Journal of Combinatorial Mathematics
and Combinatorial Computing
www.combinatorialpress.com/jcmcc

Combinatorial Press

RESTful API-based software interface testing techniques and common problems analysis

Fuju Sun[1,✉]

[1] *School of Information Technology and Intelligent Manufacturing, Shanghai Xingjian College, Shanghai, 200072, China*

ABSTRACT

Based on the common problems of the original fuzzy testing technique and the needs of RESTful API fuzzy testing, this paper proposes a white-box fuzzy testing method of REST API based on graph resource nodes for RESTful API software interface testing by using EvoMaster as a basic tool. The effectiveness of the fuzzy testing technique in this paper is analyzed. 21 apps with millions of downloads obtain more than 65,000 web request data and more than 8.5GB HAR files, and an average of 2,966 web request data is collected for each app. The REST interface filtering method of this paper's fuzzy testing approach effectively and accurately targets interface objects for fuzzy testing. The number of generated requests of the REST API white-box fuzzing test method based on graph resource nodes in this paper is much lower than that of other tools, and the efficiency of vulnerability discovery is much higher than that of other tools. The test method in this paper improves the number of lines of code covered in six hours by an average of 53.86% over other tools. The test method in this paper can identify more vulnerabilities and can cover all the vulnerabilities found.

Keywords: restful API, interface testing, fuzz testing, white-box fuzzing, validity testing

1. Introduction

Facing the challenges of the digital era, more and more enterprises are utilizing Application Programming Interfaces (APIs) to stay competitive [1]. APIs are not only suitable for digital scenarios such as Cloud Computing, Internet of Things, and Big Data Analytics, but also can help enterprises discover and maintain customers and build a new business ecosystem [2-3]. Representational State Transfer (REST) is a software architecture, a set of principles and constraints designed to provide a simple, lightweight, scalable and reliable way to build distributed systems and web services [4]. It emphasizes the design of an application as a set of resources, each identified by a unique identifier [5]. RESTful API is an API design style that conforms to the REST architecture, which enables efficient scaling and performance optimization by optimizing features such as client-server interaction,

✉ Corresponding author.
E-mail address: Shamysun@163.com (F. Sun).
Received 05 February 2024; Revised 12 April 2024; Accepted 05 December 2024; Published Online 16 April 2025.
DOI: 10.61091/jcmcc127b-386

statelessness, and good cache management [6]. Therefore, RESTful APIs are widely used in a variety of Internet applications, such as e-commerce, socialization, finance, and healthcare [7-8].

The widespread use of RESTful API technology also brings a large number of security issues. For individual users, it may lead to the leakage of personal information, such as usernames, passwords, contact information and other sensitive data [9-10]. This leakage not only violates the user's privacy, but also may lead to malicious identity theft triggering a series of chain reactions such as network fraud, resulting in economic losses and personal image damage [11-12]. For enterprises, the harm of API vulnerabilities goes far beyond short-term financial losses, but lurks in multiple far-reaching effects [13]. On the one hand, attackers often take advantage of such vulnerabilities to implement malicious operations, such as unauthorized fund transfers or precious data theft, which directly lead to substantial shrinkage of corporate assets [14-15]. On the other hand, data leakage incidents may not only lead to sensitive business intelligence or customer privacy falling into the hands of others, but also cause a lasting erosion of an organization's market competitiveness [16-17]. Once the API security vulnerability is publicized, the public image and brand reputation of the enterprise will be damaged, and the trust of users will be decreased. The intangible cost of this credibility depreciation often exceeds the measure of the loss of tangible assets, and poses an incalculable threat to the long-term development of the enterprise [18-21]. Therefore, by analyzing common problems with an application, testers can identify potential vulnerabilities or bugs and provide recommendations to the development team regarding fixes or improvements [22-23].

In this paper, out of the research on RESTful API, software vulnerabilities, traditional fuzzy testing techniques and their general problems, EvoMaster is used as the basic method for testing RESTful API interfaces, and a white-box fuzzy testing method for REST API based on graph resource nodes is designed based on the sampling and mutation of resource nodes. In order to accurately test the effect of the fuzzy test method proposed in this paper, firstly, the interface screening method is tested through 21 apps with millions of downloads. Secondly, the REST API white-box fuzzy test method based on graph resource nodes is compared with other tools for generating parameter validity, complex structural body variation and functionality to deeply analyze the effectiveness of the RESTful API white-box fuzzy test method proposed in this paper.

