 4.1), which depict end-to-end layouts of resource orchestration. *Service Flow Exploration* additionally objectives testing vulnerable resource interactions, comprehensively covering all possible resource interactions in the service flow skeleton suites. Then, MeshTest fills in the service flow bodies based on the skeleton with fuzzing-based domain-specific techniques to generate ready-to-use end-to-end input configurations with various network configuration options (§ 4.2).

The *Service Mesh Oracle* in MeshTest is a powerful testing oracle to check whether the service mesh system correctly implements the input configuration. The key idea of *Service Mesh Oracle* is to select a comprehensive set of requests to represent possible service traffic based on a domain specific model, and then check whether the results are as expected. To do so, *Service Mesh Oracle* first supplements the service flow model with more fine-grained and accurate rules (§ 4.3), and transforms the input configuration into a control flow graph (CFG) which is capable of inferring the correct effects on arbitrary requests. Then, *Service Mesh Oracle* conducts symbolic execution on the control flow graph, and generates a set of test requests where each request represents a unique path in the CFG, thereby representing a specific behavior of the input configuration (§ 4.4). These requests are then sent to and captured from the service mesh system hosted in a testbed to check whether the system behaves as expected under the input configuration. Besides, *Service Mesh Oracle* also checks negative cases, such as rejecting invalid inputs, and inspects whether the system raises exceptions.

Key results. We implement MeshTest for service mesh systems. MeshTest requires no expert knowledge on the implementation code base. MeshTest's input generator can effectively enumerate possible layouts of configuration resources orchestrated by service flows, and generate corresponding end-to-end input configurations. MeshTest's *Service Mesh Oracle* can automatically generate a real network request suite which is comprehensive to represent arbitrary service traffic and check behaviors of the service mesh system under the input configuration. We evaluate MeshTest on two most widely-used service mesh systems, Istio [2] and Linkerd [3]. To date, MeshTest has detected 23 *new* bugs that have not been reported previously. 19 out of these 23 bugs have been already confirmed and 10 have been fixed by developers. Notably, a large number of these bugs occur due to interactions between resources, leading to deep semantic violations that are difficult to detect using existing approaches. MeshTest can generate 2500 end-to-end input configurations per second, and check correctness with an average of 29 varying network requests for each input configuration.

Contributions. The paper makes three main contributions.

- We propose a service flow centric approach to model the end-to-end semantics of traffic management of service mesh in gradually finer granularity, and design a novel *Service Flow Exploration* technique to create input configurations with end-to-end service flows.
- We design and implement MeshTest, the first automatic end-to-end testing technique for traffic management of service mesh. MeshTest consists of an input generator empowered by *Service Flow Exploration* and a *Service Mesh Oracle* based on formal methods.
- MeshTest has already improved the quality of two most widely-used service mesh systems by finding 23 *new* bugs. The code of MeshTest is open-source and publicly available `https://github.com/pkusys/meshtest`.

2 Background and Motivation

2.1 Service Mesh Systems

Traffic management of service mesh. Service mesh systems provide a set of network functions to manage and control network traffic in a microservice architecture. Traffic management is the core and most important function, which includes fine-grained service routing, load balancing, and advanced traffic control such as A/B testing. The input of the service mesh is a high-level configuration file that describes the desired network behaviors with custom resources, such as service routing rules and workload discovery. The configuration file (usually written in YAML or JSON) is declarative and highly-abstracted, *i.e.* it only specifies the desired network behaviors, but not how to implement them. The main goal and challenge of service meshes is to correctly implement the abstract network behavior specifications into concrete request processing logics on each nodes.

Interactions between input configuration resources. Although each configuration resource specifies an individual network function, the end-to-end network traffic transmission requires orchestrated interactions between these resources to build end-to-end service flows. The resources define different network functions in different request processing stages, and interact with other resources via connector fields. The blue resource orchestration in Figure 3 allows the service traffic to be forwarded from traffic entrance rules to service routing rules. In contrast, the red resource interaction blocks the service flow, and drops the service request before reaching the destination workload. Besides the orchestration, competition and conflicts also appear between configuration resources. For example, two resources on routing can share the same host key but conflicting routing behaviors. In this case, the service mesh system has to resolve the priority competition and arrange the correct routing rule for arbitrary service traffic. This problem orients from the declarative nature of the input configuration, which makes resource interactions implicit.

Abstract service mesh output. The goal of service mesh is to correctly implement desired network functions specified in the input configuration for arbitrary service traffic. The output of service mesh is the abstract traffic processing behaviors in the system, which are not directly observable or measurable. Judging the correctness of the output has to rely on sending and capturing real network requests to infer the abstract traffic processing behaviors. Thus, a reliable testing oracle has to select a comprehensive network request suite which is limited in size but capable of representing various service traffic in the service mesh system. In particular, it should cover all possible individual network functions and their complicated interactions specified in the input configuration.

2.2 Testing Service Mesh Systems

The lack of existing end-to-end tests. Most popular service mesh systems already have a mature ecosystem, including unit tests and continuous integrations, but they usually have only few end-to-end tests. For example, Istio [2] has 10891 unit tests and has already been widely-used in production. However, it only has 168 end-to-end tests and 217 assertions. Most of these end-to-end tests are simple and the assertions are basic. They do not reason about complex network behaviors caused by the interactions of resources. Another popular service mesh system, Linkerd [3], also has only 31 simple end-to-end tests. The lack of high-quality end-to-end tests is a common problem in traffic management of service mesh systems, mainly due to the difficulties of creating test inputs with end-to-end semantics and the challenges of creating a strong testing oracle. On one hand, the end-to-end service flow path costs a lot of manual work to generate configuration resources and connect them with proper connector fields to form reasonable service flow paths. On the other hand, the extensive functionality combinations raise the complexity of determining the vulnerable input patterns and code regions. Thus, the lack of end-to-end tests reveals an urgent need to develop a comprehensive end-to-end testing framework for traffic management of service mesh, more specifically, to create more complex end-to-end input configurations and stronger correctness checkers.

Difficulties on applying existing techniques. Existing testing techniques, such as fuzzing [12, 13], symbolic execution [11], cannot be directly applied to service mesh systems. The main reason is that the input configuration is composed of a set of configuration resources, which are very large and have to be carefully orchestrated to form end-to-end service flow paths. Symbolic execution suffers from the path explosion problem, since each resource has a large number of detailed rule options and the combination of these resources explodes the input space exponentially. Specifically, `virtual service` includes more than one million of different configurations, and the number explodes when combined with other resources. Fuzzing does not model the service flows, so the generated resources are usually isolated and do not form end-to-end service flow paths. Another solution is to fuzz individual resources and then combine them by setting connector fields. On one hand, it is not easy to decide what resources are generated and how to connect them. On the other hand, the modified resources may not pass the configuration validation for violating constraints.

Recent works on reliability of Kubernetes controllers [14] and operators [15] cannot be directly applied to service mesh. The main reason is that these works rely on the state-centric property of Kubernetes, while service mesh traffic management is service flow centric. Specifically, state reconciliation is not the main concern in service mesh, since the network policies are completely recalculated on configuration changes. Besides, the input generation technique proposed by Acto [15] is not suitable for traffic management of service mesh, since many of the core resources in service mesh systems do not have a direct mapping to Kubernetes resources.

2.3 Challenges

Our goal is to develop a comprehensive end-to-end testing framework for service mesh traffic management. It should (1) automatically generate orchestrated input configurations with end-to-end service flow paths and (2) automatically checks the correctness of the abstract network behaviors by sending and checking real network requests. We identify two main challenges:

Challenge 1: End-to-end input generation. The first challenge is to create input configurations that have end-to-end semantics. More specifically, the resources in the input configuration have to be orchestrated and carefully connected to form complete service flow paths. Additionally, the input generator should be able to handle complex interactions and interleaving between resources, and strive to create complex combinations of configuration resources, such as service flow priority competition.

