Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2023.047051

ARTICLE

RESTlogic: Detecting Logic Vulnerabilities in Cloud REST APIs

Ziqi Wang*, Weihan Tian and Baojiang Cui

School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing, 100876, China

*Corresponding Author: Ziqi Wang. Email: wangziqi@bupt.edu.cn

Received: 23 October 2023 Accepted: 17 December 2023 Published: 27 February 2024

ABSTRACT

The API used to access cloud services typically follows the Representational State Transfer (REST) architecture style. RESTful architecture, as a commonly used Application Programming Interface (API) architecture paradigm, not only brings convenience to platforms and tenants, but also brings logical security challenges. Security issues such as quota bypass and privilege escalation are closely related to the design and implementation of API logic. Traditional code level testing methods are difficult to construct a testing model for API logic and test samples for in-depth testing of API logic, making it difficult to detect such logical vulnerabilities. We propose RESTlogic for this purpose. Firstly, we construct a test group based on the tree structure of the REST API, adapt a logic vulnerability testing model, and use feedback based methods to detect code document inconsistency defects. Secondly, based on an abstract logical testing model and resource lifecycle information, generate test cases and complete parameters, and alleviate inconsistency issues through parameter inference. Once again, we propose a method of analyzing test results using joint state codes and call stack information, which compensates for the shortcomings of traditional analysis methods. We will apply our method to testing REST services, including OpenStack, an open source cloud operating platform for experimental evaluation. We have found a series of inconsistencies, known vulnerabilities, and new unknown logical defects.

KEYWORDS

Cloud services; RESTful API; logic defects; inconsistency issues

1 Introduction

Next-generation networks [1], represented by the Cloud platform, offer users the choice of hosting their services with improved performance and reduced costs through APIs [2], significantly alleviating the operational and maintenance burden for tenants based on Shared Responsibility Models (SRM) [3]. REST architecture [4] is a commonly used and probably the most popular specification for web APIs, especially for cloud APIs. The fundamental concept of REST revolves around 'resources', meaning any entity uniquely identifiable by a Uniform Resource Identifier (URI) and accessible through a combination of operators (e.g., PUT, POST, GET, PATCH, DELETE) and a Uniform Resource Locator (URL). Generally, almost all activities within the cloud environment can be abstracted as operations on resources under the REST architecture.

In this context, the security management of resources, especially the underlying logic vulnerability, poses significant challenges for cloud operators and tenants. Unlike traditional vulnerabilities rooted in code-level errors, logic vulnerabilities often stem from faults or design flaws in the application's business logic. These can arise from an inadequate understanding of the logic or improper coding, leading to logic defects. For those logic defects that may threaten the security operation of the entire API application or encroach on the information and privacy of users. we call them API logic vulnerabilities. When occurred in cloud computing scenarios, these may result in issues like resource quota bypass, resource leakage, and problems with privilege management [5–7]. For example, if authentication is not set properly, an attacker may guess or steal other users' tokens and then delete the legal users' resources. Concerning the inconsistency issues between the code and documentation of business logic, we call them logical defects that may affect the testing process. Inconsistency issues and logical vulnerabilities are both logical defects. Generally, inconsistency issues do not directly lead to the occurrence of logical vulnerabilities but can become a challenge in the testing process of logical vulnerabilities. The above logical defects are all related to the design and implementation of APIs, therefore analyzing from the perspective of APIs is a reasonable solution.

Compared with traditional vulnerabilities, logic vulnerabilities may cause greater harm and are more difficult to find (generally won't cause a program crash) [8] as they require a deep understanding of the application's business logic. Therefore, it is imperative to propose a Cloud REST APIs logic vulnerability detecting method for securing the cloud. Microsoft's fuzzing framework *RESTler* operated in black-box mode with several active checkers to find the unhandled exceptions (e.g., service crashes) and other bugs caused by violation of safety rules [5,9–11]. EMResearch's *Evomaster* utilized a black box to randomly generate tests and a white box to evolutionarily improve the coverage [12], and successfully discovered 80 bugs. Nevertheless, there are still some common limitations in existing work such as insufficient understanding of API logic during the period of test generation and confined analysis dimension of testing results. For example, some parameters in different APIs possess different names but point to the same resource, unfortunately, existing tools cannot process such dependencies resulting in insufficient tests. Furthermore, the test target focusing on server errors and the test oracle concentrating on status code analysis are also the reasons for such a dilemma.

To address the above limitations, we propose a novel method named RESTlogic for detecting logic vulnerabilities in cloud REST APIs. Firstly, we propose a series of REST API logic information processing methods to ensure the effectiveness of our testing. Among them, we first propose an APIs division strategy to filter APIs into different test groups since they may reveal different resource lifecycles and logic rules. Although relatively mature tools, such as *Swagger-Codegen* [13], help developers realize automatic API-specification conversion, there are still a large number of inconsistencies between APIs and their specifications due to the misuse of these tools or poor understanding of either side in the manual writing stage. Hence, we propose an inconsistency-checking strategy to detect such issues. We found many problems in open-source cloud APIs and received official approval. Additionally, we introduce a method for obtaining API runtime intermediate information through code instrumentation, enabling a more comprehensive, multi-dimensional analysis of testing results.

Secondly, our approach incorporates a test generation method primarily driven by an abstracted test model and Parameter Inference (PI). Existing tools offer various perspectives for test generation, such as the heuristic-based one [14] and the dependency-based one [9], and mainly generate parameters based on request schema [14], data dictionary [9], parameter mutation [12,15], or dynamic feedback [9]. For better-detecting logic security problems, we construct abstracted logic test models based on resource lifecycle, used to describe the test procedures of API logic, including the generation and execution of the individual tests, as well as test sequence with timing relationships. Additionally, for

the inconsistency issues discovered from the previous stage, we also provide a remediation method based on PI and prove its effectiveness. Following these key principles, our method achieves improved coverage in testing API logic.

Thirdly, we propose a vulnerability analysis method based on Call Stack (CS) anomaly detection, serving as an augmentation to the conventional status code and response analysis. Traditional black-box fuzzing tools, which utilize Hypertext Transfer Protocol (HTTP) responses as test oracles [16,17], rely heavily on response matching, thus failing to detect logic defects masked by seemingly normal responses. To compensate for these common shortcomings of existing works, we introduce a call stack anomaly detection method to analyze the runtime logic of APIs. Abnormal extensions, reductions, unexpected internal changes, or even normal patterns that should not appear in the call stacks may indicate logic anomalies. In this way, we cannot only discover bugs by status code checking or response validation but also detect the logic vulnerabilities revealed by call stack information.

To assess the efficacy of our method, we implemented our work on real-world open-source REST services, including OpenStack [18–20], one of the most popular cloud computing infrastructure software. Our experiments revealed several inconsistencies that may impacted the testing process between APIs and their specifications, some of which have received official confirmation. In terms of Parameter Inference (PI), we developed a prototype that demonstrates its effectiveness in improving coverage. Utilizing our methods, we were able to replicate disclosed vulnerabilities and discover new potential security flaws in cloud operating systems (a demonstration is available at https://github.com/restlogic/diana).

In summary, our contributions are as follows:

1. We propose a REST API logic information manipulation method, associate REST APIs with logical testing models, discover inconsistency issues between API and documentation, and obtain runtime intermediate information to provide more analysis dimensions.
2. We provide a PI-enhanced model-based test generation method, in which we abstract the logic test model, generate tests according to the logic test model and resource lifecycle, and solve the inconsistency issues by PI to improve the testing depth of tests.
3. We introduce a CS-assisted tests execution analysis method based on sparse call stack anomaly detection to cooperate with status code analysis, to find security defects that may not be found only through status code checking and response validation.
4. We employed our method to test the real-world open-source REST services, especially on cloud services. We found certain inconsistency issues, disclosed vulnerabilities, and unexposed logic defects, proving the coverage and effectiveness improvement achieved by our method.

The remaining parts of the paper are organized as follows. Section 2 sums up existing works in related fields, Section 3 describes the core concepts and issues to be solved in this paper, and presents the main framework of our proposed method. Sections 4–6 introduce the detailed design of our method, including the logic information processing, test generation, and execution analysis. Section 7 explains the specific setups and results of our experiments. Section 8 summarizes our work.

2 Related Work

2.1 Logic Vulnerability Analysis

Modern vulnerabilities are more diversified and coupled with logic, and recent years have witnessed tremendous attention and development of related fields. Researchers utilized model checking, symbolic execution [21], fuzz testing, and other technologies to realize the logic vulnerability evaluation

of web applications [22,23] and protocols [24], including e-commerce and payment syndication services [8,25]. However, the early works mainly focus on single-function applications, and the proposed methods are not completely adapted to cloud operating systems.