2. RESTful API fuzz testing

2.1. RESTful Web API

Representational State Transfer (REST), a software architecture style rather than a transport protocol, does not have any absolute standard, and any program written and designed to conform to the REST design style is RESTful.

In the concept of REST, application state and functionality can be categorized into various resources. A resource is an interesting conceptual entity that is exposed to the client. Examples of resources are: application objects, database records, algorithms, and so on. Each resource gets a unique address using a URI. All resources share a uniform interface for transferring state between the client and the server. Standard HTTP methods such as GET, PUT, POST, and DELETE are used. Hypermedia is the engine for application state, and resource representations are interconnected via hyperlinks.

Whereas, RESTful Web API is a series of services (APIs) publicly published based on the WEB through REST style design guidelines [24]. RESTful Web API architecture divides the server into two parts, front-end server and back-end server, the front-end server provides the user with a model-free view, and the back-end server provides the interface for the front-end server to access the data. The browser requests a view from the front-end server and initiates an interface request to obtain the model through an AJAX function contained in the view. The RESTful Web API architecture is shown in Figure 1.

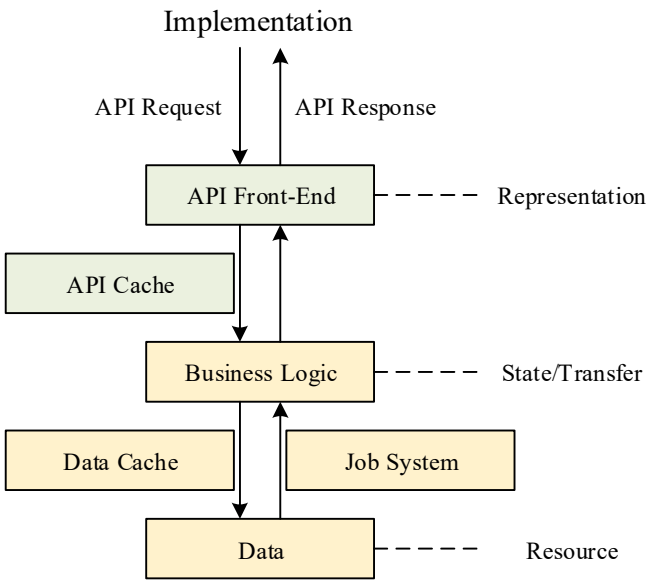

Fig. 1. RESTful Web API architecture

2.2. Software vulnerabilities

Vulnerability is a kind of security defect, commonly found in the hardware, software, and protocols of various systems, specifically manifested as flaws and deficiencies in the security strategy or implementation process. Attackers can use vulnerabilities to access the system or destroy the system without obtaining authorization and permission to achieve the purpose of malicious control system, which is a serious security threat to the system. Typically, attackers attack operating systems, application software and protocol software most commonly.

Software vulnerabilities can be classified from various levels and perspectives, classification can make testers can better grasp, this section focuses on three of the most common software vulnerabilities: buffer overflow vulnerabilities, formatting string vulnerabilities and directory traversal vulnerabilities.

2.2.1. Buffer Overflow Vulnerability. Buffer overflow vulnerability is the most common kind of software vulnerability [25], which exists in various types of operating systems and application software. In recent years, many security problems have been caused by this vulnerability, and the famous worms such as Shockwave and Shockwave have utilized this vulnerability to attack and destroy.

Simply put, the buffer overflow vulnerability is generated due to the lack of boundary checking in the program language itself, resulting in out-of-bounds access to data or pointers. Programs in the system need to apply for a certain amount of memory space to store data when running, and the size of the buffer space is fixed, when the actual input data exceeds the buffer capacity, a buffer overflow is generated.

The standard library of C is the source of most buffer overflow problems, especially for some string manipulation library functions. Depending on where the buffer is located in memory space, buffer overflow vulnerabilities can be categorized into heap overflow vulnerabilities and stack overflow vulnerabilities.

