Automated RESTful API Sequence Construction with Cross-Document Inconsistencies

Seokwon Oh
Department of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
swoh@mmlab.snu.ac.kr

Taekyoung "Ted" Kwon
Department of Computer Science and Engineering
Seoul National University
Seoul, Republic of Korea
tkkwon@snu.ac.kr

***Abstract*—RESTful APIs have become the dominant interaction mechanism for modern web services and the adoption of microservice architecture introduces new security challenges, particularly concerning stateful REST API vulnerabilities that require sequential calls to exploit. Existing methods operate on a single OpenAPI Specification (OAS) document, rendering them incapable of discovering the inter-service REST API dependencies inherent in a microservice architecture where it is ideal for each service to maintain its own independent OAS document. This lack of visibility prevents automated REST API dependency extraction in a multi-document environment. To address this gap, we propose a novel, automated method for extracting a global REST API dependency graph from multiple, distributed, and potentially inconsistent OAS documents. Our algorithm parses all provided OAS documents and employs an approach of exact matching and semantic similarity matching to resolve cross-document inconsistencies and identify REST API dependencies across service boundaries.**

Keywords—Communications, Microservice, Semantic Similarity, OpenAPI, Security, Testing

I. INTRODUCTION

Application Programming Interfaces (APIs) are the contracts that enable different software applications to communicate and exchange data with each other. They serve as an essential communication mechanism for digital systems. Additionally, Representational State Transfer (REST) has become the de facto standard for building these web services. The RESTful API leverages methods and URIs to interact, typically exchanging data via JSON or XML data formats. OpenAPI Specification (OAS), formerly known as Swagger[1], has seen widespread adoption as the most popular language for describing and documenting the RESTful APIs.

According to the Q1 2025 State of API Security by Salt Security, the threat of API attacks is growing and 99% of organizations have encountered security problems in the past year[2]. To combat these threats, API fuzzing can be leveraged. It involves sending multiple, often malformed, requests to a single REST API endpoint to test for bugs or vulnerabilities. This approach presents opportunities for improvement. Some bugs or vulnerabilities are stateful. They can be triggered by executing a specific sequence of multiple REST API operations. This has led to the development of stateful REST API fuzzing. For stateful REST API fuzzing, extracting dependencies among the REST APIs is required.

One of the efficient ways to extract dependencies between REST APIs and building REST API sequences is leveraging OAS. There are state-of-the-art methods that extract dependencies between REST APIs leveraging OAS, such as RESTler[3], RESTTESTGEN[4], Morest[5], NAUTILUS[6], and VoAPI2[7]. They explore deeper application states and uncover multi-API bugs and vulnerabilities.

In microservice architecture (MSA), it is not guaranteed that there is one single OAS document for the entire application. An MSA, by design, consists of multiple independent services, each capable of defining its own separate OAS document. This creates a direction for new research, as existing methods do not consider the aggregation of different OAS documents or the inter-service dependencies that exist between the separate OAS documents. Additionally, in MSA, attention is required on the security[8] and the consistency of the REST APIs[9].

This paper proposes a novel method of automated RESTful API sequence construction in MSA. The core of our technique is the construction of a global REST API dependency graph by parsing and analyzing multiple, distributed, and potentially inconsistent OAS documents. Our algorithm operates in two phases: graph generation and graph traversal. The graph generation phase identifies both intra-dependencies and inter-dependencies of the REST APIs using exact matching and semantic similarity matching. The graph traversal phase traverses the graph and executes the endpoints in sequences according to the dependencies identified with parameter-to-parameter matched list and schema-to-schema matched list.

The remainder of the paper is organized as follows. Section II provides the background information on the OAS and REST API dependency. Section III explains the motivation. Section IV mentions the types of cross-document inconsistencies. Section V presents the proposed method for extracting the dependencies and executing the operations. Section VI explains the evaluation of the proposed method. Section VII concludes the paper.

II. Background

This section presents an example of an OpenAPI Specification (OAS) document and illustrates a dependency relation between RESTful APIs where response data from a producer API operation serves as an input for a subsequent consumer API operation.