Deepa et al. [22] proposed DetLogic and developed models for three types of web application logic vulnerabilities, including parameter manipulation, access control, and workflow bypass vulnerabilities. They adopted a black-box approach to automatically construct finite state machines for web services, used to derive constraints related to parameter manipulation, access control, and workflow. They discover vulnerabilities by creating samples that violate these constraints.

Li et al. [23] proposed a systematic black box method for detecting logical vulnerabilities in PHP web applications. The author constructs a standardized finite state machine model of the application through a user simulator, further constructs unexpected inputs based on symbolization, and detects the actual running Finite State Machine (FSM) model. By judging the consistency between FSMs, the author preliminarily screens for potential logical vulnerabilities.

Vanhoef et al. [24] proposed a logic vulnerability testing method for the WiFi handshake protocol. They understand and define the various stages and expected behaviors of the protocol, and then construct a model for the WiFi handshake protocol. Further, generate test cases based on the model to explore state transitions, especially edge cases in each stage. By comparing expected behavior with actual behavior, identify deviations and potential vulnerabilities in implementation.

Chen et al. [8] researched predicting logical vulnerabilities in payment syndication services. They utilize natural language processing techniques to conduct automated document analysis and predict vulnerabilities by checking whether syndication services will result in security requirements being unenforceable, instead of analyzing the source code. They experimented on real-world payment applications and discovered 5 new security-critical vulnerabilities.

2.2 REST API Testing

Recent works have proved that cloud platforms are more sensitive to logic vulnerabilities, especially the privilege escalation [26,27] and authentication bypass caused by misconfiguration [28]. For other complex software systems, such logic defects are also widespread, including smartphone systems [29], kernel [30], and binary firmware [31]. We summarized the area closest to our work, i.e., REST API testing, to evaluate the latest research findings in the field.

Microsoft conducted a series of studies for testing REST APIs [9], including API security properties checking [5] and differential regression testing [11]. They mainly adopted the method of fuzzing testing, generated test sequences based on API document specifications and producer-consumer dependencies, and detected bugs such as server handling exceptions. Their recent work also proposed a method to detect inconsistencies between code and documents but did not propose any mitigation measures.

Evomaster provided both white-box and black-box testing approaches to detect server errors [12]. Provided with the API and corresponding documentation, black box mode generates random inputs, while white box mode accesses the source code of the API and uses evolutionary algorithms to optimize the coverage. They conducted experiments on five open-source RESTful services and discovered 80 real-world application bugs.

The current mainstream method is to process API documents through dependency analysis [32,33], and generate test cases based on the model [34] or property [35]. The most recent work released by Kim et al. [36] analyzed and evaluated the works in REST API testing and put forward a series of

suggestions for researchers in related fields. The authors also point out that inconsistencies between API documentation and implementation hinder the effectiveness and coverage of the testing.

Therefore, our method RESTlogic focuses on the resource lifecycle of the cloud platform from the perspective of REST API, providing a general method, including an API logic information manipulation method, a PI-enhanced model-based test generation method, and a CS-assisted tests execution analysis method, aiming to improve the coverage and effectiveness of API logic defect analysis, especially for cloud REST APIs. The summary of existing research in the field is as Table 1:

Table 1: Summary of related work

Name	Ref.	Target	Testing approach	Test generation	Detect approach
DetLogic	[22]	web application	Black-box	FSM based	FSM matching
Logicscope	[23]	web application	Black-box	FSM based	FSM matching
Vanhoef et al.	[24]	WiFi protocol.	Black-box	Model based	Expectation matching
Chen et al.	[8]	Payment service	–	–	Doc analysis
Restler	[9]	REST service	White-box	Dependency based	Status code
Evomaster	[12]	REST service	Black/white-box	Random/evolutionary	Status code
Morest	[34]	REST service	Black-box	Model based	Execution feedback
Quickrest	[35]	REST service	Black-box	Property based	Response checking

3 Method Design

3.1 Concept Description

REST API URI designing best practices. REST APIs are designed to represent interfaces for resources served by a web service, typically implemented over HTTP. Best practices, such as those recommended by Microsoft [37], advocate for organizing URIs into a hierarchical structure for collections and items. For instance, OpenStack Compute API uses *GET*/*servers* for listing servers and *GET*/*servers*/{*server_id*} for specific server details. Relationships between different resource types are represented hierarchically, like *GET*/*servers*/{*server_id*}/*metadata*} for a server's metadata.

Trie. A trie, or prefix tree, organizes a set of strings over a finite alphabet set, where strings sharing prefixes follow the same path from the root to their common prefix node. Each node is labeled with a character, except for the root, which is empty.

Test group. This concept filters APIs into groups based on specific criteria. Each group contains APIs sharing common composition modes and potential logic threats.

Logic information. Encompasses both static elements like code and API documentation, including their consistency, and dynamic elements such as observed runtime characteristics.

3.2 Challenge for Test REST Service

Complex API logic. In real-world REST services, multiple APIs often need to cooperate to provide services, meaning that a test for a business logic flow requires a combination test of multiple APIs. Different API combinations present varied security threats. For instance, a sequence like "Conduct *DELETE* method after *POST* method on the same resource" could lead to privilege bypass issues.

Despite revealing crucial resource lifecycles, such API combinations are often overlooked in existing studies.

Code-document inconsistency issues. REST services generally face inconsistency issues like request definition mismatches, incorrect URI paths, and parameter discrepancies. These inconsistencies, often neglected during testing, can hinder automated testing and impact the reliability of safety analysis results.

Inadequate analyzing dimension. Especially in cloud platforms, the lack of visibility into internal implementations poses challenges for security analysis. Existing approaches mainly focus on external information, like HTTP responses, but often overlook the runtime intermediate information, which may indicate abnormal logic execution. This limitation remains a challenge in effectively analyzing logic vulnerabilities in REST services.

3.3 Motivation\|Design Goals

Effectiveness. Randomly generated tests may be discarded at an early stage, restricting the testing depth and efficiency, and causing inadequate testing on logic flow. Resource lifecycle-oriented test generation is designed to construct a test sequence that will not be discarded as early as possible.

Coverage. The reduction of testing coverage may be affected by insufficient understanding of API combination patterns and Code-document inconsistency issues. The division of test groups and parameter inference are designed for this purpose.

Capability. Here capability refers to the ability to detect security defects. To expand the analysis dimension, we introduce runtime intermediate information analysis as a supplement, which can help analysts improve their analysis ability and thus improve the possibility of detecting security defects.

3.4 Main Framework

As illustrated in Fig. 1, we follow the above challenges and design goals when designing our methods, and propose a general framework, which can be divided into three parts, and the former parts provide the basis for the latter parts. The content in the solid box is the core content of our method design, and that in the dotted box is used to support the whole test process.

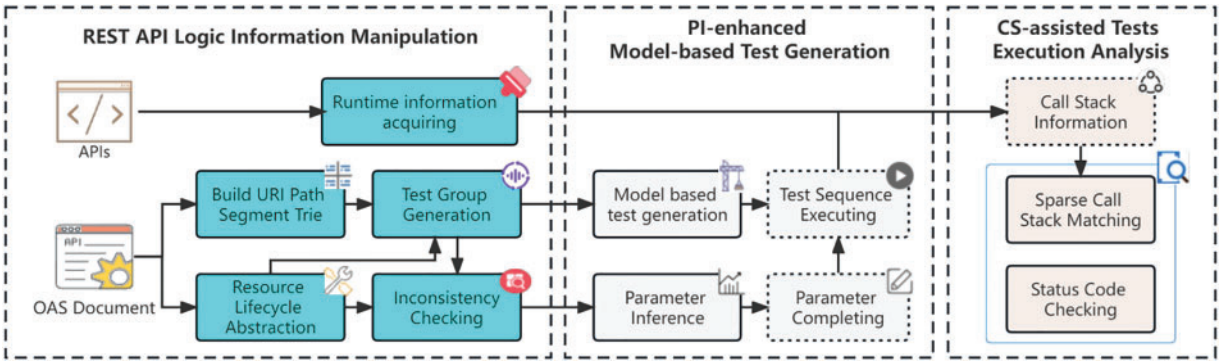

Figure 1: Main framework of our method

REST API Logic Information Manipulation. The design objective of this part is to analyze the external expression of REST API logic and provide a detailed basis for sample generation and result analysis. For static logic information, we analyze the API documents, reveal the possible logical

relations between APIs, and generate test groups for API groups that may be faced with the same kind of security issue. At the same time, we abstract the resource lifecycle and check the inconsistency issues based on resource attributes and test group information, aiming to improve test depth, i.e., coverage. Moreover, we provide an optional dynamic runtime information-acquiring method to obtain runtime features that can represent API logic.

PI-Enhanced Model-Based Test Generation. First of all, we generate test cases based on resource lifecycle, test group, and associated vulnerability model. The test case is in the form of a request sequence with a relationship or dependency. Parameter inference, as a solution proposed by us to alleviate the inadequacy of REST API testing, is designed to offer help in the parameter completion stage. The inferred parameters can effectively alleviate the reduction of dependency analysis accuracy caused by inconsistency issues, further improving the coverage of test execution.