For buffer overflow, the direct result of the attack is to destroy the stack space, resulting in an exception or crash of the target program. Attackers can also use it to gain system privileges, and then carry out a variety of illegal operations.

2.2.2. Formatted String Vulnerability. Format string vulnerability exists in the formatting output function of the printf class [26], which is caused by not validating the user's input data. When certain

input and output functions (such as fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, etc.) are used as input data in a program to perform a formatting operation parameterized by “%s”, “%d”, “%n” and other formatting characters, you can output data from the stack or other memory locations in memory space, and you can also use formatting characters to write data to arbitrary memory addresses.

The formatting string vulnerability is very harmful. An attacker can use this vulnerability to display important data and information, and can arbitrarily destroy the program's memory space and allow the program to execute any malicious commands.

2.2.3. Directory Traversal Vulnerability. The reason for the directory traversal vulnerability is still that the program does not carefully verify the content of the input data, when the sequence of dots (the input data has the request directory appended ".. /" Or ".. /" or even its encoding, etc.), may cause the directory service to redirect to the default resource of this type of request, so that it is easy to obtain a complete list of existing resources in the program directory, resulting in the leakage of system resources.

2.3. Fuzzy testing

2.3.1. Overview of fuzzy testing. Fuzzy testing is a vulnerability mining method based on defect injection [27], the basic method of this technique is to inject malformed, unintended data into the application, device to be tested or target test system to mine software vulnerabilities. Comparatively, the research area of fuzzy testing is broader. From the classification criteria of software vulnerability discovery techniques, fuzzy testing mainly focuses on black-box testing and gray-box testing.

The effectiveness of fuzzy testing techniques relies on two basic assumptions: first, there are a certain number of vulnerabilities in the target software, and a certain number of input sets can trigger these software vulnerabilities; second, the triggering of software vulnerabilities can be manifested in visible ways, such as generating unexpected outputs, application crashes, or inability to complete normal functions.

2.3.2. Fuzzy Testing Basic Architecture:

1) Fuzzy Test Engine (Fuzzer). The fuzzy test engine is used to generate test cases, which is the core part of the whole fuzzy test technology architecture, and the effectiveness of the fuzzy test technology totally depends on the design of the fuzzy test engine. The collection of test cases contains malformed test data, which is different from normal test data and usually contains invalid data elements, unstandardized data syntax or carefully constructed fuzzy test values.

2) Test Interface. The acquisition of test interfaces is a necessary prerequisite for the implementation of fuzzy testing techniques. Fuzzy testing is an injective software vulnerability mining method, if there is no interface for test case injection, then the test cases cannot reach the test object. Further, if a stable data transmission channel cannot be established between the fuzzy testing engine and the test object, fuzzy testing cannot be carried out smoothly. Since the test cases of fuzzy testing are injected at the test interface, fuzzy testing can also be called interface robustness testing in a sense.

3) Test Objects. Depending on the location of the fuzzy test engine and the target software, the fuzzy test objects can be categorized into local fuzzy test objects, remote fuzzy test objects and memory fuzzy test objects.

4) Monitor. In the fuzzy test technology architecture, the monitor is a very important but often neglected part. The monitor monitors the operation of test objects during the execution of test cases, and is the only basis for judging the effectiveness of fuzzy test implementation, as well as an important means for improving fuzzy test analysis.

Despite the outstanding performance of fuzzy testing in vulnerability mining, it also has some problems and challenges, including: coverage problems, input generation problems, resource consumption problems, false positive and false negative problems, vulnerability exploitation problems,

etc. Therefore, this paper optimizes fuzzy testing of RESTful API software interfaces.

3. RESTful API white-box fuzzing test methodology

3.1. Description of the problem

EvoMaster is an open source automated web services fuzzy testing tool [28], which generates test cases based on evolutionary algorithms, it supports three types of APIs, REST, GraphQL and RPC, and it can perform white-box and black-box fuzzy testing for web services.EvoMaster is the only testing tool that supports both white-box and black-box testing as well as multiple EvoMaster is the only testing tool that supports both white-box and black-box testing and multiple API types.

EvoMaster usually randomly selects multiple chromosome templates to generate test cases when testing REST web services, and the combination of them has a large search space, which is usually difficult or time-consuming to generate meaningful test cases. To solve this problem, this paper designs a white-box fuzzy testing method for REST API based on graph resource nodes.