OAS is the language used for describing RESTful APIs. Fig. 1 shows an example OAS document describing an HTTP GET operation for the path '/v1/user/{userId}'. The operation is tagged under 'user'. The operation requires a 'userId' parameter, which is a string value, which is to be included in the URI path. If the request is successful, it returns '200' status code with the response body schema defined as 'user'. It also documents '401' response for unauthorized requests.

```
1: /v1/user/{userId}:
2:   parameters:
3:     - name: userId
4:       in: path
5:       required: true
6:       schema:
7:         type: string
8:   get:
9:     summary: Get user
10:      tags:
11:        - user
12:      responses:
13:        '200':
14:          description: Successful retrieval of user information
15:          content:
16:            application/json:
17:              schema:
18:                $ref: '#/components/schemas/user'
19:        '401':
20:          description: Unauthorized
```

Fig. 1. OpenAPI Specification (OAS) document example

The dependencies between the REST APIs require execution of the REST APIs in specific sequences. Fig. 2 shows an example of a REST API dependency. It illustrates a producer-consumer dependency in the RESTful APIs. The endpoint on the right is the producer and the endpoint on the left is the consumer. The producer endpoint is a POST request used to create a 'user'. When successful, the operation produces a unique 'userId' value that can be consumed by the consumer endpoint. The producer endpoint needs to be executed beforehand to successfully execute the consumer endpoint.

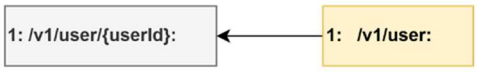

Fig. 2. REST API Dependency Example

III. Problem and Motivation

This section explains the problem this work focuses on and the motivation behind it.

REST API dependency is inferred by following the pointers within the OAS document. Fig. 3 illustrates a producer-consumer REST API dependency in the OAS document. API 1 is a producer which generates 'data B' and API 2 is a consumer which uses 'data B'. The data exists within one OAS document and therefore, consistent internal reference is guaranteed.

When adopting multiple OAS documents for an MSA application, inter-service dependencies need to be identified. The inter-service dependencies can be discovered by matching the data across different OAS documents. From Fig. 4, a dependency between API 1 and API 2 can be inferred because they share common data.

However, if there is an inconsistency between the data used across OAS documents, it presents a challenge. According to Fig. 5, although both API 1 and API 2 use the same data, due to the inconsistency between the data 'surname' and 'firstname', it cannot be inferred that API 1 and API 2 share a common data. The data structures are not identical. As a result, a simple matching algorithm would fail to connect them or identify dependency between them, even though there is one. This example demonstrates why an advanced technique is required to resolve the inter-dependencies in the environment where multiple OAS documents are independently managed.

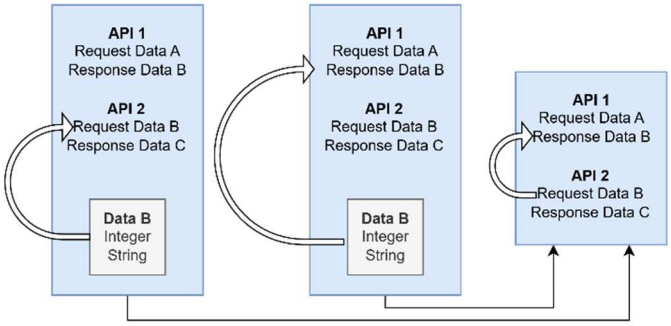

Fig. 3. Single OAS document

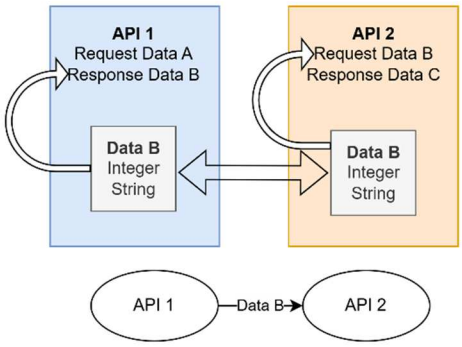

Fig. 4. Multiple OAS documents, identical data

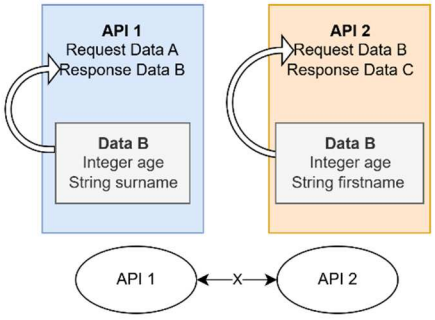

Fig. 5. Multiple OAS documents, disparate data

IV. TYPES OF INCONSISTENCIES

The work identifies 12 cross-document inconsistency types categorized into 3 domains: schema-to-parameter, schema-to-schema, and property-to-property. The 12 inconsistency types are illustrated from Fig. 6 to 17.