CS-Assisted Tests Execution Analysis. Test analysis based on status code is intuitive and effective but also has certain limitations. Call stack information obtained from runtime information acquiring can effectively assist in defect analysis of complex logic external expression, especially when the defect cannot be distinguished according to the status code. The matched specific call stack structure indicates the existence of the target pattern, which can be used for the detection of normal or abnormal logic runtime features. Finally, we can detect logic defects with the help of joint analysis of HTTP response and call stack information in this stage.

4 REST API Logic Information Manipulation

Following the main framework proposed in Section 3.4, we supplement the description of our method as follows. To make an adequate understanding of REST API logic and further support the test generation and result analysis, our REST API logic information manipulation method mainly contains test group generation, code-document inconsistency checking, and runtime intermediate information acquiring.

4.1 API Paths Trie and Test Group Generation

The main approach of most existing fuzzing technologies to find program defects is to improve the code coverage, and it shows beneficial effects on traditional vulnerabilities such as memory corruption. However, such consideration may not work the same since the composition of API is more complex and implicates more logical information. Therefore, it shall be meaningful to realize the understanding of API logic to reach higher coverage for API logic testing. We refer to [38] and propose a method for API analyzing and clustering according to OpenAPI Specification [39] (OAS, one of the most popular API documentation specifications) documentation for improving the ability to find logic defects.

4.1.1 Resource Lifecycle Abstraction

Every instance of all Resource types must have lifecycles being implemented. For example, throughout the whole lifecycle of a block storage instance, it should be created, read, updated, and eventually deleted. We can describe that instance lifecycle as a FSM: $M(\Sigma, S, s_0, \delta, F)$, where:

- Σ is the operation method set, in this case, is the lifecycle function set related to the Resource type, like {*Create, Read, Update, Delete*}, and each method in Σ has an HTTP method corresponding to it, as illustrated in Fig. 2a.
- S is a finite non-empty set of states, in this case, is {*Pre-Creation, Available, Finalized*}.
- s_0 is the initial state, an element of S, in which case is *Pre-Creation*.

- δ is the state-transition function δ: S $\times$ Σ $\rightarrow$ S, which was described in Fig. 2a column "FSM TD".
- F is the set of final states, a subset of S, i.e., $F \subseteq S$, in this case, $F = \{Finalized\}$.

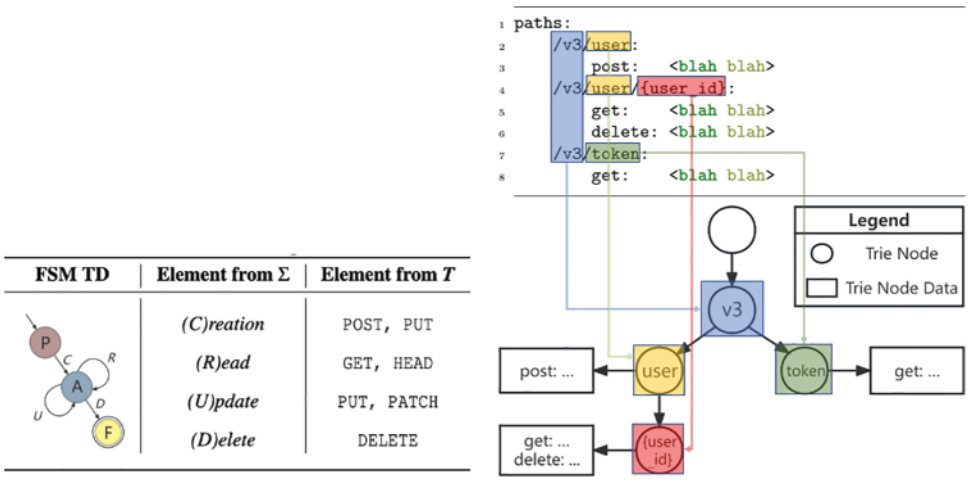

Figure 2: (a) Resource lifecycle FSM TD, (b) generate segment tries based on API docs

The FSM transition diagram (TD) is depicted in Fig. 2a. The lifecycle of an instance starts at state *(P)re-Creation*, transits into state *(A)vailable* by input *Create*, and finally ends up in state *(F)inalized* by input *Delete*. While the instance is at the state *Available*, input *Read* or *Update* does not change its state. As for REST API, the according representation of lifecycle FSM's input is the HTTP method. We define two sets, one is FSM's input Σ, and the other is HTTP method set T (according to RFC 5789 [40], RFC 7231 [41]). The relation between them is also summarized as shown in the table.

4.1.2 REST API URI Path Segment Trie

Considering that URIs related to the same resource often share the same URI prefix, we can now build up a data structure similar to URI trie by converting REST APIs representation into the data structure as the trie nodes [42], where strings with the same prefixes share the same path from the root node to their common prefix node.

REST API URI hierarchical design indicates the internal logic relations that the more similar the APIs are, the closer the logical relationship they possess. Inspired by the conventional trie and REST API URI Designing Best Practices, we use the URI path segment as the trie node label. This leads us to devise a method for Trie construction, where REST APIs of a cloud service are assigned to Trie nodes. Consequently, REST APIs can be clustered based on their shared parent nodes. The pseudocode of constructing a URI Path Segment Trie is as shown in Algorithm 1, and the corresponding procedure of constructing a URI Path Segment Trie is listed as follows:

- Parse all OAS document "*paths*" segment. (Line 1)
- Initialize a trie by constructing a root node. (Line 2)
- Iterate through all API paths and conduct the following sub-procedure for each of them:
  - Split path by URI segment delimiter "/". (Line 6)
  - Strip the empty segment before the first delimiter. (Line 7)

- For each segment, attach a node with that segment as a label when that segment is absent in its parent's sub. Otherwise, get a node from the parent node's sub with that segment as a key. (Lines 9–22)
- Attach the API's detailed description to the last node's *api_unit* field. (Line 23)

To demonstrate the general procedure for building URI Path Segment Trie, consider the following case: As illustrated in the upper part of Fig. 2b, given an OAS document example, the Trie being built from it can be described by the lower part of the figure. In conclusion, a REST API URI Path Segment Trie was built from the OAS document for testing group generation, which will be described in the following section.

Algorithm 1: Build URI Path Segment Trie

Input: OpenAPI Specification Documentation *oas_doc*
Output: URI Path Segment Trie root *root*

```
1:  path ← oas_doc.getnode("paths")
2:  # used to describe API
3:  root ← {segment ← " ", api_unit ← null, sub ← { }}
4:  for p in path.key( ) do
5:      # URI delimiter
6:      key_split ← p.split("/")
7:      if key_split.length( ) > 1 then key_split.pop( )
8:      end if
9:      if not root.sub.keys( ).contains(key_split [0]) then
10:         current_node ← {segment ← key_split [0], api_unit ← null, sub ← { }}
11:         root.sub.put(key_split [0], current_node)
12:     else if then
13:         current_node ← root.sub.get(key_split [0])
14:     end if
15:     api_exist_node ← current_node
16:     for k in key_split.pop( ) do
17:         tmp_node ← {segment ← k, api_unit ← null, sub ← { }}
18:         if not current_node.sub.keys( ).contains(k) then
19:             current_node.sub.put(k, tmp_node)
20:         end if
21:         current_node ← current_node.sub.get(k)
22:     end for
23:     api_exist_node.api_unit ← paths.get(p)
24: end for
```

4.1.3 Testing Group Generation

Given that APIs associated with the same resource often share a common URI path prefix and may possess logical relevance, iterating through the URI Path Segment Trie allows for the straightforward identification of API sets sharing the same URI path prefix through a sub-tree. This process further enables the filtering of potential testing groups from these sub-trees. A valid testing group is defined as a non-empty set of APIs related to the same resource. Such a group is a sub-tree that satisfies all the following criteria:

- The API sets mounted to the sub-tree nodes **must** be non-empty.
- The sub-tree root has mounted **at least** one API whose HTTP method is POST or PUT.

The DELETE method is not always being implemented by all kinds of resources. For example, generally, there is no *DELETE* method for logs, blockchain blocks, etc. Therefore, to generate more test groups, we search for object creation accordingly, *POST* or *PUT* method in the sub-tree root. In this way, testing groups are filtered from all the sub-trees from URI Path Segment Trie. So far, we can further filter target test groups for test case generation (*juno*, publicly available at https://github.com/restlogic/juno).