3.2. Description of the methodology

3.2.1. Description of basic definitions. Definition 1. For the chromosome template T, the definition is shown in (1):

$$T = \left(id, methodName, methodType, parameters \right) \tag{1}$$

For the chromosome template T, it is an abstract model of the operation requested by Evomaster, where id represents the unique identifier of the chromosome template; $methodName$ represents the name of the operation in the chromosome template; $methodType$ represents the type of the operation in the chromosome template, $parameters$ represents the is the set of parameters in the chromosome template, which includes input parameters and output parameters, the schematic diagram of the components of the chromosome template is shown in Figure 2.

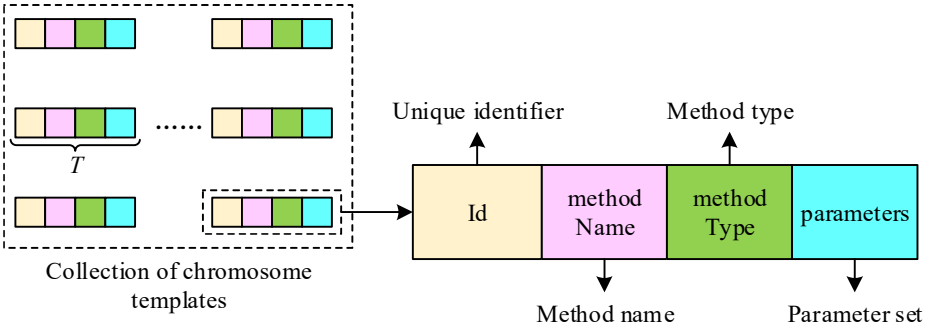

Fig. 2. Part of the composition of the chromosome template

Definition 2. For the test case Ind, the definition is shown in (2):

$$Ind = (T_1, T_2, \cdots, T_n) = (HTTP_1, HTTP_2, \cdots, HTTP_n) \tag{2}$$

For a test case Ind, containing an ordered series of HTTP requests, the test case generates multiple HTTP requests by instantiating a chromosome template T. In this paper a test case Ind generated based on a resource node is represented as a series of processing of a resource, and a test case generated based on a resource node represents a series of operations on a resource, all of which are performed via HTTP requests.

Definition 3. For the graph structure represented as $Graph$, which is defined as shown in (3):

$$Graph = (Nodes, Edges) \tag{3}$$

The graph $Graph$ is a directed graph, where $Nodes$ represents as nodes, which include resource nodes $ResNode$, ordinary nodes $Node$ and operation nodes $OpNode$, and $Edges$ represents unidirectionally connected edges between nodes.

Definition 4. for resource node $ResNode$, which is defined as shown in (4):

$$\mathrm{Re}sNode = (name, Object, Ts, SubNodes, Table) \tag{4}$$

For a resource node $ResNode$, where $name$ represents the name of the node, $Object$ represents the object defined in the Schema, which includes the name of the object and the associated fields and associated types. Ts represents a collection of multiple chromosome templates T, $Table$ represents the name of the corresponding database table on which $Object$ matches in the database, and $SubNodes$ is a collection of sub-resource nodes, which represents a collection of other resource nodes that can be traversed in one step through the resource node in the graph.

A resource node $Node_i$ exists in the $SubNodes$ of another resource node $Node_j$, which is reflected in the graph by the fact that through the $Object_j$ of this node $Node_j$ can be queried to get the object $Object_i$ of the resource node $Node_i$, i.e., there is an association between them, which is reflected in the database to represent that There may be associated foreign keys between them. Resource node means: the operation of a resource or a class of resources, nodes in the chromosome template Ts can be regarded as the resource of the addition, deletion, alteration and other operations, the reaction in the back-end of the Web service is the need to add, delete, alteration and other operations on the data in the database table $Table$.

Definition 5. for the ordinary node $Node$, which is defined as shown in (5):

$$Node = (name, Object) \tag{5}$$

For the ordinary node $Node$, its structure is similar to that of the resource node $ResNode$, compared to the resource node, which has only $name$ and $Object$, and does not have Ts, $SubNodes$ and $Table$.