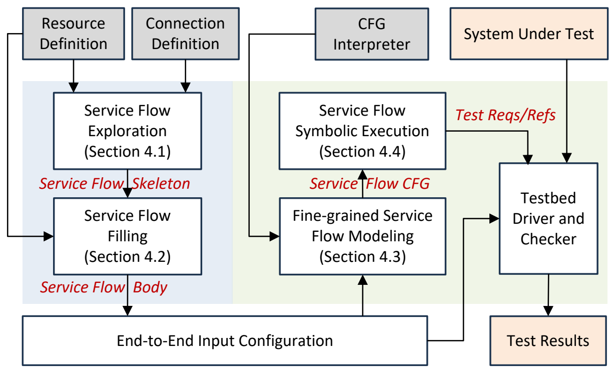

Resource Definition
Connection Definition
CFG Interpreter
System Under Test
Service Flow Exploration (Section 4.1)
Service Flow Symbolic Execution (Section 4.4)
Test Reqs/Refs
Service Flow Skeleton
Service Flow CFG
Testbed Driver and Checker
Service Flow Filling (Section 4.2)
Fine-grained Service Flow Modeling (Section 4.3)
Service Flow Body
End-to-End Input Configuration
Test Results

Figure 4: The workflow of MeshTest. The manual parts are gray, the automatic parts are white, and the system input and output are orange.

Challenge 2: Oracle for complex end-to-end inputs. The second challenge is to create a strong testing oracle that can judge whether the service mesh system correctly implements the input network configuration. The testing oracle has to create a limited but comprehensive network request suite which is able to represent various service traffic in the service mesh system. It also has to infer the expected output of arbitrary input requests as a reference to compare with the actual output.

3 Overview

MeshTest centers on service flows, which indicate how the service traffic flows through network functions specified in configuration resources. Specifically, MeshTest explicitly models the service flow in different granularity levels, in order to balance the need of covering larger input space and judging the correctness of service mesh behaviors. First, MeshTest's input generator models the input space of end-to-end configurations as service flow skeletons and service flow bodies, respectively representing the high-level combination layouts and detailed rules of resources. Second, MeshTest's oracle generates a comprehensive set of real requests based on a precise model of the service flow, and checks the correctness of the target system.

3.1 Workflow

Figure 4 shows the full workflow of MeshTest. MeshTest gradually creates and completes test cases in four stages: (1) it generates service flow skeletons by leveraging a domain specific *Service Flow Exploration* technique, indicating the high-level layout of service flows; (2) it fills service flow bodies with detailed rules and options and creates ready-to-use input configurations; (3) it automatically models the input configuration into a fine-grained service flow CFG; and (4) it checks the service mesh system with real requests generated. The first two stages focus on the input generation, and the last two stages focus on the output checking.

Through the four stages, the service flow modeling is gradually refined in finer granularity, and corresponds to a smaller scope. Each service flow skeleton specifies a high-level combination layout of resources, but does not contain detailed rules inside resources. Each service flow body (and its equivalent input configuration) is one of the possible fuzzing results of the service flow skeleton, and fills detailed rules inside resources. Each service flow CFG is a formal model for one input configuration, and contains precise priority selection and any other implicit effects of configuration. Each test request corresponds to a specific service flow path in the CFG, and is used to check a detailed effect. The less detailed modeling in MeshTest's input generator focuses on service flow layouts and reduces the complexity of input space without loss of generality. The fine-grained modeling in *Service Mesh Oracle* provides better correctness checking ability and covers more subtle explicit and implicit effects.

Usage. To use MeshTest, one needs to provide (1) the definition of the configuration resources used in the service flow, including constraints and the range of values, (2) connection rules between different resources, such as possible predecessor and successor resource types, and (3) an interpreter to automatically convert the input configuration to its equivalent service flow CFG, which includes system-specific detailed rules such as priority selection and implicit routes. On this basis, MeshTest automatically generates ready-to-use input configurations, creates a comprehensive service request suite, and applies the test on a testbed environment. Figure 4 illustrates the various components, system inputs, and outputs of MeshTest, where the manual parts are indicated in gray and the automatic parts are shown in white. The definition of the configuration resources and connection rules can be directly obtained from the documentations of the service mesh system. The interpreter requires users to utilize system-specific knowledge to develop and provide to MeshTest, thus requiring some manual efforts when updating the target service mesh or applying MeshTest to a new service mesh system. MeshTest provides a user-friendly interface, facilitating generalization across various service mesh implementations. These interfaces include configuration template definition and connection rule matrix for the input generator, and domain specific primitives used in CFG and symbolic execution for the testing oracle. In our practice, adapting MeshTest to another service mesh system costs less than two person-weeks. After the adaptation, MeshTest can work automatically and continuously until the input constraints or rules change.

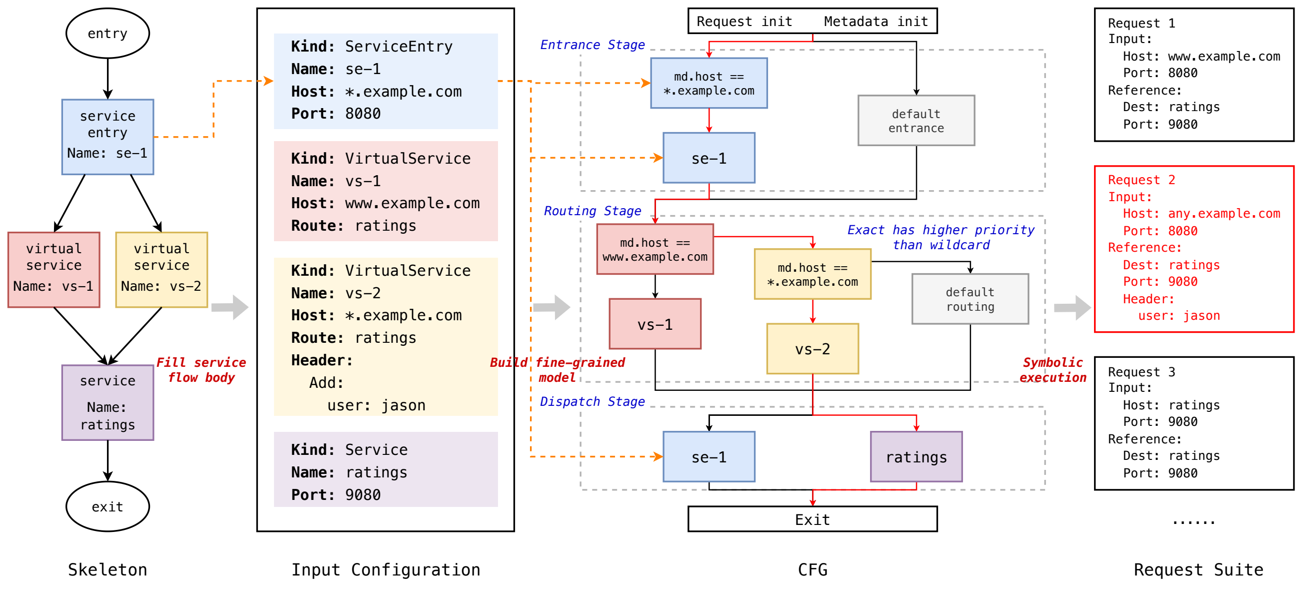
3.2 Technique

Collecting service flow skeletons. The goal of this stage is to generate service flow skeletons which depict a high-level layout of service flows. We model a service flow skeleton as a directed acyclic graph (DAG). Each node represents a configuration resource, and each edge represents a connection between two resources. To ensure the end-to-end service flow, the service flow skeleton contains end-to-end service flow paths for all internal resources, making it ready for end-to-end testing. To do so, we leverage a domain specific *Service Flow Exploration* technique to systematically build service flow skeletons with end-to-end service flows. The *Service Flow Exploration* starts from a skeleton seed with resource interactions, and gradually connects more resources on the predecessor side and successor side. In this way, the skeleton seed is extended to larger size, and eventually connects the start and the end points to form a complete end-to-end service flow skeleton. Additionally, we heuristically explore all possible interactions of three typical types for any two resources to be the skeleton seed, in order to check the service meshes on more service flow orchestrations.