4.2 Code-Document Inconsistency Checker

In light of that coding and writing documents are two individual stages of software development, there is a possibility that the documentation may not always accurately reflect the code's behaviors. Therefore, such circumstances, also named code-document inconsistencies, are widespread in modern software development practice and significantly influence the process of software testing and maintenance. Typical code-document inconsistency scenes include:

- *Parameter name mismatch between different APIs*, labeled as ①. In the business backstage, resources are manipulated between APIs with different names.
- *The missing parameter in the request definition* is recorded as ②. The document lacks a complete and concrete description of API parameters.
- *Wrong URI path*, denoted by ③. The API described in the document is inconsistent with the actual code.
- *Parameter type mismatch with code* marked as ④. The declaration of parameter types in the document is inconsistent with the code, even if their names are consistent.

These above-mentioned inconsistency issues may degrade the availability and maintainability of REST services, and even cause security problems. The grey-box testing method we proposed can detect code-document inconsistency by crafting requests, sending them to the test server, and detecting inconsistencies according to the responses (*Mercurius*, publicly available at https://github.com/restlogic/mercurius). Detailed descriptions are shown as follows:

Request Crafting. By parsing the API document following OpenAPI Specification, we can detect missing parameter definitions of the request body schema, which were referenced in the request. With the help of *swagger-codegen*, we can generate the corresponding Software Development Kit (SDK) from the document. Then, fill the request parameters with dictionary-based values. At this stage, request payloads have been meticulously crafted and are ready to be dispatched to the testing server.

Request Sending. The sending stage is simple:

1. Provide the testing server address.
2. Obtain a token or session for authenticating.
3. Initialize SDK for creating an SDK client instance.
4. Call SDK function with crafted payload.
5. Collect responses.

Detecting. Based on the previous steps, we can conduct code-documentation inconsistency detection from the collected responses from the testing server by simply analyzing HTTP semantics (according to RFC 9110 [43]). Possibly, the analysis comes out with the following cases as shown in Table 2. For example, status code 20x means there are probably no inconsistency issues since the

request is successfully processed. When the request method is not *GET* and the returned status code is 30x, there possibly is a wrong URI path in the document leading to Redirection. Similarly, 400 indicates possible parameter missing in request payload or parameter type mismatching, 405 indicates that a non-existent method is defined which should be considered as code-document inconsistency, etc. We have detected many code-document inconsistencies and submitted patches.

Table 2: Code-documentation inconsistency check reference

Status code	RFC 9110 definition	Possible issues
20x	Successful	–
30x	Redirection	③
400	Bad request	② ④
401	Unauthorized	②
403	Forbidden	–
405	Method not allowed	③
5xx	Server error	① ② ③ ④

4.3 Runtime Information Acquiring

Our objective is to collect runtime intermediate information for detecting vulnerabilities that cannot be traditionally detected from analyzing REST API responses since runtime information may reveal another aspect of logic external expression. Therefore, we proposed a method for runtime information collecting on Python code through decorators [44], which were intended for applying the transformation to a function or method. Through this, we can insert data-collecting logic before and after the actual function logic, collecting the function's parameter and return values. Therefore, we can collect function parameters and return values with minimal source code changes.

OpenTracing client libraries provide APIs for configuring data storage server addresses, starting a tracing span by Python with scope, and sending traced spans to configured data storage server [45,46]. OpenTracing client works well with *Jaeger* [47], which is intended to monitor and troubleshoot microservices-based distributed systems, and also serves as the data storage server in a black-box mode, providing tracing data transmission over Internet Protocol (IP) network. To collect and store tracing data for further analysis, we only need:

1. Deploy a *Jaeger* server for data storage, and write down its address.
2. Install *jaeger_client* as OpenTracing library for Python.
3. Install our implemented tracing helper library which contains the decorator, which is designed for data collecting before and after invoking the actual function (*bees*, publicly available at https://github.com/restlogic/bees).
4. Add decorators to the functions intended to be traced (We propose *openstack/nova* traced via bees, publicly available at https://github.com/restlogic/openstack-nova).
5. Start the cloud service which has been traced.

We provide an implementation of a tracing decorator, in which, a decorator for tracing function is defined with the help of the OpenTracing client library. Note that:

1. *functools.wraps( )* is for updating wrapper function's attributes like *__module*, *__name__*, etc, making the wrapper function look like the wrapped tracing_function.

2. Pattern "$*args$, $**kwargs$" is for passing all function parameters to tracing_function as they were passed in the wrapper function. *args*, *kwargs*, and *ret_val* shall be reported to *Jaeger* when the scope closes.

Also, note that the tracing granularity can be adjusted as needed. In conclusion, we have presented a method, with the help of OpenTracing, to collect and store runtime information including function parameters, return values, and call stack structures, which shall be analyzed in the final stage.

5 PI-enhanced Model-Based Test Generation

5.1 Definitions of Vulnerability Models

In this section, we describe a typical logical vulnerability type we mainly discuss and support: Insecure Direct Object References (IDOR) [48]. It occurs when a cloud service trusts the user's input and provides direct object manipulation. Attackers can bypass authorization and reference any object by modifying the Identity document (ID) parameter in requests. We propose the following lifecycle model for request crafting: Assume pre-created *User A* and *B*. They should not access objects which were not created by themselves. There is an IDOR vulnerability when *User B* can access objects created by *User A*. Therefore, we can detect IDOR vulnerability by the following procedure:

1. Enable *User A* to create *Object X*.
2. Parse *Object X*'s ID from the response.
3. Referencing *Object X* by its ID, try to delete *Object X* using *User B*'s token.
4. Response collecting and analysis.
5. Post-testing clean up.

The same procedure can be displayed as intuitive as the following sequence diagram shown in Fig. 3.

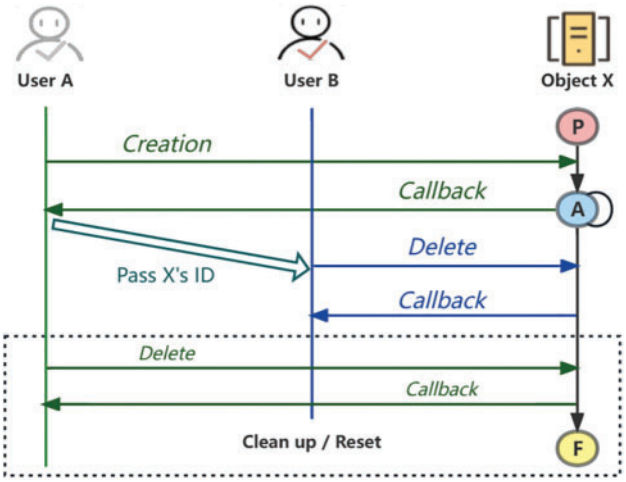

Figure 3: IDOR model procedure sequence diagram

5.2 Model-Based Tests Generation Method

Having been defined in vulnerability models, the testing procedure will be applied to all resources of the target cloud service, that meet all the requirements from the vulnerability model by analyzing

corresponding API Path Segment Trie or further test group, intending to detect whether the cloud service is being influenced. The generation of testing cases is strictly instructed by its corresponding vulnerability model. Taking the IDOR vulnerability model as an example, the detailed test case generation method is shown below as a demonstration:

- ***Initializing.*** In a test group, search for *POST* method API, get a parameter list from the corresponding SDK method, and fill parameter values based on a dictionary or samples. Initialize an SDK client by *User A*'s credential and then call that filtered *POST* method API for object creation.
- ***Parsing.*** i.e., ***parameter inference.*** Further search the test group for *DELETE* method API with a path parameter, which was supposed to be the ID parameter. To overcome inconsistency issues which lead to an inadequate test of logic, we calculate distances between the ID parameter and each parameter name from the response of creation from *initializing*, and select the most related one as inferred ID parameter since "inconsistency is possibly caused by careless coding or document writing, rather than completely irrelevance between API and docs". Pseudocode is shown in Algorithm 3, in which Lines 7–17 implement a classic algorithm to search the longest common substring (LCS) between two strings, Line 3 iterates between *path_name* and *tmp_name* within the parameter set to calculate the longest LCS and record its corresponding *tmp_name*, Line 21 filters all parameters and selects the parameter with the longest LCS as the inferred parameter.
- ***Logic violation.*** Initialize an SDK client by *User B*'s credential. Extract the ID parameter value from the response of object creation and fill it to the *DELETE* method's ID parameter. Note that when there is no response parameter matched with the latter ID parameter, fill in the inferred one via PI. After that, call the SDK method of *DELETE*.
- ***Response collecting.*** Collect response as well as runtime information from the testing server. Store them in a database for later analysis.
- ***Test resetting.*** Like step *logic violation*, but delete *Object X* with *User A*'s own token to reset the test environment so that subsequent tests will not be affected. Fill the ID parameter value to the *DELETE* method, and call the SDK method from *User A*'s client.
- ***Analyzing.*** In the analysis step, we propose a sparse call stack matching method to test whether the target call stack exists in collected runtime information. At the same time, we can simply discover the existence of vulnerability from response status codes. Details will be described in the following section.