The schema-to-parameter domain from type 1 to 3 addresses cases where a producer's response schema property acts as a consumer's request parameter, varying from exact matches to semantically related or unrelated names. For type 1, the producer API returns a schema containing a property that is used by the consumer API as a parameter with an identical name. For type 2, the producer API returns a schema property that corresponds to the consumer API parameter with a different but semantically related name. For type 3, the producer API returns a schema property that corresponds to the consumer API parameter with a different and semantically unrelated name.

The schema-to-schema domain from type 4 to 9 encompasses schema-to-schema relationships, managing scenarios where a producer's response schema corresponds to a consumer's request body schema despite discrepancies in schema names, property counts, or semantic similarities. For type 4, the producer's response schema and the consumer's request body schema share the identical name and possess identical properties. For type 5, the producer's response schema and the consumer's request body schema share the identical name but they differ in the count of the contained properties. For type 6, the producer and consumer schema names are different but semantically related, while their underlying properties are identical. For type 7, the producer and consumer schema names are identical but the internal property names are different, yet semantically related. For type 8, both the schema names and the property names differ, yet they remain semantically related. For type 9, both the schema and the property names differ and they are semantically unrelated.

The property-to-property domain from type 10 to 12 involves inconsistencies, where a consumer's request schema is composed of properties derived from multiple producer APIs, distinguished by whether the property names exactly match, are semantically related, or are unrelated. For type 10, the consumer API request schema is composed of properties derived from the multiple producer APIs where all the property names match. For type 11, the consumer API request schema is composed of properties derived from the multiple producer APIs where the property names differ but are semantically related. For type 12, the consumer API request schema is composed of properties derived from the multiple producer APIs where the property names differ and are semantically unrelated.

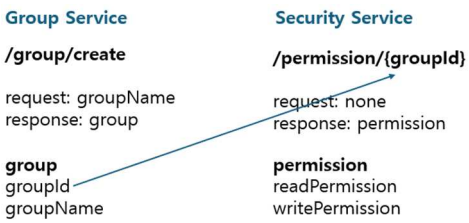

Fig. 6. Type 1 inconsistency.

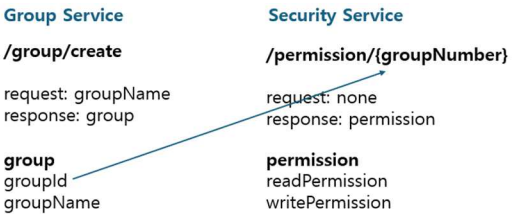

Fig. 7. Type 2 inconsistency.