Filling service flow bodies. The goal of this stage is to fill detailed rules and options for each resources in the service flow skeleton, and then create ready-to-use input configurations. The service flow body describes (1) the detailed rules and options for each resource in the service flow skeleton, and (2) which pair of fields connects the resources. The main challenge of this stage is how to realize the connectivity defined in the service flow skeleton into real field values. To do so, MeshTest leverages a two-stage approach in which the first stage fills the connector fields to ensure connectivity, and the second stage extends the configuration with more detailed rules and options with fuzzing. As shown in the second part of Figure 4, one abstract service flow skeleton is filled with various detailed options, and becomes a set of corresponding input configurations. The configurations realize service flows designed in the skeleton, and cover more options on non-connector fields.

Fine-grained service flow modeling. The goal of this stage is to build a precise fine-grained model for each input configuration. The model describes the expected behaviors of arbitrary service request under the input configuration. The model depicts the fine-grained control flow of processing the service requests in the service mesh, and considers all explicit and implicit effects of the input configuration. Compared to the service flow skeleton and body, the fine granularity is mainly reflected in considering the complete role of resources, the priority competition between resources, and other implicit intrinsic rules in the service mesh. The design of the structure of the service flow CFG is based on the intrinsic three-stage structure of the service processing model (traffic entrance, service routing, and workload dispatching). We embed each resource's effects to these three stages, and resolve the priority competition and implicit routes carefully.

Checking by real requests. The goal of this stage is to check if the service mesh system correctly realizes the input configuration by sending and checking real service requests. The service request suite is a comprehensive set of real service requests, which is capable to represent arbitrary possible request. Specifically, it should cover all individual and combined rules



```
Kind: ServiceEntry
Name: se-1
Host: *.example.com
Port: 8080

Kind: VirtualService
Name: vs-1
Host: www.example.com
Route: ratings

Kind: VirtualService
Name: vs-2
Host: *.example.com
Route: ratings
Header:
  Add:
    user: jason

Kind: Service
Name: ratings
Port: 9080
```

```
Request 1
Input:
  Host: www.example.com
  Port: 8080
Reference:
  Dest: ratings
  Port: 9080

Request 2
Input:
  Host: any.example.com
  Port: 8080
Reference:
  Dest: ratings
  Port: 9080
  Header:
    user: jason

Request 3
Input:
  Host: ratings
  Port: 9080
Reference:
  Dest: ratings
  Port: 9080

......
```

Figure 5: An illustrative example on the creation and evolvement of test cases for Istio [2]. The process starts from a service flow skeleton, evolves to a detailed input configuration, models the full effects to a CFG, and finally composes a real test request suite.

specified in the input configuration, and infer the expected outputs of those requests. To get a comprehensive test request suite, MeshTest leverages symbolic execution and traverses all possible paths on the service flow CFG. Each service flow path in the CFG corresponds to a specific test request, and further corresponds to a specific rule in the input configuration. After that, these test requests are sent to and captured from the testbed environment, and compared with the expected results.

4 Design and Implementation

This section explains the design of MeshTest and how we implement it. Figure 5 shows an illustrative example of the whole test campaign. It shows how a test case is created and evolved in the testing workflow. Specifically, we explain how MeshTest explores the resource combinations to orchestrate end-to-end service flow skeletons(§ 4.1), fills all fields to compose ready-to-use end-to-end input configurations (§ 4.2), models full fine-grained effects of the configuration (§ 4.3), and tests the target system with real service requests (§ 4.4).

4.1 Service Flow Exploration

Generating end-to-end service flows to guide the orchestration of resources is the first step of the workflow and the key to end-to-end test input generation. As shown in the first part of Figure 5, the service flow skeleton depicts how the service flow passes through resources. Further, it serves as a backbone indicating the resource orchestration structure. Service flow skeletons are high-level and not ready for system input, since they do not contain detailed rules and options. To find all possible service flow structures, MeshTest models and then explores the service flow skeletons with a domain specific *Service Flow Exploration* technique.

Service flow model. To get a formal definition, we model the resources as nodes, and the service flow between resources as directed edges. Further, we model the service flow skeleton as a directed acyclic graph (DAG) $G = (V, E)$. Each resource is of one resource type $R(v) \in \{R_0, R_1, \cdots, R_n\}$. The connectability between two kinds of resources is modeled by an adjacency matrix $\mathcal{A} = [A_{ij}]$. The acyclic property of the graph is intrinsic in the service mesh traffic management, because any request has to be processed in bounded steps. We model the end-to-end service flow path as a sequence of resources connected by edges, which starts from the start point and ends at the end point. We define that the service flow skeleton does not contain any dangling resources, *i.e.* each resource is covered by at least one end-to-end service flow path.

Generating service flow skeletons. The goal of *Service Flow Exploration* is to generate end-to-end service flow skeletons which depict the layouts of resource orchestration. The key insight of *Service Flow Exploration* is to first explore resource interactions as skeleton seeds and then explore the end-to-end service flow paths traversing the seeds to compose end-to-end service flow skeletons. The skeleton seed describes the resource interactions used in the service flow skeleton. Other resources are orchestrated to establish end-to-end service flow paths that traverse through the skeleton seed, thereby enabling the composition of complete service flow skeletons. The inputs of *Service Flow Exploration* are the types of resources $\mathcal{R}$ and the adjacent matrix $\mathcal{A}$ describing the connectability of resources. They can be inferred from the documentations and specifications of traffic management of service mesh. The output is a set of end-to-end service flow skeletons showing orchestrations and interactions of resources.

Algorithm 1 *Service Flow Exploration*

```
Inputs:
  – $\mathcal{R} = \{R_0, \cdots, R_n\}$: resources for traffic management of service mesh
  – $\mathcal{A} = [A_{ij}]$: adjacency matrix for resource connection compatibility
Outputs:
  – $\mathcal{S}keleton$: end-to-end service flow skeletons
 1: function ExploreSkeletonSeeds($\mathcal{R}$, $\mathcal{A}$)
 2:     // Enumerate resource interactions as seeds
 3:     seeds ← ∅
 4:     for res1, res2 ∈ $\mathcal{R}$ do
 5:         if $\mathcal{A}_{res1,res2}$ then
 6:             seeds.add(Connect(res1, res2))
 7:         if $\exists res3\ s.t.\ \mathcal{A}_{res3,res1} \cap \mathcal{A}_{res3,res2}$ then
 8:             seeds.add(Split(res3, res1, res2))
 9:         if $\exists res3\ s.t.\ \mathcal{A}_{res1,res3} \cap \mathcal{A}_{res2,res3}$ then
10:             seeds.add(Merge(res3, res1, res2))
11:     return seeds
12: function ServiceFlowExploration($\mathcal{R}$, $\mathcal{A}$)
13:     skeletonSeeds ← ExploreSkeletonSeeds($\mathcal{R}$, $\mathcal{A}$)
14:     $\mathcal{S}keleton$ ← ∅
15:     for seed ∈ skeletonSeeds do
16:         partialSkeleton ← seed
17:         seedResourceSet ← seed.getResourceSet()
18:         queue ← Queue(seedResourceSet) // init with seed resources
19:         while queue ≠ ∅ do
20:             res ← queue.pop()
21:             if res ≠ $\mathcal{R}_{entry}$ and res has no predecessor then
22:                 pred ← GetPossiblePred(res, $\mathcal{R}$, $\mathcal{A}$)
23:                 partialSkeleton.connect(pred, res)
24:                 queue.push(pred)
25:             if res ≠ $\mathcal{R}_{exit}$ and res has no successor then
26:                 succ ← GetPossibleSucc(res, $\mathcal{R}$, $\mathcal{A}$)
27:                 partialSkeleton.connect(res, succ)
28:                 queue.push(succ)
29:         $\mathcal{S}keleton$.add(partialSkeleton)
30:     return $\mathcal{S}keleton$
```

Algorithm 1 shows the *Service Flow Exploration* algorithm, which consists of two steps: exploration of skeleton seeds and exploration of end-to-end service flow paths. First, *Service Flow Exploration* explores the skeleton seeds by enumerating all possible interactions between resources (line 1-11). We limit the interactions to be pairwise, since the documentation and implementation of traffic management of service mesh typically split large interactions into several pairwise interactions. *Service Flow Exploration* explores three types of interactions: direct connection (line 5), service flow splitting (line 7), and service flow merging (line 9), since the direct connection is the most common interaction in traffic management of service mesh, and the splitting and merging interactions challenge correctness of complex priority competition and conflict resolution. Notably, the interactions are not limited in scale and type, and they can be extended to more complex interactions in practice, such as increasing the number of interleaving resources or adding more complex interactions.