As above, we realize a method to search APIs from the test group, infer ID parameter names, and call SDK APIs following the resource lifecycle test logic defined by the vulnerability model. Combined with the analyzing method shown below, we can measure the existence of logic vulnerability.

Algorithm 2: LCS-based parameter inference algorithm

Input: path parameter name *path_name*, parameter name set obtained from the response of creation *param_set*

Output: parameter name inferred to be the same with path_name *inferred_name*

1. inferred_name ← " "
2. param_lcs_dict ← { }
3. for *tmp_name* in *param_set* **do**
4. *c* ← *tr([path_name.len() + 1, tmp_name.len() + 1])*
5. Initialize all elements of matrix c to 0

(Continued)

Algorithm 2 (continued)

```
longest ← 0, lcs_set ← { }
for i in range(path_name.len( )) do
    for j in range(tmp_name.len( )) do
        if path_name[i] = tmp_name[j] then
            tmp ← c[i][j] + 1, c[i+1][j+1] ← tmp
            if tmp ≥ longest then
                longest ← tmp
                lcs_set.add(s.substring(i - tmp + 1, i + 1))
            end if
        end if
    end for
  end for
  lcs ← the longest element in lcs_set
param_lcs_dict[tmp_name] ← lcs
end for
inferred_name ← tmp_name with the longest lcs
```

6 CS-Assisted Tests Execution Analysis

Call Stack Information. Our proposed method detects specific function call sequences and analyzes the execute logic inside cloud services, which can detect logic vulnerabilities that cannot be discovered by simply analyzing the HTTP status code and response body. With the help of *Jaeger*, we can fetch all tracing data in a period from its Google Remote Procedure Call (gRPC) endpoint. Tracing data is collected by OpenTracing compatible client libraries, in which, custom labels are included.

Currently, our analysis is based on the Spans in Traces signal from OpenTelemetry, which is intended to describe what happens when a request is being processed by a cloud service. A span in OpenTelemetry is a unit of work or operation, that includes the following information:

- *Attributes*. Attributes are the key-value pairs that contain customizable metadata that can be used to annotate a Span, which also can be used to carry information about the operation it is tracking.
- *Span Status*. Span status is the field describing runtime exceptions. For instance, an error status will be set when a known error in the application occurs.
- *Parent Span ID*. Parent span ID is for nesting spans, especially for tracking call stack structural information, in other words, the caller and callee relationship between operations, in which case the value of the callee's Parent Span ID is set to be the caller's Span ID.

Afterward, we fetch all tracing data by invoking *Jaeger*'s gRPC API, then restore the call stack by spans' Parent ID.

Sparse Call Stack Matching. Since logic vulnerabilities generally show global effects rather than local ones, it is difficult to judge all the anomalies when merely focusing on local statements or feedback. Therefore we propose a sparse matching method of call stack to detect logic vulnerabilities, which is inspired by finite state machine defined for resource lifecycle (Section 4.1.1): call stack information also has its FSM features. It is common to match strings by finite state machine which can

be used to match a specific call stack by several criteria, which include: Span Name, Span Attribute, and Span Relationship.

A pseudocode describing the matching procedure is elaborated in Algorithm 3. In Line 2, δ equals to state-transition function, i.e., $S \times \Sigma \rightarrow S$, *m_sm. state* points to the current matching state, hence *rule* is the state-transition rule. Lines 3–12 iteratively matches the span with the *rule* as long as the span is not empty, and if a state in *m_sm* is matched, the matching rule will jump to the next state (Line 8 - 9). It's worth mentioning that .next in Line 11 defines the next matching span, in our method the direction is opposite (Line 1). In addition, δ can also be in the opposite direction, i.e., from *STOP* to *START*, keeping consistent with the matching direction. Finally, if the states in *m_sm* are all matched, the result will be output to prove that the call stack contains relevant patterns (Line 6).

Algorithm 3: Sparse call stack matching algorithm

Input: tail of span *span_tail*, state machine used to match with *m_sm*
Output: matching result *true* or *false*

```
1.  cur_matching_span ← span_tail
2.  rule ← m_sm.δ.get(m_sm.state)
3.  while cur_matching_span ≠ null do
4.      if or(rule matches cur_matching_span) then
5.          if matched_rule.transition_to = STOP then
6.              return true
7.          end if
8.          m_sm.state ← matched_rule.transition_to
9.          rule ← m_sm.δ.get(m_sm.state)
10.     end if
11.     cur_matching_span ← cur_matching_span.next
12  end while
13. return false
```

The rationality of sparse matching is that the logic is reflected in the global expression based on a certain degree of abstraction instead of a localized execution feature, no matter the normal one or the abnormal one, so it is not also necessary to follow the traditional mechanism to match with blacklist or whitelist. By adding constraints to the *Span Relationship*, sparse or compact matching can be achieved. Note that when the test sample is malicious and triggers the call stack sequence with a normal pattern, the logic defect can be detected directly according to the status code.

7 Evaluation

In this section, we mainly discuss and evaluate our proposed method around the following 3 research questions:

- RQ1: How much risk of inconsistency issues is faced by REST services?
- RQ2: How much coverage (target the test group) and the effectiveness of our generated test cases can we achieve?
- RQ3: What problems can be found, and which kind of problem is more suitable to be found by our method?

To answer RQ1, we conducted a comprehensive survey of the most popular open-source RESTful services (over 100 stars up to Aug 16, 2022) on GitHub and got a statistical conclusion, furthermore,

we selected several typical RESTful applications among them and analyzed their inconsistency issues. For RQ2, we set up a series of experiments on OpenStack, i.e., our SUT (system under test), and evaluated the coverage improvement for logic vulnerability analysis that our overall method can achieve compared with state-of-the-art. As for RQ3, we discussed the logic vulnerabilities and defects reproduced and newly found in our experiments, deeply analyzed the causes of these vulnerabilities and defects, and additionally discussed why our method can find them more efficiently.

7.1 Experiment Setup and Details of Dataset

All the experiments described in the following sections were carried out on a virtual machine Ubuntu 20.04.2 LTS (2.4 GHz 32-core CPUs and 32 GB RAM), deployed on QEMU emulator version 2.11.1 (Debian 1:2.11+dfsg-1ubuntu7.41) environment. The version of OpenStack we used is *Wallaby*, and the target language of instrumentation is Python, which is the programming language of OpenStack.

All our experimental datasets were sourced from real-world datasets. To assess the risks faced by current REST services, we analyzed popular projects on GitHub, retrieving REST service-related projects with over 100 stars. This was done to evaluate the compliance of the current REST service code and documentation. Furthermore, we selected several REST services with API documentation, analyzed the consistency between their code and documentation, and verified the effectiveness of the parameter inference method based on the LCS algorithm. Finally, we chose the open-source cloud platform OpenStack, which has numerous API calls and complex resource management tasks, as the testing subject for vulnerability testing methods.

7.2 RQ1: Inconsistency Issues

To systematically analyze the inconsistency risks and reveal the related problems faced by modern RESTful services and cloud operating systems, we conducted a series of manual analyses by analyzing the popular open-source repositories on GitHub. We utilized the keywords "REST service" and "RESTful service" and tags "rest-api" and "restful-api" for retrieving Java and Python repositories, sorted them according to the stars they received, and finally obtained 8 groups of statistical conclusions.

Statistical analysis. The statistics and analysis results are illustrated in Fig. 4. The bars from outside to inside are "total retrieved entries", "RESTful services entries", "RESTful services entries with API documents", "RESTful services entries with possibly manually-written API docs" and "RESTful services entries conforming to OAS/Swagger document specifications" respectively. According to the chart, our statistical conclusion shows that most of the retrieved entries are projects that produce or consume RESTful services rather than REST service itself, such as *vitalik/django-ninja* [49] and *microsoft/restler-fuzzer* [50]. Among them, about half of these REST services possess API documents, yet most of which are generated manually or do not declare the way they are generated, and merely a few of them are compliant with OpenAPI or Swagger specifications. Thus inconsistency issues are widespread in RESTful services, and testing these services will be a great challenge.

Inconsistency issues. Among the above entries, we chose several representative RESTful services and conducted our inconsistency-checking method on them. As shown in Table 3, "T-TG" means the proportion of our target test groups in total preliminarily filtered test groups. "CMN-Key" means the number of common keys found in the total keys acquired by analyzing the target test group. "CMN-Type" is the further analysis based on a common key, which means that there is probably no inconsistency issue if the parameter name and type are both the same. The results show that the

inconsistency issues are indeed widespread. Especially, OpenStack inconsistency issues submitted by us can be retrieved through the Issue Tracker ID, with IDs 805972, 807332, 807312, and 807315, respectively.