Definition 6. For the operation node $OpNode$, it is defined as shown in (6):

$$OpNode = (name, Ts) \tag{6}$$

Operational node $OpNode$ is a node with node names $Name$ as $Query$ and $Mutate$, and Ts is a collection of multiple chromosome templates T in the operational node. The nodes in Fig. $Graph$ can be divided into operation nodes, ordinary nodes and resource nodes, and after dividing the operation nodes and resource nodes, the remaining nodes are ordinary nodes.

3.2.2. Resource node-based sampling and variation. In this paper, with reference to the RESTful resource-based sampling and mutation in Evomaster, four sampling methods are designed for resource nodes and sub-resource nodes, and the following are those four sampling methods:

1) Select a resource node $S1$ and generate test cases based on the chromosome templates in that resource node, each of which is a series of operations performed on the same or the same class of resources.

2) Select a resource node and a sub-resource node $S2$, generate test cases based on the chromosome templates in that resource node and sub-resource node, each test case is a series of operations performed on the same or the same class of resources and the corresponding sub-resources.

3) Select a resource node and multiple sub-resource nodes $S3$, generate test cases based on the chromosome templates in the resource node and multiple sub-resource nodes, each test case is a series of operations performed on the same or the same class of resources and the corresponding sub-resources.

4) Select a plurality of resource nodes $S4$ and generate test cases based on the chromosome templates in the corresponding plurality of resource nodes. Each test case is a series of operations performed on multiple resources.

This method corresponds to the hierarchical relationship between resources by dividing the child resource nodes under a resource node by the directed edges of that resource node on the graph structure.

The test case will be the above four sampling methods thus generating test cases, with a probability P to select one of the methods to generate test cases, the four sampling methods corresponding to the selection probability P is different, the sum of the selection probability of the four sampling methods is 1.

A resource node $S1$ is sampled and its selection probability is computed as shown in equation (7) below:

$$P(S1) = \frac{n_{resNode}}{n_{resNode} + n_{subNode} * K} \tag{7}$$

where $n_{resNode}$ is the number of all resource nodes in the graph structure, $n_{subNode}$ is the sum of the number of $SubNodes$ in all resource nodes, and K is a configurable weight. Then for the other three sampling methods, the formula (8) for the selection probability is shown below:

$$P(s_i, w_i) = (1 - P(S1)) \frac{w_i}{\sum_{j=1}^{S_{num}} w_j} \tag{8}$$

where s_i is the sampling method corresponding to $S2$, $S3$, $S4$, w_i is the weight of the method, $P(S1)$ is the probability of sampling a resource node $S1$, S_{num} is the value of all the other sampling methods except for \$S1\$, which has the value of 3 in the present method, $\sum_{j=1}^{S_{num}} w_j$ is the sum of the weights of the three methods.

This methodology is designed with four variant operations to change the structure of the test cases in order to exploit the relationships between resources and to explore the relationships between two or more resources:

(1) DELETE: which is categorized into two ways, deleting all the associated chromosome template-based generated HTTP requests within a corresponding resource node in a test case, and the other one deleting the HTTP requests generated by a particular chromosome template in a corresponding resource node.

(2) SWAP: which is categorized into two ways, exchanging the location of two HTTP requests in a test case, and the other is to choose between two test cases to exchange the corresponding resource node associated with the HTTP request generated based on the chromosome template.

(3) ADD: which is divided into two ways, in a test case to add the corresponding resource node within a chromosome template generated HTTP request, the other way is to add the corresponding resource node within all the associated chromosome template-based HTTP requests generated.

(4) REPLACE: which is divided into two ways, in a test case corresponding to a resource node within a chromosome template-generated HTTP request is replaced by another resource node-generated HTTP request, the other way is to correspond to the resource node associated with the corresponding resource node based on the chromosome template-generated HTTP request is replaced by another resource node-generated HTTP request.

A partial schematic of the four variant operations is shown in Figure 3.