Second, *Service Flow Exploration* explores end-to-end service flow paths for each skeleton seed to construct end-to-end service flow skeletons (line 16-29). The key idea of the exploration is to extend the service flow path from the skeleton seed to the designated entry and the exit points. For the entry side, the algorithm initializes a queue with seed resources (line 18), recording resources without predecessors. It then iteratively pops a resource from the queue, appending any possible predecessor if the resource lacks one (lines 21-23). This predecessor is subsequently re-enqueued for further exploration. This iterative process drives the service flow path towards the entry point, continuing until every service flow path includes the entry point (line 21). The exit side is explored analogously, ensuring that the service flow paths remain connected from end to end. The correctness of the algorithm and its property of not falling into an infinite loop are guaranteed by an intrinsic property of traffic management for service mesh: any type of resource has an end-to-end service flow path that passes through it, and the length of the path is bounded. Thus, the algorithm will not fall into a loop, since a service flow with unbounded length is not permitted in the system. Importantly, the algorithm does not seek to enumerate all possible service flow skeletons for a given skeleton seed, since the interaction among resources outside the initial seed can be later considered as new seeds to construct new service flow skeletons. *Service Flow Exploration* prioritizes maintaining simplicity in service flow skeletons to facilitate bug determination and localization.

4.2 Filling Service Flow Bodies

Filling service flow bodies is the second stage of the testing workflow. It extends service flow skeletons to complete input configurations which are ready for the system input and following testing steps. Compared with the service flow skeleton, filling the service flow bodies requires more considerations on the configuration formats and detailed constraints. Besides, MeshTest needs to realize the connectivity described in the abstract service flow skeleton in concrete service mesh configurations. To do so, MeshTest leverages a two-step approach to fill the service flow bodies. The first step fills the connector fields to ensure the connectivity of resources. The second step extends the configuration with more detailed rules and options.

The main challenge of the first step is how to connect resources as specified in the service flow skeleton. The service flow skeleton only describes the connectivity of resources, but does not specify the specific connector fields. MeshTest automatically sets connector fields to realize the connectivity of resources. For example, MeshTest connects a routing resource and a workload dispatching resource by setting related values on connector fields in both resources. Further in some cases, connecting adjacent resource pairs does not guarantee the connectivity of the whole service flow path. Taking a practical Istio test case as an example, the service flow skeleton specifies a service flow path that sequentially passes through routing rules, workload dispatching rules, and service definitions. Since the routing rule specifies the host and port fields, the service definition has to specify the same host and port fields though they are not directly connected.

To ensure the connectivity of the whole service flow path, MeshTest tracks the core keys of service flow (such as host and port), propagates among resources from the entry, and fills the connector fields in a topological order.

When the connectivity is ensured, the second step fills other fields with a constraint-aware fuzzing technique. The constraints include ranges of values for each field (*e.g.*, the type of `StringMatch` has to be one of `EXACT`, `PREFIX`, `REGEX`) and field conflicting (*e.g.*, if `resolution` is set to `DNS`, host can not be wildcard). Besides, this step has to guarantee newly filled fields do not conflict with the connector fields. Moreover, we intentionally set some fields with invalid values (prohibited in the documents) or some special values (*e.g.*, empty string, wildcard domain) to challenge the robustness of the service mesh systems. To summarize, the service flow body filling transforms each abstract service flow skeleton into many concrete and detailed input configurations, meanwhile enriches the input configurations with more random options and rules.

4.3 Fine-grained Service Flow Model

The first two stages of the testing workflow focus on creating various end-to-end input configurations for the service mesh system. With the generated input configurations, the next two stages build a strong *Service Mesh Oracle* to check whether the service mesh system correctly realizes the network functions defined in the input configurations.

Gaps between skeletons and facts. Although first two stages are service flow centric, they do not build a precise and fine-grained model to reason about the accurate effects of the input configuration. The first three parts of Figure 5 show the gap between the service flow skeleton, input configuration and the real network behaviors. On one hand, the service flow skeleton does not model the full effects of resources. For example, in the service flow skeleton, the `se-1` is only used to define the traffic entrance, but in real service mesh systems the `service entry` also influences the dispatching stage as a supplementary endpoint provider. On the other hand, the service flow skeleton does not model fine-grained rules such as priority selection and default routes. For example shown in the third part of Figure 5, `vs-1` has higher priority than `vs-2` since exact matching has higher priority than wildcard matching. Another example is the grey blocks, which represent the default routes on port 80.

Modeling fine-grained semantics with CFG. In order to provide a precise and fine-grained model for the whole semantics specified in the input configuration, MeshTest uses a control flow graph (CFG) to describe the processing logics of an arbitrary network request. The CFG is a directed acyclic graph (DAG), which is composed of `action` nodes and `predicate` nodes. Each `action` node represents setting a field or metadata in the network request. Each `predicate` node represents a condition that determines whether the following nodes adapt to the specific request. The CFG inherits intrinsic three-stage structure of service mesh systems: traffic entrance, service routing, and workload dispatching. Each resource acts as a subgraph in the CFG, and the priority selections are modeled as the priority comparisons between different subgraphs.

An example of the fine-grained service flow CFG is shown in the third column of Figure 5. MeshTest uses interpreters (provided by developers) to convert the input configurations into service flow CFGs. The interpreters vary with different service mesh systems, require domain specific knowledge, and cost manual efforts. Notably, the manual efforts are inevitable: on one hand, different service mesh systems have varying input APIs and specific rules, making it infeasible to develop a universal interpreter; on the other hand, the interpreter must accurately represent the functionalities of the service mesh, which is difficult to infer automatically from documentation or code.

MeshTest provides general CFG APIs to aid the users and reduce manual efforts in developing the interpreter. First, MeshTest provides a set of universal CFG primitives, which are common in all service mesh systems. The primitives include nodes and edges in the control flow graph, basic three-stage CFG backbone structure, and expressions and statements to describe the semantics of service mesh resources. Second, MeshTest provides universal abstraction of network requests and service mesh metadata, which is applicable to all service meshes. The network request abstraction includes host, headers and other contents in the TCP and HTTP request. The metadata depict the status of the request, such as source and destination. MeshTest also provides APIs to effectively encode hosts and URIs by splitting them into substrings and further encoding them to integers. Third, MeshTest provides common utility functions, which are frequently used in all service mesh systems. The utility functions include string matching, priority comparisons, default routes, and so on.

MeshTest also provides strategies and guides to build the interpreter. First, the interpreter should build up the backbone structure with its CFG API, set entry and exit nodes, and initialize request fields and metadata. Second, the interpreter needs to encode each resources in the input configuration to individual subgraphs. Each subgraph contains a `predicate` node as the entry guardian, and other following nodes for the detailed semantics of the resource. Third, the interpreter should connect the subgraphs into the holistic CFG, resolve priority competitions between different resources, and add default routes to the CFG. In the example of Figure 5, the `vs-1` and `vs-2` are in different priority levels due to different matching types. The interpreter also utilizes other detailed rules to resolve priority conflicts, such as resource sequence and resource name. After initial development, the interpreter might be incorrect due to bugs or misunderstandings. To address this issue, MeshTest adopts an iterative development approach, conducting tests on the interpreter and the target service mesh system simultaneously, ultimately obtaining an interpreter without false positives (§ 5.4).