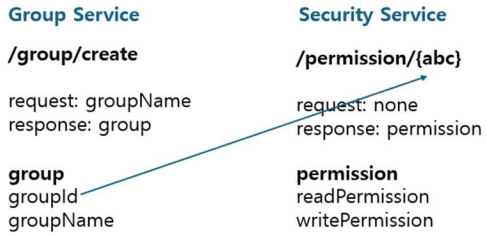

Fig. 8. Type 3 inconsistency

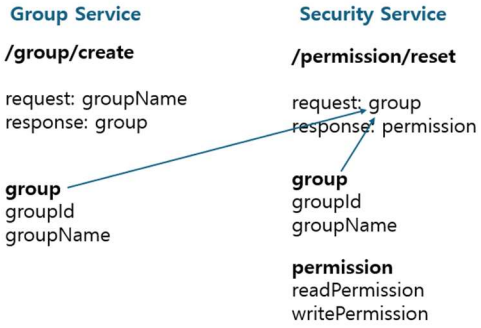

Fig. 9. Type 4 inconsistency

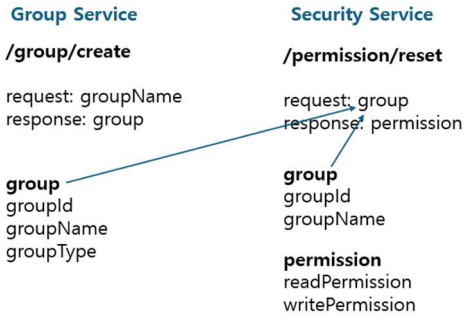

Fig. 10. Type 5 inconsistency

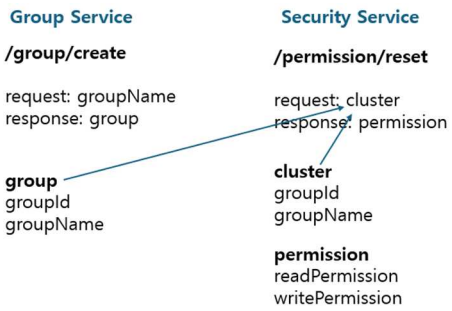

Fig. 11. Type 6 inconsistency

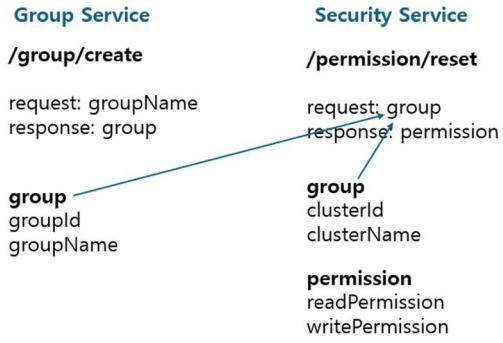

Fig. 12. Type 7 inconsistency

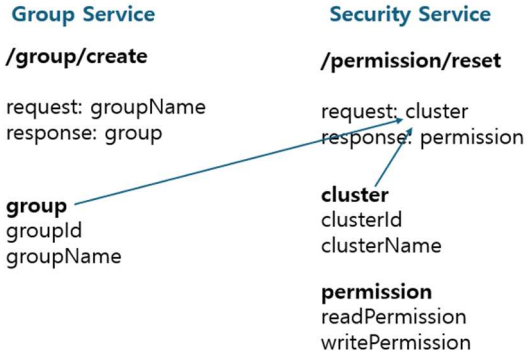

Fig. 13. Type 8 inconsistency

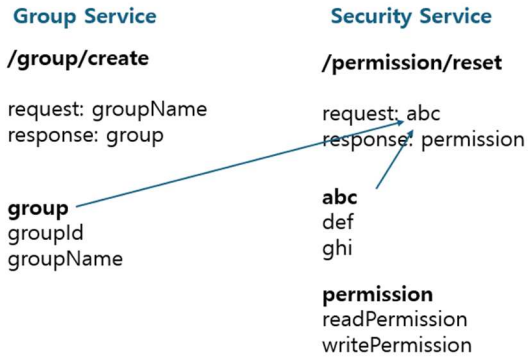

Fig. 14. Type 9 inconsistency

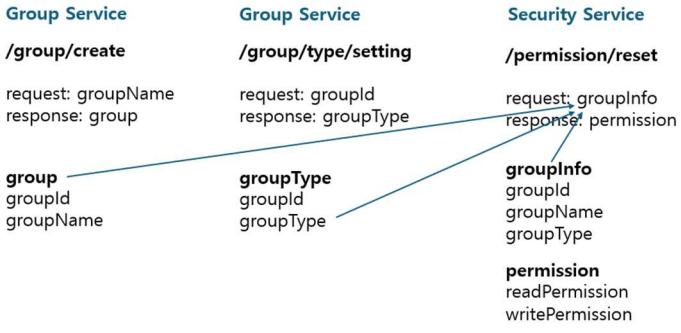

Fig. 15. Type 10 inconsistency

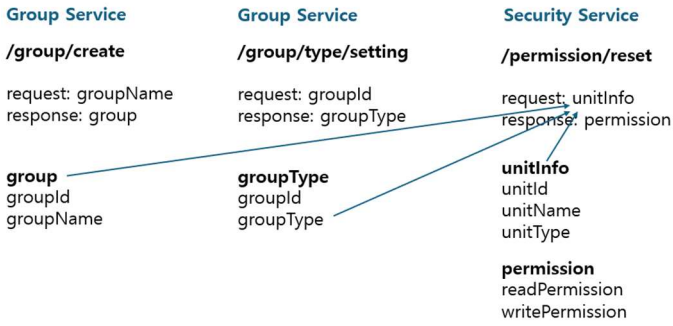

Fig. 16. Type 11 inconsistency

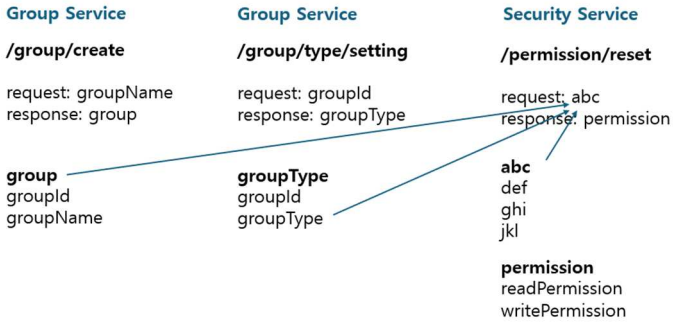

Fig. 17. Type 12 inconsistency

V. METHODOLOGY

This section illustrates the detailed proposed method of extracting RESTful API dependency in MSA applications.

The diagram in Fig. 18 illustrates the high-level design of the proposed method. It begins by taking multiple OAS documents as input. It parses the documents and extracts the parameters and schemas. Based on the parameters and schemas, exact matching and semantic similarity matching are conducted to identify dependencies between the REST API operations. A graph is generated with an endpoint as a node and a dependency as an edge. The matching algorithm generates the data structures (parameter-to-parameter matched list, schema-to-schema matched list) while processing the first step. In the next step, graph traversal is executed to call the REST API operations in the sequences according to the identified dependencies.

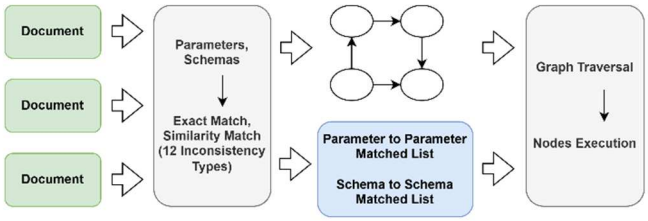

Fig. 18. Design overview

A. Graph Generation