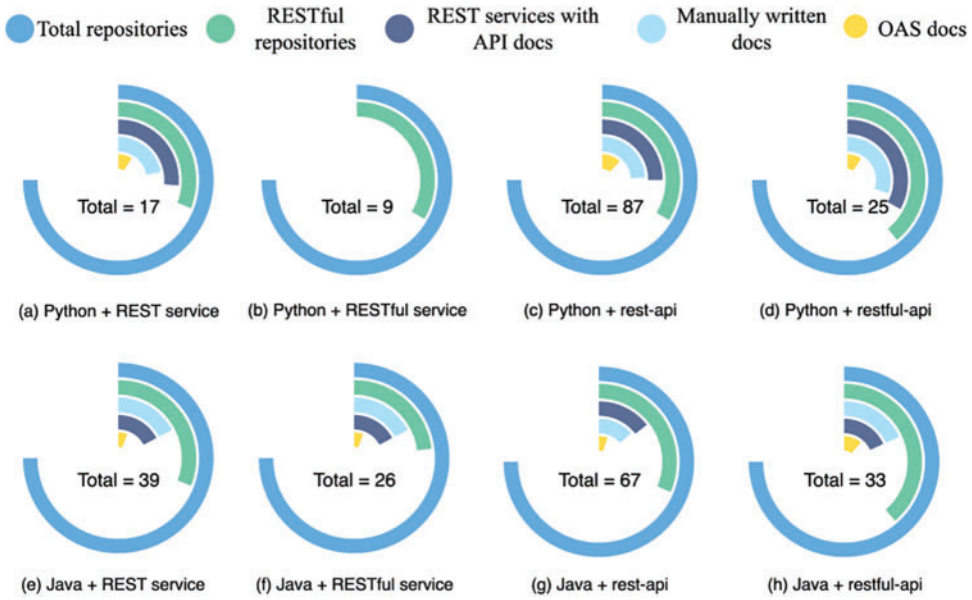

Figure 4: Github star repository statistic analysis results

Table 3: Code-documentation inconsistency checking results

Benchmark	RESTlogic			RESTler
	T-TG	CMN-Key	CMN-Type	
Gin-vue-admin	0–69	0–0	0–0	Ø
Go-gin-example	2–6	0–5	0–0	Ø
Harbor	1–3	0–0	0–0	Ø
Hydra	7–14	0–15	0–0	Ø
Petstore	3–7	3–11	3–3	Ø
OpenStack-nova	23–49	0–0	0–0	Ø

Selection of SUT. The characteristics of most of these open-source RESTful services in some aspects make them unsuitable for our experiments. For example, some repositories are services that provide basic query functions, such as *ExpDev07/coronavirus-tracker-api* [51], which do not have complex API operations, so they are not suitable for logical vulnerability detecting, but only for finding data processing exceptions. Thus, the above conclusion drives us to choose OpenStack, which has complex and massive resource operations, for our method validation.

7.3 RQ2: Coverage and Effectiveness

Our evaluation about code coverage was divided into two directions, including horizontal coverage focusing on APIs, and the vertical one concentrating on code. Test effectiveness as another evaluation criterion will be described subsequently.

The root cause of low coverage. As shown in Table 4, we list typical inconsistency issues in OpenStack and Petstore. Different APIs share the same resource in a test group but possess different parameter names, which leads to the dependency analysis (DA) of existing work being unable to accurately identify the relationship, further hindering the automatic testing procedure. For example, the parameter name of name returned by *POST /v2.1/os-keypairs* is inconsistent with the parameter *floating_ip_id* referenced by *DELETE /v2.1/os-floating-ips/{floating_ip_id}* but they point to the same resource, which is caused by careless coding or manual document writing. In this phenomenon, existing tools can still test the bugs that cause server errors by fuzzing the individual API, but cannot detect complex logic defects.

Table 4: Sample of inconsistency issues and the parameter inference results

Test group-test case	API 1 Request parameter	API 1 Response parameter	PI	DA
1. POST /v2.1/os-keypairs 2. DELETE /v2.1/os-keypairs/{**keypair_name**}	{‘keypairs_import_post_req’:{ ‘keypair’:{‘**name**’: ‘test-keypair-1’} ‘user_ud’: None, ...}	{‘keypair’:{‘fingerprint’: ‘1e:2c...’, ‘**name**’: ‘test-keypair-1’, ‘user_id’: ‘cedc88...’, ...}}	lcs: name	Ø
1. POST /v2.1/os-floating-ips 2. DELETE /v2.1/os-floating-ips/{**floating_ip_id**}	{‘floating_ips_create_req’:{ ‘pool’: ‘public1’ }}	{‘floating_ip’: {‘fixed_ip’: None, ‘**id**’: ‘9e4d0a51-...’, ‘pool’: ‘public1’, ...}	lcs: id	Ø
1. POST /v3/pet 2. DELETE /v3/pet/{**petID**}	{‘category’: ‘Category’, ‘**id**’: ‘1’ ‘name’: ‘dog’, ...}	{‘category’: ‘Category’, ‘**id**’: ‘1’, ‘name’: ‘dog’, ...}	lcs: id	Ø

API coverage. Since most modern REST API fuzzing technologies employ the blind fuzzing strategy and focus on detecting software quality defects, they generally perform better on the number of covered APIs. However, due to the unsolved inconsistencies, they generally cannot cover the complete API logic, i.e., the combination of APIs. As illustrated by the examples in Table 4, current tools, such as *RESTler* consider the 3 test groups as 6 individual APIs, therefore, the generated test cases were unable to meet the requirements of testing the API logic. In this background, existing tools can better realize the testing of a single API, while our method can take into account the logical integrity of the APIs, via test group, for better analyzing the API logic.

Code coverage. As for code coverage, we calculated it by tracking the called function methods with the help of our runtime intermediate information-acquiring approach. As Fig. 5 shows, we enumerate the filtered Nova’s APIs on the horizontal axis and calculate the case coverage depth for each function according to trace spans. We define the coverage depth for each function as the depth of the call stack from that function to the deepest function, in other words, trace span depth. Experimental results intuitively demonstrate the equivalent even higher coverage depth of our method achieved than

RESTler since our generated case considers more about resource lifecycle. In addition, PI effectively solves the problem that tests are discarded prematurely by the system, achieving beneficial effects as shown.

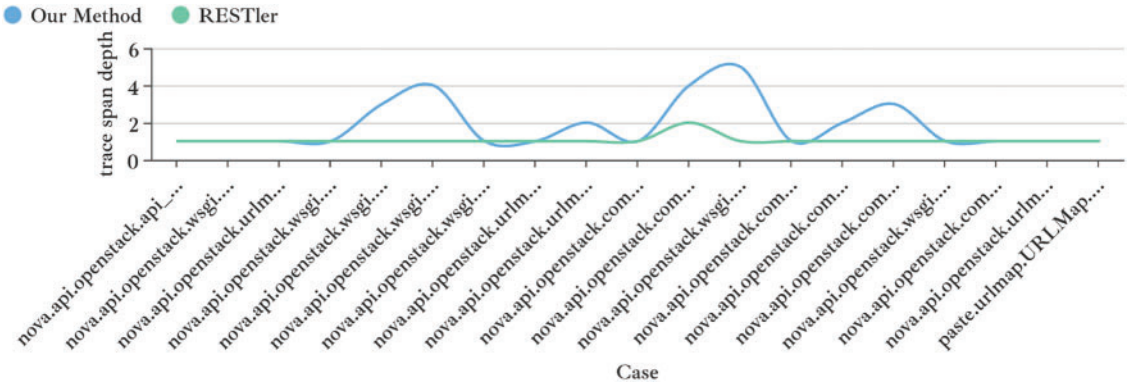

Figure 5: Comparative analysis of code coverage depth between our method and *RESTler*

Test effectiveness. As for assessing test effectiveness, the first way is to refer to the coverage in the previous paragraph, and the other is to evaluate the detection ability of the generated tests on logic defects. Compared with existing works in the field, we propose a test group filtering method, along with a resource-lifecycle/vulnerability model-based test generation method, remedying the problems of insufficient ability and strong contingency of existing methods to detect logical vulnerabilities. In general, our method is specific and effective for detecting logic vulnerabilities, and the test effectiveness will be illustrated by concrete detected logic defects analysis.

7.4 RQ3: Detected Logic Defects

Here we mainly introduce the differences between our method and existing works for detecting logic defects to prove our method's effectiveness. Comparative analysis is as follows:

Test Oracle. Most existing studies consider 5xx HTTP status codes as the target error for detection and define the combination of error code and error message as error type. Therefore, the main approach they detect target defects is by analyzing HTTP responses, and they consider 2xx as normal in most cases. However, in our approach, we define the legal resource lifecycle and its corresponding violations. When the elaborated test case triggers the call stack in common with the normal one, it proves that the malicious call sequence was successfully executed and there possibly exists a logic defect. Therefore, instead of focusing on 5xx errors, we can detect the possible logic defects hidden behind the 2xx status code.