With these APIs and strategies, developing and maintaining the interpreter for specific service mesh systems become much easier for the developers. In practice, we have developed interpreters for Istio and Linkerd, and the manual effort is less than two person-weeks. After that, the interpreter is reusable for further development and evolvement

4.4 Testing with Real Network Requests

Judging whether the service mesh system correctly realizes the input configuration is the final step of the testing workflow. The realization is highly abstracted network behavior and cannot be directly observed or compared. The only way is to send real network requests into the mesh network and judge the correctness of output requests. Because of the infinity of network requests, we have to select a finite set of network requests which is comprehensive enough to represent arbitrary network requests. In particular, the test request suite should cover all detailed rules and corner cases appeared with subtle interactions of resources. It should also produce reference results for each output request. After that, we design a test driver which is able to set up the testbed environment, apply the input configuration to the service mesh system, send the network requests, capture output requests, and check results with the expected behaviors.

To produce the test request suite, we leverage symbolic execution [11, 16] on the service flow CFG. The symbolic execution enumerates all types of network requests by exploring all possible paths in the CFG. During the execution, the symbolic execution engine tracks and updates the states and path conditions on each nodes. When the execution reaches a `predicate` node, it evaluates the condition with a SMT solver [17] and decides whether the path is reachable. When the execution reaches an `action` node, it updates the request fields and metadata based on the action. After exploring all paths in the CFG, the symbolic execution engine evaluates the corresponding input requests and reference results, and thus forms a comprehensive test request suite. Notably, since the input configurations obtained from the MeshTest's input generator are not large in scale, symbolic execution here will not encounter scalability issues (§ 5.3).

After that, the test driver automatically conducts tests on a real service mesh testbed. The input of each test campaign contains an input configuration and a set of network requests and reference results. The test driver converts the test suite into real requests by supplementing arbitrary values (unrestricted in the symbolic execution), attaching a test magic number to distinguish test requests and other network requests, and setting indexes to classify individual tests. It then sends the requests to the service mesh system, captures and parses the results, and compares the results with the reference results. The test driver also analyzes system status and logs to check for crashes and internal errors. Finally, the test driver reports the test results, identifying the failed input configurations and their corresponding network requests.

4.5 Implementation

We implement MeshTest with roughly 8300 lines of JAVA and Python code, among which 1400 lines for configuration resource descriptions (850 lines for Istio, 350 lines for Linkerd and 200 lines of shared codes), 1800 lines for the input generator, 3600 lines for CFG modeling and symbolic execution (1700 lines for Istio, 500 lines for Linkerd and 1400 lines of shared codes) and 1500 lines for the test driver. To enhance the generality and extensibility of MeshTest, we architect the system components in a loosely-coupled manner and delineate a suite of domain-specific interfaces within the shared codebase. Specifically, the shared code responsible for resource descriptions incorporates a parser tailored for domain-specific resource templates. Similarly, the shared code dedicated to CFG modeling furnishes a comprehensive set of interfaces for constructing the CFG, including nodes and edges, domain-specific expressions and constraints, and utility functions designed to encapsulate common semantic operations such as string matching and priority selection.

5 Evaluation

We conduct extensive evaluation on two most popular service mesh systems, Istio [2] and Linkerd [3], to demonstrate the effectiveness of MeshTest in finding new bugs and improving the reliability of service mesh systems. Our main evaluation results are summarized as follows:

- MeshTest has found 23 *new* bugs that escape from existing test cases. Among them, 19 have been confirmed and 10 has been fixed.
- MeshTest covers all properties specified by single resource and pairwise interactions between resources.
- MeshTest is efficient in end-to-end test case generation and oracle checking. The input generator produces 2500 end-to-end test cases per second. On average, the *Service Mesh Oracle* checks each test case using 29 distinct real-world requests within 15 seconds.
- MeshTest reports no false positives. Each test alarm during the test campaigns points to a violation of the specification of the target service mesh system.

5.1 Finding New Bugs

As shown in Table 1, MeshTest has found 23 previously unknown bugs in two most popular service mesh systems. We reported the bugs to developers and they confirmed 19 of them and fixed 10. Many of them stand in the critical path of request processing and can lead to severe failures, including incorrect state, load imbalance and security vulnerabilities. Figure 6 shows a real bug found by MeshTest, where the service request mistakenly skips the routing stage. This bug may lead to imbalanced load, and more seriously, it may disable security policies in the routing stage, and cause security vulnerabilities. It is noteworthy that the primary reasons for the majority of bugs being found in Istio [2] are twofold. On one hand, Istio

Index	Implementation	Bug Description	Status
1 [18]	Istio 1.19–1.21	Empty prefix in specific fields causes an internal error	Fixed
2 [19]	Istio 1.19–1.21/dev	Port 80 is not open by default when Istio gateways are not installed	Reported
3 [20]	Istio 1.19–1.21/dev	Traffic passthroughs cluster when `service entry endpoints` set to an internal IP	Reported
4 [21]	Istio 1.19–1.21/dev	`Service entry` with wildcard host makes traffic skip service routing	Confirmed
5 [22]	Istio 1.19–1.21/dev	`Service entry` defined on port 80 disables `virtual service`	Confirmed
6 [23]	Istio 1.22dev	Routing fails under multiple interleaved resources	Fixed
7 [24]	Istio 1.19–1.21/dev	Traffic is not dropped when port not matched in `virtual service`	Confirmed
8 [25]	Istio 1.19–1.21	`WithoutHeaders` matching fails without target header	Fixed
9 [26]	Istio 1.19–1.21	Delegation influences the priority between `virtual services`	Fixed
10 [27]	Istio 1.19–1.21/dev	Match conditions influence the choice of `virtual service` for gateway	Confirmed
11 [28]	Istio 1.19–1.21/dev	`Service` defined on port 80 disables `virtual service`	Reported
12 [29]	Istio 1.19–1.21	Update on `targetPort` does not trigger update on `EDS`	Fixed
13 [30]	Istio 1.19–1.21/dev	Wildcard matching fails on destination host	Reported
14 [10]	Istio 1.19–1.21	Collision between `service entries` with same host but different workloads	Fixed
15 [31]	Istio 1.19–1.21/dev	`EDS` missing for `service entry` defined on the same host as `service`	Confirmed
16 [32]	Istio 1.19–1.21/dev	`WorkloadSelector` takes effect at wrong place	Confirmed
17 [33]	Istio 1.19–1.21/dev	Header manipulation fails when the value is empty string	Confirmed
18 [34]	Istio 1.19–1.21/dev	Special headers are not ignored in match conditions	Confirmed
19 [35]	Istio 1.19/1.20	Header manipulation fails on pseudo headers	Fixed
20 [36]	Linkerd 2.14	Linkerd extension drives specific pods crash	Fixed
21 [37]	Linkerd 2.14	Routing error under rules with the same matching conditions	Fixed
22 [38]	Linkerd 2.14	Routing error under `http routes` bound on the same `gateway`	Fixed
23 [39]	Linkerd 2.14/dev	Incorrect `hostnames` effects	Confirmed

Table 1: Bugs found by MeshTest. Istio [2] and Linkerd [3] are two most widely-used service mesh systems. All bugs are newly discovered in supported versions or main branch (dev version). All bugs had escaped from existing test suites.

provides more complex and flexible features compared to Linkerd [3], which inherently raises the risk of encountering bugs. Istio exhibits significantly more issues than Linkerd. On the other hand, due to Istio's widespread adoption and rich feature set, MeshTest has implemented more configuration resources and more detailed service flow modeling specifically for Istio. Table 2 demonstrates categories and sources of these new bugs: entrance error, routing error, dispatching error, internal error and others.