The first phase is generating a global REST API dependency graph. Fig. 19 outlines the matching algorithm used to identify REST API dependencies. The process begins by aggregating all the OAS documents into a single logical pool. It then iterates

through each individual API operation endpoint, assigning a unique ID for each endpoint.

After parsing the operations, it selects a target API operation and checks whether the target API operation only requires path or query parameters. If it does, it conducts *parameter match*, which is checking the existence of an identical field between the target API operation's parameters and the source API operation's response properties, and *parameter semantic similarity match*, which is checking the existence of a semantically similar field between the target API operation's parameters and the source API operation's response properties. If any match is found, it is inferred that there is a dependency between the two API operations. It also checks whether the target API has a request body. If it does, it conducts *parameter match*, *parameter semantic similarity match*. Additionally, between the target API operation's request body schema and the source API operation's response schema, it conducts *schema name match*, which is checking the existence of an identical schema name, *schema match*, which is checking the number of identical contained properties, *schema name semantic similarity match*, which is checking the existence of a semantically similar schema name, and *property semantic similarities match*, which is checking the existence of a semantically similar property. If any match is found, it is inferred that there is a dependency between the two API operations.

After completing the matching algorithm, each API operation is mapped to a node and each dependency is mapped to an edge to form a global REST API dependency graph.