Test Analysis. As shown in Fig. 6, an exposed logic defect caused by metadata service storing sensitive information was successfully detected by us, and it was made public as OpenStack Security Notes–0074 [52]. Following our abstraction about resource lifecycle and definition of the IDOR model, we generated a suite of test cases and got the call stack information as shown in Fig. 6a. According to our sparse call stack matching algorithm, we matched among 28 total spans and discovered an unexpected successful call stack pattern which contains the mainline spans sequence with depth of 3, *_format_instance_mapping* ← *get_metadata_by_instance_id* ← *MetadataRequestHandler*, from tail to head, whatever else function call is inserted in the middle of them. When there is no such vulnerability, access requests should be rejected in advance, and the call stack depth will be less than 3 (in our example, the legal call stack depth will be 2, i.e., call fails at *get_metadata_by_instance_id*). It is worth

mentioning that call stack analysis is also optional, since 202 Accepted triggered by logic-violated tests can also be agreed in advance that it should not occur. A newly detected API logical defect without calling stack analysis is shown in Fig. 6b, which was caused by loose default permission that assigns too high permissions to users so that they can delete or modify other users' resources beyond their authority, as shown in the highlighted lines.

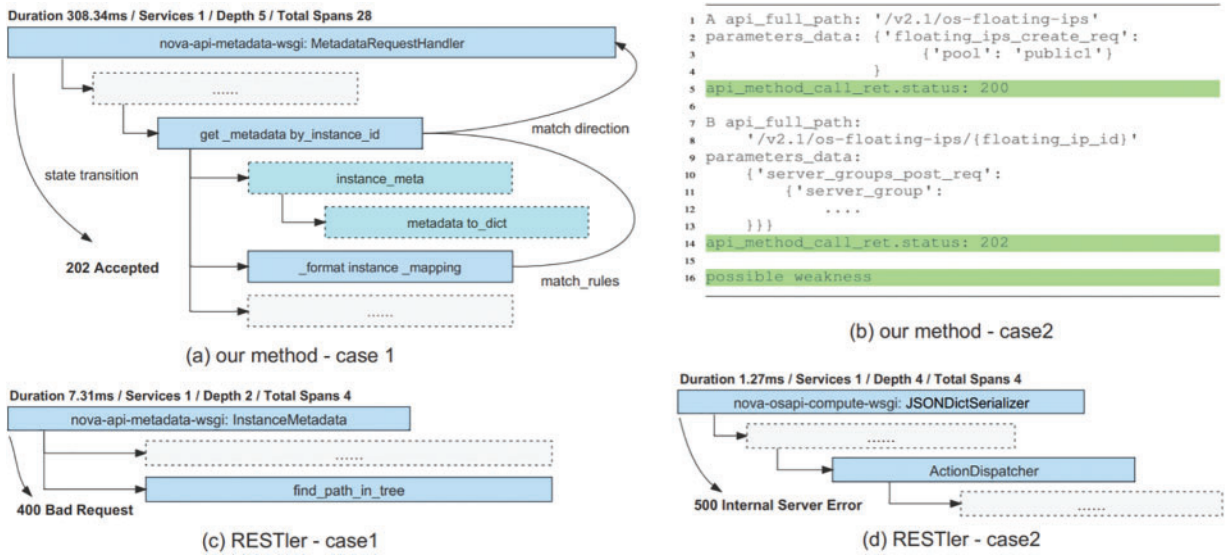

Figure 6: Sample defects detected by *RESTlogic* and *RESTler*, depicted by call stack analysis (For readability, we redraw the figure strictly based on tracing results acquired from *Jaeger* UI)

Compared with our method, existing work such as *RESTler* generates tests based on DA, but cannot suitably solve the inconsistencies till now. Therefore, when testing target services without well-organized API documentation, they may be incapable of testing complex logic defects, but they can still detect 5xx server errors if they are skilled. For example, the case from Fig. 6c is the one whose request syntax does not meet the demands of the server, the case from Fig. 6d successfully discovers 5xx server error, respectively. As we can see, they have a significant effect on testing a single API, but cannot obtain a longer call stack sequence, in other words, the test depth, hence lacking the ability to test complex logic.

Conclusion. Therefore, compared with existing works, our method not only offers an option to analyze logic defects by call stack analysis via sparse matching but also provides the ability to detect exceptions based on normal status codes combined with elaborated tests, hence showing its effectiveness in the specific field of logic defects detecting and analyzing.

8 Conclusion

In this paper, we propose a systematic and novel method RESTlogic for detecting logic vulnerabilities in cloud REST APIs. Firstly, we provide a REST API logic information manipulation method, revealing the logical relation between APIs, detecting the inconsistency issues, and acquiring runtime logic call stack expression. Secondly, we generate test cases guided by the resource lifecycle testing model and employ parameter inference to mitigate low testing coverage caused by inconsistencies. Thirdly, we propose a test execution analysis method combining call stack and status code analysis,

to discover the logic defects hidden behind 2xx status codes. Experimental results on REST services prove the efficiency and effectiveness of our method, and we found a series of inconsistencies and logic defects in open-source cloud REST APIs. In the future, we will introduce more intelligent methods, including artificial intelligence [53,54], and data mining [55], to guide the generation of tests and the detection of abnormal logic vulnerability event patterns [56,57].

Acknowledgement: The authors wish to acknowledge team members Mr. Debao Bu and Mr. Wenhang Bu for assistance in the experiment.

Funding Statement: The authors received no specific funding for this study.

Author Contributions: The authors confirm contribution to the paper as follows: Ziqi Wang and Weihan Tian contributed equally to this work; Study conception and method design: Ziqi Wang, Weihan Tian; experiment design and implement: Ziqi Wang, Weihan Tian; analysis and interpretation of results: Ziqi Wang, Weihan Tian; draft manuscript preparation: Ziqi Wang, Weihan Tian; topic selection, guidance, and supervision: Baojiang Cui; all authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data and tools that support the findings of this study are openly available at RESTlogic: https://github.com/restlogic.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] R. K. Samanta, B. Sadhukhan, H. Samaddar, S. Sarkar, C. Koner *et al.,* "Scope of machine learning applications for addressing the challenges in next-generation wireless networks," *CAAI Transactions on Intelligence Technology*, vol. 7, no. 3, pp. 395–418, 2022.

[2] R. S. Sharma, P. N. Mannava and S. C. Wingreen, "Reverse-engineering the design rules for cloud-based big data platforms," *Cloud Computing and Data Science*, vol. 3, no. 2, pp. 39–59, 2022.

[3] Amazon, "Shared responsibility model," [Online]. Available: https://aws.amazon.com/compliance/shared-responsibility-model/?nc1=h_ls (accessed on 16/08/2022).

[4] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[5] V. Atlidakis, P. Godefroid and M. Polishchuk, "Checking security properties of cloud service REST APIs," in *IEEE 13th Int. Conf. on Software Testing, Validation and Verification (ICST)*, Porto, Portugal, IEEE, pp. 387–397, 2020.

[6] J. Cable, D. Gregory, L. Izhikevich and Z. Durumeric, "Stratosphere: Finding vulnerable cloud storage buckets," in *Proc. of the 24th Int. Symp. on Research in Attacks, Intrusions and Defenses*, San Sebastian, Spain, pp. 399–411, 2021.

[7] S. Khasim and S. S. Basha, "An improved fast and secure CAMEL based authenticated key in smart health care system," *Cloud Computing and Data Science*, vol. 3, no. 2, pp. 77–91, 2022.

[8] Y. Chen, L. Xing, Y. Qin, X. Liao, X. Wang *et al.,* "Devils in the guidance: Predicting logic vulnerabilities in payment syndication services through automated documentation analysis," in *28th USENIX Security Symp. (USENIX Security 19)*, Santa Clara, CA, USA, pp. 747–764, 2019.

[9] V. Atlidakis, P. Godefroid and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *IEEE/ACM 41st Int. Conf. on Software Engineering (ICSE)*, Montreal, QC, Canada, 2019, IEEE, pp. 748–758, 2019.