Entrance error. These bugs indicate that requests are not captured correctly by the service mesh system. Entrance errors can cause the system to capture the wrong traffic or miss some traffic. Issue #2 in Table 1 shows a typical entrance error found by MeshTest. When no gateway is explicitly set, the default traffic entrance on port 80 is not added for internal requests to the virtual service. These bugs usually result in traffic loss and disable other functionalities.

Routing error. These bugs are rooted in the complex routing rules, including those specified by users and those generated by default. They are usually related to some complicated functionalities, such as priority competition and wildcard matching. Figure 7 describes a bug when two `virtual services` share the same host but have different routing behaviors. The correct priority should follow the creation order of the `virtual services`, but the `delegation` causes incorrect priority. Previous approaches cannot detect this bug, since it is triggered by specific combinations of routing rules. MeshTest generates test cases to cover the priority competition of routing rules with the *Service Flow Exploration* technique, and successfully detected this bug.

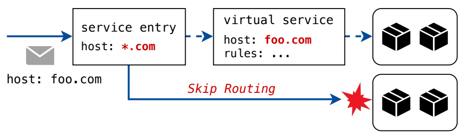


Figure 6: A real bug [21] in Istio discovered by MeshTest, where `service entry` with wildcard host makes the request skip the routing stage.

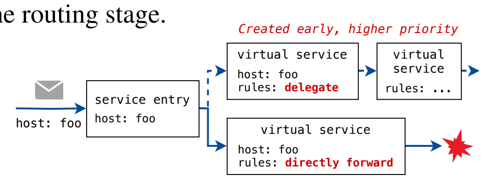


Figure 7: A real bug [26] in Istio discovered by MeshTest, where delegation causes incorrect priority.

Dispatching error. These bugs occur in dispatching the service requests to the concrete workloads. They are usually caused by incorrect assignment of the traffic to the workloads or wrong attributes of the services. Figure 2 shows a severe dispatching error, where two `service entries` with the same host collide with each other when they refer to different workloads. This bug is caused by the incorrect transformations from workloads to services, and it makes the system fail to dispatch traffic to partial workloads. It is confirmed by the developers and fixed in the next release.

Internal error. These bugs occur with an unhandled exception or an undesired state due to the absence of exception handling or internal conflicts. Although internal errors are not the primary target of MeshTest, we still detect some of them by simple but effective techniques such as setting invalid values for configuration properties, checking logs, and

	Istio	Linkerd	Total
Entrance error	1	0	1
Routing error	9	3	12
Dispatching error	5	0	5
Internal error	1	1	2
Others	3	0	3
Total	19	4	23

Table 2: Classification of the bugs found by MeshTest.

monitoring component liveness.

Others. Besides the above four types of bugs, MeshTest also finds some other bugs, such as manipulation failure and inconsistency between specification and implementation. We reported them to developers and received positive feedback.

5.2 Coverage

MeshTest achieves 100% coverage on the functionalities specified in single resource and functionalities indicated by the interactions between pairwise resources. Both coverages are critical for the correctness of service mesh traffic management, and are not fully covered with existing test suites. Even the coverage of functionalities in resource interactions is rarely considered in the design of test suites. MeshTest traverses all possible value types (exact, wildcard, etc.) for each resource field when filling the service flow bodies. MeshTest leverages the domain specific *Service Flow Exploration* technique to enumerate all possible pairwise interactions between resources to cover the functionalities rooted within. For the results, MeshTest detects many bugs related to resource interactions based on the full coverage, including Issue #4, Issue #5 and Issue #9 in Table 1. Besides, MeshTest also finds some bugs related to functionalities specified by single resource, such as Issue #8 in Table 1.

MeshTest improves code coverage by enumerating all possible resource interactions with the *Service Flow Exploration* technique and fuzzing detailed options with service body filling. We conduct code coverage analysis on the controller of Istio (`pilot-discovery`). The results show that: (1) MeshTest achieves 41.2% statement coverage for traffic management packages and 32.2% statement coverage for the entire controller; (2) MeshTest increases the statement coverage of existing test suites from 74.1% to 78.8% for traffic management packages and from 73.1% to 77.0% for the entire controller; and (3) *Service Flow Exploration* is effective in testing resource interactions, increasing the corresponding statement coverage from 70.9% to 79.4%.

5.3 Test Efficiency

The efficiency of MeshTest is evaluated in two aspects: how fast MeshTest can generate diverse end-to-end test inputs and how long it takes to execute and check each test case. We evaluate MeshTest in a virtual Kind [40] cluster on a lightweight host machine. On one hand, the input generator of MeshTest is efficient and effective. MeshTest leverages *Service Flow Exploration* to create service flow skeletons and fills service flow bodies to generate diverse end-to-end test cases with different service flows. MeshTest produces 2500 end-to-end test cases per second, and the cases need to be tested with an average of 29 different service requests. On the other hand, the *Service Mesh Oracle* checks whether the target system behaves consistently with the input configuration efficiently. For each test input, the *Service Mesh Oracle* builds the CFG, performs symbolic execution, and sends and receives requests in less than 15 seconds on average. As shown in Table 3, more than 99% of time is spent on environment setting, request sending and receiving, which is inevitable in the end-to-end testing. The environment setting time includes applying and deleting the end-to-end test configuration on the testbed environment, and cannot be reduced by testing tools. It is possible to set up multiple test environments in parallel to significantly improve the efficiency of *Service Mesh Oracle*.

	Set Env	**Build CFG**	**Symbolic Execution**	**Send/Recv Requests**	**Total**
Istio (Sidecar)	5.405	0.008	0.046	5.913	11.372
Istio (Ambient)	4.793	0.009	0.03	5.605	10.437
Linkerd	9.92	0.08	0.043	4.895	14.938

Table 3: Average time (in seconds) for each test input spent in *Service Mesh Oracle*.

5.4 False Positives

MeshTest prevents false positives under the assumptions of *Service Mesh Oracle* correctness. Each test alarm during the test campaign points to an inconsistency between the exprected behavior (specified in documentation) and actual behavior of the target system, and further indicates a bug or undefined behavior. To make sure the correctness of *Service Mesh Oracle*, we develop the service flow CFG model iteractively when we are detecting bugs in the target system. Specifically, we develop a basic service flow CFG model based on the documentation of the target system, and then we run the test campaigns to find violations. Then we determine whether the violations are caused by the bugs in the target system or the incorrectness of the CFG model. If the violations are caused by the CFG model, we refine the model and iteratively repeat the process until there are no more model errors. It is not feasible to develop a universal CFG model for all service mesh systems, since they have different functionalities and no standard specifications. We respectively developed the CFG model for Istio and Linkerd with less than two person-weeks of effort and finally made it reliable. We argue that although we cannot prove the correctness of the CFG model, it stops producing false positives quickly just after several iterations.

6 Discussion

Limitations and future work. MeshTest is a first step towards automatic end-to-end testing for traffic management of service mesh. Like other testing techniques, MeshTest is incomplete and it can miss bugs. MeshTest does not cover all possible input configuration space. Both the input generator and the testing oracle rely on the domain specific service flow model, which can not cover implementation specific details. This design aims to balance efficiency and coverage—it covers all pairwise interactions between resources and requests specified in the documentation. The results show that MeshTest can find bugs in real-world service mesh systems, but the coverage and scope of testing can be further improved. MeshTest relies on domain knowledge to provide the adjacency matrix in *Service Flow Exploration* and build the fine-grained CFG model for *Service Mesh Oracle*. Although most of the knowledge can be obtained from the documentation, there are some details that are not well documented and require experience as system users. The soundness of test results rely on the correctness of the service flow CFG model. MeshTest does not test service mesh performance, security, or network topologies. MeshTest does not aim to test the state reconciliation of the service mesh system.