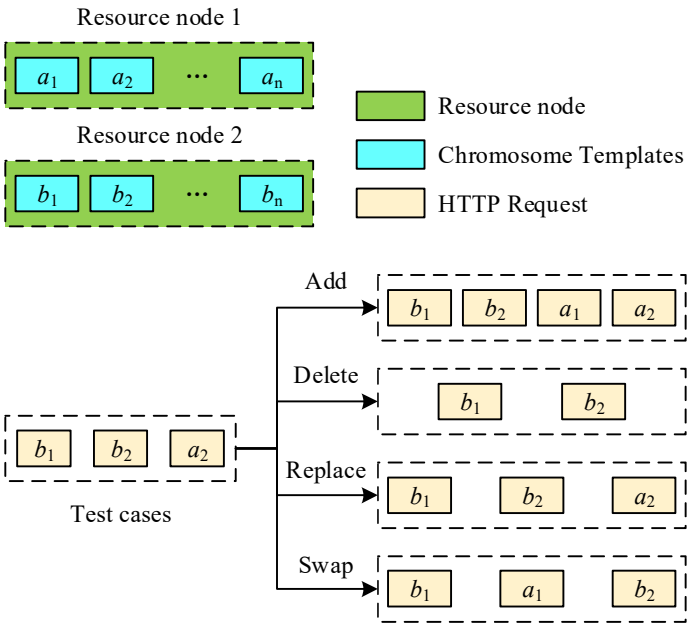

Fig. 3. Part of diagram of for mutation operation

4. Analysis of testing techniques

4.1. Evaluation of interface screening methods

In this section, an evaluation test is conducted to validate the fuzzy testing tool (white-box fuzzy testing method for RESTful API based on graph resource nodes) for mobile REST API interfaces. The fuzzy test is conducted on 21 apps of different categories with more than one million downloads sourced from the real application market utilizing the fuzzy test method proposed in this paper, to check whether it can be practically applied to the mobile applications in the real application market, and to effectively find out the robustness issues of the business server of the mobile apps.

Using the App Network Traffic Data Collector in the interface collection module of the test method in this paper and the REST interface screening method to perform a 10-minute Maxim automated test on 21 apps in different categories with downloads of more than one million in the mobile application

marketplace the results of the REST interface classification collected are shown in Table 1. In total, more than 65,000 pieces of web request data (8.5GB HAR file, average 2966 pieces of web request data collected per app) were obtained, and more than 20,000 business interface requests were obtained after classification and screening by the filtering method. It can be seen that using the designed REST interface screening method, a large number of unstructured interface requests with multimedia data and responses can be sifted out, and REST business interface requests with structured requests and responses are obtained, and the method is generalizable to different types of mobile apps in the real application market.

Table 1. App Dataset and Result of REST API Classification

App	Baidu	Taobao	JD	PDD	Meituan	Mafengwo	Ctrip
Category	Tool	Shopping	Shopping	Shopping	Shopping	Tourism	Tourism
HAR file size	760M	314M	115M	613M	310M	605M	284M
Request	2040	3756	2002	3218	4402	1833	3037
No return type	57	8	26	22	102	3	1128
Image	404	604	786	636	134	946	1235
Video	7	0	19	0	4	10	20
Binary data	0	0	5	0	0	3	0
Non JSON text	234	80	960	365	817	879	1336
Non JSON interface parameter	561	632	870	437	1476	616	1259
REST business interface request	1093	205	1012	872	1632	563	723
App	Tuniu	RED	Dianping	Eleme	NetEase CloudMusic	Bilibili	IQiyi
Category	Tourism	Social	Life	Life	Life	Video	Video
HAR file size	554M	156M	125M	147M	381M	675M	177M
Request	1991	4183	1814	3214	3517	2171	2209
No return type	156	32	25	625	385	45	75
Image	642	400	2040	931	2176	1119	504
Video	0	0	6	9	16	16	0
Binary data	9	15	22	0	6	8	1
Non JSON text	364	808	371	1550	799	1076	109
Non JSON interface parameter	412	306	982	1772	188	990	1760
REST business interface	386	451	1200	349	1234	1647	1171

request							
App	Youku	Douyin	Sina news	Sohu news	Toutiao	Tencent news	Hupu
Category	Video	Video	News	News	News	News	Read
HAR file size	878M	354M	147M	574M	828M	386M	323M
Request	4275	1691	2717	4373	3765	3423	2662
No return type	36	34	20	365	64	8	74
Image	1391	573	436	1309	667	1005	8167
Video	11	0	10	0	0	12	0
Binary data	3	4	6	0	0	900	7
Non JSON text	650	142	393	670	244	217	924
Non JSON interface parameter	2290	1109	1406	397	825	357	1512
REST business interface request	1183	752	771	1598	1534	1473	1394