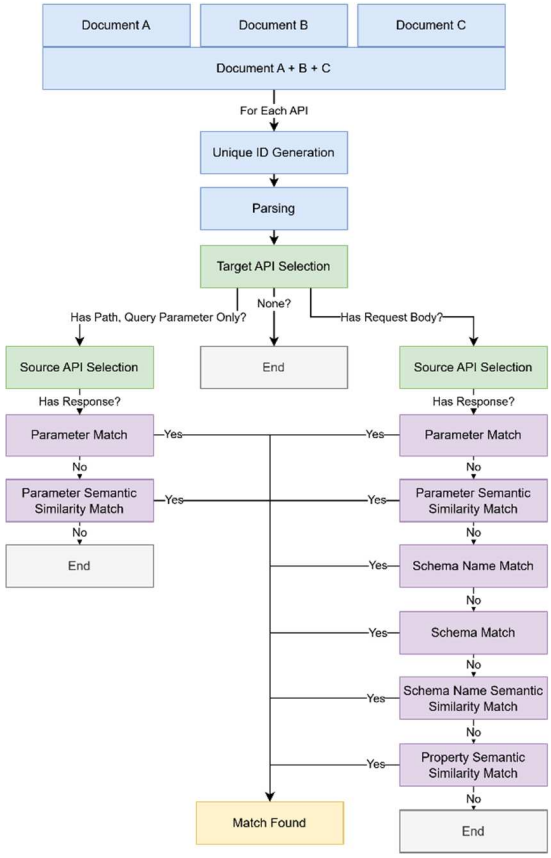

Fig. 19. Matching algorithm

B. *Graph Traversal*

The second phase is traversing the global REST API dependency graph and executing the API operations in sequences according to the dependencies identified. During the first phase for any match found, an entry is added in parameter-to-parameter matched list or schema-to-schema matched list. The graph traversing is done leveraging these two lists.

Graph traversing with parameter-to-parameter matched list allows executing API operations with values from hidden dependent API operations due to inconsistent parameters. An example is shown in Fig. 20. The producer endpoint '/group/create' is executed, which returns 'group' object containing 'groupId'. Then it executes a consumer API operation endpoint '/permission/{groupNumber}' which requires a path parameter named 'groupNumber'. By consulting the parameter-to-parameter matched list, the system finds an entry that 'groupId' and 'groupNumber' are matched. It allows processing that the 'testGroupId' value produced by '/group/create' endpoint can be used for the 'groupNumber' parameter in the '/permission/{groupNumber}' endpoint. This successfully executes the API operation with dependency despite the different property and parameter names, which is a type of inconsistency between two independent OAS documents.

Graph traversing with schema-to-schema matched list allows executing API operations with the values from hidden dependent API operations due to inconsistent schemas. An example is shown in Fig. 21. The producer endpoint '/group/create' is executed, which returns a 'group' object. Then, it needs to execute the consumer endpoint '/permission/reset', which requires a 'cluster' object as its input. By consulting the schema-to-schema matched list, it finds an entry that 'group' schema matches 'cluster' schema. The system knows that it can use the 'group' object data to make a call to '/permission/reset' endpoint. It successfully executes the API with dependency despite the different schema names, which is a type of inconsistency between two independent OAS documents.

After completing the second phase, API operations are executed in order according to the identified dependencies. If there is a match found between two API operations in the first phase, the dependency between these two API operations are identified and the API operations are executed in order in the second phase. If there is not a match found in the first phase, there is not a dependency between the two API operations and the operations are not executed in order in the second phase.

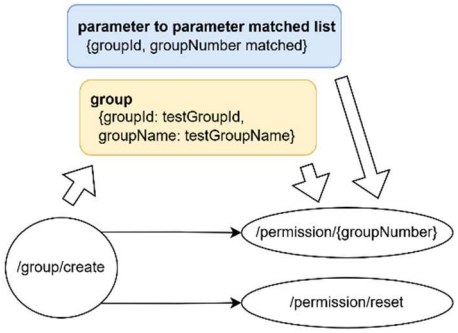

Fig. 20. Traversing with parameter-to-parameter matched list

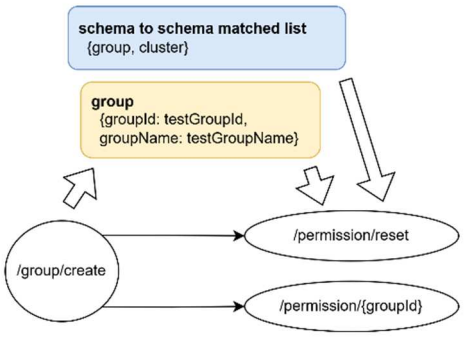

Fig. 21. Traversing with schema-to-schema matched list

VI. EVALUATION

To evaluate the proposed method, a prototype was implemented in Python[10] using WordSegment[11] for parameter, schema name, property value segmentation and Sentence-BERT[12] for semantic similarity calculation. To test the prototype, a test application was developed with Spring Boot[13], which consists of 3 microservices hosted on a Ubuntu 24.04.3 virtual machine. The testbed comprises 19 API endpoints and 30 ground-truth API dependencies, designed to incorporate 16 instances of the 12 cross-document inconsistency types embedded in inter-service and intra-service dependencies.