Generality. The approach of MeshTest is not tied to the specific implementation of service mesh systems and can be generalized to other service mesh systems with different functionality models and configuration resource definitions. The generalization requires domain knowledge of service mesh for *Service Flow Exploration* and service flow CFG model construction. The generalization cost is inevitable due to the absence of a standard for service mesh systems. To reduce the effort of generalization, MeshTest introduces the universal principles of service mesh traffic management into the design and implementation, and provides a set of domain specific interfaces to facilitate generalization. These interfaces includes template resource definitions, domain specific expressions and constraints, and utility functions designed to encapsulate common semantic operations. Besides service mesh, the approach of MeshTest can be extended to other service flow centric systems with a high level declarative input specification for network behaviors, such as application defined networks [8] and software defined networks [41].

7 Related Work

Application defined networks and service meshes. Both service mesh [42] and application defined network (ADN) [8, 9] provide a way to decouple the network communication and the application logic, and releases the burden of development and maintenance. Besides industry practices [7], service mesh systems can be used to empower secure microservice applications [43], API managements [44], tracing systems [45], and so on [46]. Some works focus on the performance and efficiency of service mesh systems [9, 47, 48], which is orthogonal to our work. Application defined networks (ADN) [8] share the similar goal with service meshes, and it has more flexibility and better performance than service mesh [9]. ADN is also service flow centric and is expected to be a potential alternative of service meshes. We believe MeshTest can be easily retrofitted to ADN systems.

Reliability of microservice systems. There is rich literature on improving microservice systems by testing clusters [14, 49–52], RPC communication [53], system operators [15], and so on [54–64]. Among them, Sieve [14] and Acto [15] focus on the state reconciliation in the microservice systems. MeshTest focuses on the service flow and the end-to-end communication in service mesh systems, which is orthogonal to the state change reconciliation. The input generation technique in Acto [15] does not suit for service mesh systems since it relies on a mapping between the input resources and the system state resources. The traffic management of service mesh mainly focus on network communication rules and do not have a direct mapping to the system state.

Checking system correctness with formal techniques. Formal methods have been used to improve the system correctness in various domains, such as verification on system codes [65–75], system configuration verification [76–84]. Ucheck [85] is a tool to check whether a given invariant would hold based on modular microservice models. Ucheck [85] mainly focuses on the correctness of application logic, which is orthogonal to the communication correctness in service mesh systems. Model based testing builds a model of the system and creates test cases covering the model rather than the system code. It is lightweight and has been widely-used in testing large and complex systems [86–91]. *Service Mesh Oracle* is inspired by the art of model based testing, and builds a common model for service mesh functionalities.

8 Conclusion

We present MeshTest, an automatic end-to-end testing framework for traffic management of service mesh. MeshTest centers around an explicit service flow model, generates end-to-end test inputs with a domain specific *Service Flow Exploration* technique, and checks correctness with a *Service Mesh Oracle* which leverages formal methods to generate test request suites. MeshTest is proved to be practical for detecting 23 *new* bugs in real service mesh systems.

Acknowledgments. We thank our shepherd, T. S. Eugene Ng, and the anonymous reviewers for their valuable feedback. We thank all the Istio, Linkerd, and Kubernetes developers who engaged with us and reviewed our reports. This work was supported by the National Key Research and Development Program of China under the grant number 2022YFB4500700, the National Natural Science Foundation of China under the grant numbers 624B2007, 62172008 and 62325201, and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Xin Jin is the corresponding author. Naiqian Zheng, Tianshuo Qiao, Xuanzhe Liu, and Xin Jin are also affiliated with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.

[2] Istio Authors, "The istio service mesh." `https://istio.io`, 2024.

[3] Linkerd Authors, "The linkerd service mesh." `https://linkerd.io`, 2024.

[4] E. Song, Y. Song, C. Lu, T. Pan, S. Zhang, J. Lu, J. Zhao, X. Wang, X. Wu, M. Gao, *et al.*, "Canal mesh: A cloud-scale sidecar-free multi-tenant service mesh architecture," in *ACM SIGCOMM*, 2024.

[5] Amazon Web Services, "The aws app mesh." `https://aws.amazon.com/app-mesh`, 2024.

[6] H. Gui, Y. Xu, A. Bhasin, and J. Han, "Network a/b testing: From sampling to estimation," in *ACM Web Conference*, 2015.

[7] I. Authors, "Istio case studies." `https://istio.io/latest/about/case-studies/`, 2024.

[8] X. Zhu, W. Deng, B. Liu, J. Chen, Y. Wu, T. Anderson, A. Krishnamurthy, R. Mahajan, and D. Zhuo, "Application defined networks," in *ACM SIGCOMM HotNets Workshop*, 2023.

[9] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, *et al.*, "Dissecting overheads of service mesh sidecars," in *ACM SOCC*, 2023.

[10] "Collision between `service entries` with the same host." `https://github.com/istio/istio/issues/49550`, 2024.

[11] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, 1976.

[12] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[13] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *IEEE Symposium on Security and Privacy*, 2018.

[14] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic reliability testing for cluster management controllers," in *USENIX OSDI*, 2022.

[15] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu, "Acto: Automatic end-to-end testing for operation correctness of cloud system management," in *ACM SOSP*, 2023.

[16] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[17] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[18] "Empty prefix causes routing error." `https://github.com/istio/istio/issues/48534`, 2023.

[19] "Port 80 is not open by default when istio gateways are not installed." `https://github.com/istio/istio/issues/49991`, 2024.

[20] "Http traffic passthroughs cluster when `service entry endpoints` set to internal address." `https://github.com/istio/istio/issues/49430`, 2024.

[21] "Wildcard host in `service entry` results in routing error." `https://github.com/istio/istio/issues/49485`, 2024.

[22] "`Service entry` disables `virtual service` on port 80." `https://github.com/istio/istio/issues/49508`, 2024.

[23] "Routing error under multiple `virtual services` and `service entries`." `https://github.com/istio/istio/issues/49509`, 2024.

[24] "Traffic is not dropped when `virtual service HTTPMatch` is not matched by port." `https://github.com/istio/istio/issues/49516`, 2024.

[25] "`WithoutHeaders` denies packets without the target header." `https://github.com/istio/istio/issues/49537`, 2024.

[26] "Delegation influences which `virtual service` takes effect." `https://github.com/istio/istio/issues/49539`, 2024.

[27] "`HTTPMatch` affects which `virtual service` to be chosen for gateway." `https://github.com/istio/istio/issues/49588`, 2024.

[28] "Service defined on port 80 causes `virtual service` cannot capture traffic with specific host." `https://github.com/istio/istio/issues/49673`, 2024.

[29] "Update on `service entry` `targetPort` does not trigger update on eds." https://github.com/istio/istio/issues/49878, 2024.

[30] "Wildcard host in `service entry` cannot match the same host in the routing destination." https://github.com/istio/istio/issues/49482, 2024.

[31] "`CDS EDS` are missing for `service entry` defined on the same host and different port with another service." https://github.com/istio/istio/issues/50163, 2024.

[32] "`WorkloadSelector` takes effect in `service entry` with resolution `DNS`." https://github.com/istio/istio/issues/50164, 2024.

[33] "Header operation does not take effect if value is set to empty string." https://github.com/istio/istio/issues/49553, 2024.

[34] "Uri, scheme, method, authority are not ignored in `HTTPMatchRequest` header keys." https://github.com/istio/istio/issues/48555, 2023.

[35] "Inconsistent behaviors on pseudo headers." https://github.com/istio/istio/issues/48605, 2024.

[36] "`Viz` crashes `Pods` created by ingress controller." https://github.com/linkerd/linkerd2/issues/12344, 2024.

[37] "Different routing behavior between gateway and sidecar under several same `HTTPRouteMatch`." https://github.com/linkerd/linkerd2/issues/12267, 2024.

[38] "Inconsistent routing choice between gateway and sidecar under `http routes` with same parent," 2024. Github issue id omitted for anonymity.