[10] P. Godefroid, B. Y. Huang and M. Polishchuk, "Intelligent REST API data fuzzing," in *Proc. of the 28th ACM Joint Meet. on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering,*, USA, pp. 725–736, 2020.
[11] P. Godefroid, D. Lehmann and M. Polishchuk, "Differential regression testing for REST APIs," in *Proc. of the 29th ACM SIGSOFT Int. Symp. on Software Testing and Analysis*, USA, pp. 312–323, 2020.
[12] A. Arcuri, "RESTful API automated test case generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1–37, 2019.
[13] Swagger, "swagger-codegen," [Online]. Available: https://github.com/swagger-api/swagger-codegen (accessed on 16/08/2022).
[14] E. Viglianisi, M. Dallago and M. Ceccato, "Resttestgen: Automated black-box testing of restful apis," in *IEEE 13th Int. Conf. on Software Testing, Validation and Verification (ICST)*, Porto, Portugal, 2020, IEEE, pp. 142–152, 2020.
[15] M. Zhang and A. Arcuri, "Adaptive hypermutation for search-based system test generation: A study on rest apis with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–52, 2021.
[16] Z. Hatfield-Dodds and D. Dygalo, "Deriving semantics-aware fuzzers from web api schemas," in *Proc. of the ACM/IEEE 44th Int. Conf. on Software Engineering: Companion Proc.*, Pittsburgh, Pennsylvania, pp. 345–346, 2022.
[17] A. Martin-Lopez, S. Segura and A. Ruiz-Cortés, "RESTest: Automated black-box testing of RESTful web APIs," in *Proc. of the 30th ACM SIGSOFT Int. Symp. on Software Testing and Analysis*, Virtual, Denmark, pp. 682–685, 2021.
[18] Openstack, [Online]. Available: https://github.com/openstack/openstack (accessed on 16/08/2022).
[19] T. Rosado and J. Bernardino, "An overview of openstack architecture," in *Proc. of the 18th Int. Database Engineering & Applications Symp.*, Porto, Portugal, pp. 366–367, 2014.
[20] O. Sefraoui, M. Aissaoui and M. Eleuldj, "OpenStack: Toward an open-source solution for cloud computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, 2012.
[21] V. Felmetsger, L. Cavedon, C. Kruegel and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *19th USENIX Security Symp. (USENIX Security 10)*, Washington DC, USA, 2010. https://doi.org/10.5555/1929820.1929834
[22] G. Deepa, P. S. Thilagam, A. Praseed and A. R. Pais, "DetLogic: A black-box approach for detecting logic vulnerabilities in web applications," *Journal of Network and Computer Applications*, vol. 109, pp. 89–109, 2018.
[23] X. Li and Y. Xue, "LogicScope: Automatic discovery of logic vulnerabilities within web applications," in *Proc. of the 8th ACM SIGSAC Symp. on Information, Computer and Communications Security*, Hangzhou, China, pp. 481–486, 2013.
[24] M. Vanhoef, D. Schepers and F. Piessens, "Discovering logical vulnerabilities in the Wi-Fi handshake using model-based testing," in *Proc. of the 2017 ACM on Asia Conf. on Computer and Communications Security*, Abu Dhabi, United Arab Emirates, pp. 360–371, 2017.
[25] F. Sun, L. Xu and Z. Su, "Detecting logic vulnerabilities in E-commerce applications," in *NDSS*, San Diego, California, 2014. https://doi.org/10.14722/NDSS.2014.23351
[26] P. Cullum, "A survey of the host hypervisor security issues presented in public IAAS environments and their solutions," *2020 International Journal of Engineering Applied Sciences and Technology*, vol. 5, no. 8, pp. 36–48, 2020.
[27] R. Stultiens, "Compliant but vulnerable: fixing gaps in existing AWS security frameworks," M.S. thesis, Eindhoven University of Technology, Netherlands, 2020.
[28] J. Cable, D. Gregory, L. Izhikevich and Z. Durumeric, "Stratosphere: Finding vulnerable cloud storage buckets," in *Proc. of the 24th Int. Symp. on Research in Attacks, Intrusions and Defenses*, San Sebastian Spain, pp. 399–411, 2021.

[29] C. Marforio, A. Francillon and S. Capkun, "Application collusion attack on the permission-based security model and its implications for modern smartphone systems," in *Technical Report*, vol. 724, ETH Zürich, Department of Computer Science, 2011. https://doi.org/10.3929/ETHZ-A-006936208
[30] K. Lu, A. Pakki and Q. Wu, "Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences," in *28th USENIX Security Symp. (USENIX Security 19)*, Santa Clara, CA, USA, pp. 1769–1786, 2019.
[31] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware," in *NDSS*, vol. 1, pp. 1.1–8.1, San Diego, California, 2015.
[32] A. Martin-Lopez, "Automated analysis of inter-parameter dependencies in web APIs," in *Proc. of the ACM/IEEE 42nd Int. Conf. on Software Engineering: Companion Proc.*, Seoul, Korea (South), pp. 140–142, 2020.
[33] M. Zhang, B. Marculescu and A. Arcuri, "Resource and dependency based test case generation for RESTful Web services," *Empirical Software Engineering*, vol. 26, no. 4, pp. 76, 2021.
[34] Y. Liu *et al.* "Morest: Model-based RESTful API testing with execution feedback," in *Proc. of the 44th Int. Conf. on Software Engineering*, Pittsburgh Pennsylvania, pp. 1406–1417, 2022.
[35] S. Karlsson, A. Čaušević and D. Sundmark, "QuickREST: Property-based test generation of OpenAPI-described RESTful APIs," in *IEEE 13th Int. Conf. on Software Testing, Validation and Verification (ICST)*, Porto, Portugal, IEEE, pp. 131–141, 2020.
[36] M. Kim, Q. Xin, S. Sinha and A. Orso, "Automated test generation for rest apis: No time to rest yet," in *Proc. of the 31st ACM SIGSOFT Int. Symp. on Software Testing and Analysis*, Virtual, South Korea, pp. 289–301, 2022.
[37] Microsoft, "Best practices for cloud applications," [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design (accessed on 16/08/2022).
[38] H. Wu, L. Xu, X. Niu and C. Nie, "Combinatorial testing of restful apis," in *Proc. of the 44th Int. Conf. on Software Engineering*, Pittsburgh, PA, USA, pp. 426–437, 2022.
[39] OAI, "Openapi-specification," [Online]. Available: https://github.com/OAI/OpenAPI-Specification (accessed on 16/08/2022).
[40] L. Dusseault and J. Snell, "PATCH Method for HTTP," RFC 5789, 2010. [Online]. Available: https://www.rfc-editor.org/info/rfc5789 (accessed on 16/08/2022).
[41] R. Fielding and J. Reschke, "Hypertext transfer protocol (HTTP/1.1): Semantics and content, RFC 7231," RFC 7231, 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7231 (accessed on 16/08/2022).
[42] P. E. Black, "trie," [Online]. Available: https://www.nist.gov/dads/HTML/trie.html (accessed on 16/08/2022).
[43] R. Fielding, M. Nottingham, J. Reschke and H. T. T. P. Semantics, "HTTP Semantics," RFC 9110, 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9110 (accessed on 16/08/2022).
[44] Python, "Pep 318," [Online]. Available: https://peps.python.org/pep-0318 (accessed on 16/08/2022).
[45] Opentracing," [Online]. Available: https://opentracing.io (accessed on 16/08/2022).
[46] Opentracing api," [Online]. Available: https://github.com/opentracing (accessed on 16/08/2022).
[47] Jaeger," [Online]. Available: https://github.com/jaegertracing/jaeger (accessed on 16/08/2022).
[48] PortSwigger, "Insecure direct object references (idor)," [Online]. Available: https://portswigger.net/web-security/access-control/idor (accessed on 16/08/2022).
[49] Vitalik, "django-ninja," [Online]. Available: https://github.com/vitalik/django-ninja (accessed on 16/08/2022).
[50] Microsoft, "Restler-fuzzer," [Online]. Available: https://github.com/microsoft/restler-fuzzer (accessed on 16/08/2022).
[51] ExpDev07, "Coronavirus-tracker-api," [Online]. Available: https://github.com/ExpDev07/coronavirus-tracker-api (accessed on 16/08/2022).
[52] Robert Clark, IBM, "Nova metadata service should not be used for sensitive information," [Online]. Available: https://wiki.openstack.org/wiki/OSSN/OSSN-0074 (accessed on 16/08/2022).

[53] G. Hu and B. Yu, "Artificial intelligence and applications," *Journal of Artificial Intelligence and Technology*, vol. 2, no. 2, pp. 39–41, 2022. https://doi.org/10.37965/jait.2022.0102
[54] M. Sami, "Foundations of artificial intelligence and applications," *Journal of Artificial Intelligence and Technology*, vol. 2, no. 1, pp. 1–2, 2022. https://doi.org/10.37965/jait.2022.01
[55] J. Brieva, "Datamining and its applications," *Journal of Artificial Intelligence and Technology*, vol. 2, no. 3, pp. 77–79, 2022. https://doi.org/10.37965/jait.2022.0125
[56] M. K. Hooshmand and D. Hosahalli, "Network anomaly detection using deep learning techniques," *CAAI Transactions on Intelligence Technology*, vol. 7, no. 2, pp. 228–243, 2022.
[57] X. Zheng, Y. Zhang, Y. Zheng, F. Luo and X. Lu, "Abnormal event detection by a weakly supervised temporal attention network," *CAAI Transactions on Intelligence Technology*, vol. 7, no. 3, pp. 419–431, 2022.