The method's effectiveness is measured based on the percentage of API operations successfully executed during graph traversal. The result demonstrates that at optimal semantic similarity thresholds of 0.2 and 0.4, it achieves 84% coverage. As the similarity threshold increases to 0.8, the coverage drops to 58% as the algorithm rejects valid dependency links that are semantically related but not highly similar, causing breaks in the execution chains. Table I shows the coverage according to each threshold. A direct comparison with the existing methods was not feasible by the limitation that the existing methods operate on single OAS document, rendering them incompatible in the multi-document environment.

The number of edges in the generated graph represents the potential producer-consumer API dependencies identified. The number of edges varies significantly depending on the semantic similarity threshold applied during the graph generation phase. At the low threshold, the algorithm generates an over-approximation of the graph with 323 edges and at the high threshold, the algorithm filters out low-confidence edges, drastically reducing the count to 46. Table I shows how the number of edges changes according to the thresholds. As the threshold increases, it reduces the coverage but it reduces the resource required to compute the proposed method.

TABLE I. Node count, edge count, coverage

Threshold	0.2	0.4	0.6	0.8	Ideal
Node Count	19	19	19	19	19
Edge Count	323	199	68	46	30
Executed Nodes	16	16	15	11	19
Coverage	84%	84%	79%	58%	100%

VII. CONCLUSION

This work addresses the challenge for stateful RESTful API testing in modern microservice architecture (MSA). In MSA, it is ideal to maintain an independent OAS document for each service while the existing methods assume one single comprehensive OAS document. To handle multiple OAS documents in MSA, inter-service REST API dependencies across separate independent OAS documents need to be identified. We define the 12 types of the cross-document inconsistencies and propose the method that constructs the global REST API dependency graph that identifies the dependencies despite the inconsistencies. Based on the graph, the REST API operations are executed in order.

This work provides an automated black-box approach to construct comprehensive REST API sequences from multiple independent OAS documents in MSA. This method provides a foundation for enabling stateful bug or vulnerability detection across the entire MSA application with multiple OAS documents, rather than detecting within an isolated service.

ACKNOWLEDGMENT

This paper is based on the first author's 2026 master's thesis.

REFERENCES

[1] Swagger, https://swagger.io/

[2] Salt Security, Q1 2025 State of API Security, https://content.salt.security/state-api-report.html

[3] V. Atlidakis, P. Godefroid and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 2019, pp. 748-758

[4] E. Viglianisi, M. Dallago and M. Ceccato, "RESTTESTGEN: Automated Black-Box Testing of RESTful APIs," 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, 2020, pp. 142-152

[5] Y. Liu et al., "Morest: Model-based RESTful API Testing with Execution Feedback," 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 2022, pp. 1406-1417

[6] Gelei Deng, Zhiyi Zhang, Yuekang Li, Yi Liu, Tianwei Zhang, Yang Liu, Guo Yu, and Dongjin Wang, "NAUTILUS: automated RESTful API vulnerability detection," In Proceedings of the 32nd USENIX Conference on Security Symposium (SEC '23), USENIX Association, USA, 2023, Article 313, 5593–5609

[7] Wenlong Du, Jian Li, Yanhao Wang, Libo Chen, Ruijie Zhao, Junmin Zhu, Zhengguang Han, Yijun Wang, and Zhi Xue, "Vulnerability-oriented testing for RESTful APIs," In Proceedings of the 33rd USENIX Conference on Security Symposium (SEC '24). USENIX Association, USA, 2024, Article 42, 739–755

[8] R. Kanishka Jayalath, H. Ahmad, D. Goel, M. Shuja Syed and F. Ullah, "Microservice Vulnerability Analysis: A Literature Review With Empirical Insights," in IEEE Access, vol. 12, pp. 155168-155204, 2024

[9] Alexander Lercher, "Managing API Evolution in Microservice Architecture", In Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), Association for Computing Machinery, New York, NY, USA, 2024, 195–197.

[10] Python, https://www.python.org/

[11] WordSegment, https://pypi.org/project/wordsegment/

[12] Nils Reimers and Iryna Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," In EMNLP-IJCNLP, 2019, pp. 3982-3992

[13] Spring Boot, https://spring.io/projects/spring-boot