[39] "Linkerd implementation of gateway api does not follow `GAMMA` specification." https://github.com/linkerd/linkerd2/issues/12295, 2024.

[40] Kubernetes Community, "Kind," 2024.

[41] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, 2014.

[42] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019.

[43] R. Chandramouli, Z. Butcher, *et al.*, "Building secure microservices-based applications using service-mesh architecture," *NIST Special Publication*, 2020.

[44] F. Hussain, W. Li, B. Noye, S. Sharieh, and A. Ferworn, "Intelligent service mesh framework for api security and management," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2019.

[45] D. Cha and Y. Kim, "Service mesh based distributed tracing system," in *2021 International Conference on Information and Communication Technology Convergence (ICTC)*, 2021.

[46] R. R. Karn, R. Das, D. R. Pant, J. Heikkonen, and R. Kanth, "Automated testing and resilience of microservice's network-link using istio service mesh," in *2022 31st Conference of Open Innovations Association (FRUCT)*, 2022.

[47] H. Saokar, S. Demetriou, N. Magerko, M. Kontorovich, J. Kirstein, M. Leibold, D. Skarlatos, H. Khandelwal, and C. Tang, "{ServiceRouter}: Hyperscale and minimal cost service mesh at meta," in *USENIX OSDI*, 2023.

[48] M. Ganguli, S. Ranganath, S. Ravisundar, A. Layek, D. Ilangovan, and E. Verplanke, "Challenges and opportunities in performance benchmarking of service mesh for the edge," in *2021 IEEE international conference on edge computing (EDGE)*, 2021.

[49] C. Lou, Y. Jing, and P. Huang, "Demystifying and checking silent semantic violations in large distributed systems," in *USENIX OSDI*, July 2022.

[50] H. Wu, J. Pan, and P. Huang, "Efficient exposure of partial failure bugs in distributed systems with inferred abstract states," in *USENIX NSDI*, 2024.

[51] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan, "Understanding and detecting software upgrade failures in distributed systems," in *ACM SOSP*, 2021.

[52] X. Yuan and J. Yang, "Effective concurrency testing for distributed systems," in *ACM ASPLOS*, 2020.

[53] Y. Chen, X. Sun, S. Nath, Z. Yang, and T. Xu, "Push-Button reliability testing for Cloud-Backed applications with rainmaker," in *USENIX NSDI*, 2023.

[54] T. Tu, X. Liu, L. Song, and Y. Zhang, "Understanding real-world concurrency bugs in go," in *ACM ASPLOS*, 2019.

[55] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *ACM SOSP*, 2017.

[56] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in rust at the ecosystem scale," in *ACM SOSP*, 2021.

[57] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis, "Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis," in *ACM SOSP*, 2021.

[58] X. Fu, W.-H. Kim, A. P. Shreepathi, M. Ismail, S. Wadkar, D. Lee, and C. Min, "Witcher: Systematic crash consistency testing for non-volatile memory key-value stores," in *ACM SOSP*, 2021.

[59] A. Quinn, J. Flinn, M. Cafarella, and B. Kasikci, "Debugging the {OmniTable} way," in *USENIX OSDI*, 2022.

[60] Y. Zhang, K. Rodrigues, Y. Luo, M. Stumm, and D. Yuan, "The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure," in *ACM SOSP*, 2019.

[61] Z. Chen, Y. Hua, Y. Zhang, and L. Ding, "Efficiently detecting concurrency bugs in persistent memory programs," in *ACM ASPLOS*, 2022.

[62] L. Tang, C. Bhandari, Y. Zhang, A. Karanika, S. Ji, I. Gupta, and T. Xu, "Fail through the cracks: Cross-system interaction failures in modern cloud systems," in *EuroSys*, 2023.

[63] Y. Hu, G. Huang, and P. Huang, "Automated reasoning and detection of specious configuration in large systems with symbolic execution," in *USENIX OSDI*, 2020.

[64] X. J. Ren, S. Wang, Z. Jin, D. Lion, A. Chiu, T. Xu, and D. Yuan, "Relational debugging—pinpointing root causes of performance problems," in *USENIX OSDI*, 2023.

[65] H. Ma, H. Ahmad, A. Goel, E. Goldweber, J.-B. Jeannin, M. Kapritsos, and B. Kasikci, "Sift: Using refinement-guided automation to verify complex distributed systems," in *USENIX ATC*, 2022.

[66] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, "I4: incremental inference of inductive invariants for verification of distributed protocols," in *ACM SOSP*, 2019.

[67] N. Zheng, M. Liu, Y. Xiang, L. Song, D. Li, F. Han, N. Wang, Y. Ma, Z. Liang, D. Cai, E. Zhai, X. Liu, and X. Jin, "Automated verification of an in-production dns authoritative engine," in *ACM SOSP*, 2023.

[68] U. Sharma, R. Jung, J. Tassarotti, F. Kaashoek, and N. Zeldovich, "Grove: a separation-logic library for verifying distributed systems," in *ACM SOSP*, 2023.

[69] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *ACM ASPLOS*, 2023.

[70] X. Li, X. Li, W. Qiang, R. Gu, and J. Nieh, "Spoq: Scaling {Machine-Checkable} systems verification in coq," in *USENIX OSDI*, 2023.

[71] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell, "Design and verification of the arm confidential compute architecture," in *USENIX OSDI*, 2022.

[72] R. Tao, J. Yao, X. Li, S.-W. Li, J. Nieh, and R. Gu, "Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware," in *ACM SOSP*, 2021.

[73] J. Yao, R. Tao, R. Gu, and J. Nieh, "Mostly automated verification of liveness properties for distributed protocols with ranking functions," in *ACM POPL*, 2024.

[74] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with serval," in *ACM SOSP*, 2019.

[75] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, "{DistAI}:{Data-Driven} automated invariant learning for distributed protocols," in *USENIX OSDI*, 2021.

[76] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *USENIX NSDI*, 2017.

[77] S. Renganathan, B. Rubin, H. Kim, P. L. Ventre, C. Cascone, D. Moro, C. Chan, N. McKeown, and N. Foster, "Hydra: Effective runtime network verification," in *ACM SIGCOMM*, 2023.

[78] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "APKeep: Realtime verification for real networks," in *USENIX NSDI*, 2020.

[79] K. Zhang, D. Zhuo, A. Akella, A. Krishnamurthy, and X. Wang, "Automated verification of customizable middlebox properties with gravel," in *USENIX NSDI*, 2020.

[80] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *USENIX NSDI*, 2020.

[81] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "NetSMC: A custom symbolic model checker for stateful network verification," in *USENIX NSDI*, 2020.

[82] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *ACM SIGCOMM*, 2020.

[83] S. Pirelli, A. Valentukonytė, K. Argyraki, and G. Candea, "Automated verification of network function binaries," in *USENIX NSDI*, 2022.

[84] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock, "Bagpipe: Verified bgp configuration checking," in *ACM OOPSLA*, 2016.

[85] A. Panda, M. Sagiv, and S. Shenker, "Verification in the age of microservices," in *USENIX OSDI*, 2017.

[86] C. Li, Y. Jiang, C. Xu, and Z. Su, "Validating jit compilers via compilation space exploration," in *ACM SOSP*, 2023.

[87] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield, "Using lightweight formal methods to validate a key-value storage node in amazon s3," in *ACM SOSP*, 2021.

[88] K. D. Albab, J. DiLorenzo, S. Heule, A. Kheradmand, S. Smolka, K. Weitz, M. Timarzi, J. Gao, and M. Yu, "Switchv: automated sdn switch validation with p4 models," in *ACM SIGCOMM*, 2022.

[89] N. Zheng, M. Liu, E. Zhai, H. H. Liu, Y. Li, K. Yang, X. Liu, and X. Jin, "Meissa: scalable network testing for programmable data planes," in *ACM SIGCOMM*, 2022.

[90] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," in *USENIX OSDI*, 2004.

[91] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *USENIX NSDI*, 2013.