The number of REST business interface requests screened in Table 1 contains multiple calls to different REST business interfaces, and the REST business interfaces that pass the screening will go through the APItrace generator to get the APItrace specification document of the interface's calls at the time of testing. In this step, an interface corresponds to the generation of an APItrace file, which can be understood as the multiple invocations of the interface are grouped and organized, and the number of APItrace files obtained is the actual number of REST business interfaces obtained by screening. We have counted and organized the total number of requests, the number of requests to REST business interfaces, and the number of REST business interfaces, i.e., the number of corresponding APItrace specification documents generated by the screened REST business interfaces for 21 different categories of mobile application network request traffic, and the results are shown in Fig. 4, which shows that, by using the REST interface screening methodology, we have greatly refined the fuzzy We can see that by utilizing the REST interface filtering method, we have greatly improved the accuracy of the target interface objects for fuzzy testing, and the number of REST business interfaces obtained by filtering out the interfering interface objects is on average less than 5% of the total number of requests before adopting the REST interface filtering method.

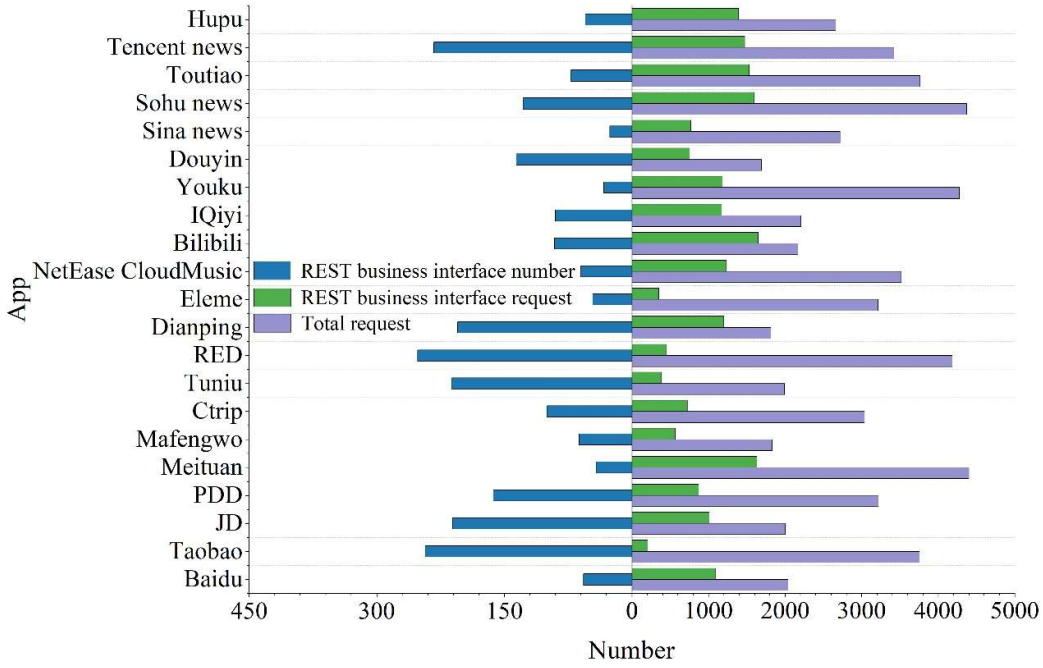

Fig. 4. REST API filtering result of App dataset

Similarly, we collected and counted the average number of requests for each mobile app's REST business interface.The average number of requests for REST business interfaces is shown in Figure 5. It can be seen that basically, each REST business interface can have an average of 0.8~38.9 calls in the network request data collected during the 10min test duration, and in more cases apps such as Youku and Meituan can even reach nearly 40 calls for one interface, which provides a reliable dataset for the subsequent interface parameter prediction module. At the same time, we found that the number of REST business interfaces for Taobao, Jingdong, Tuniu Travel, Xiaohongshu, Dianping and Tencent News are all over 200, and the excessive number of REST business interfaces may be due to the lack of precision in the screening method.

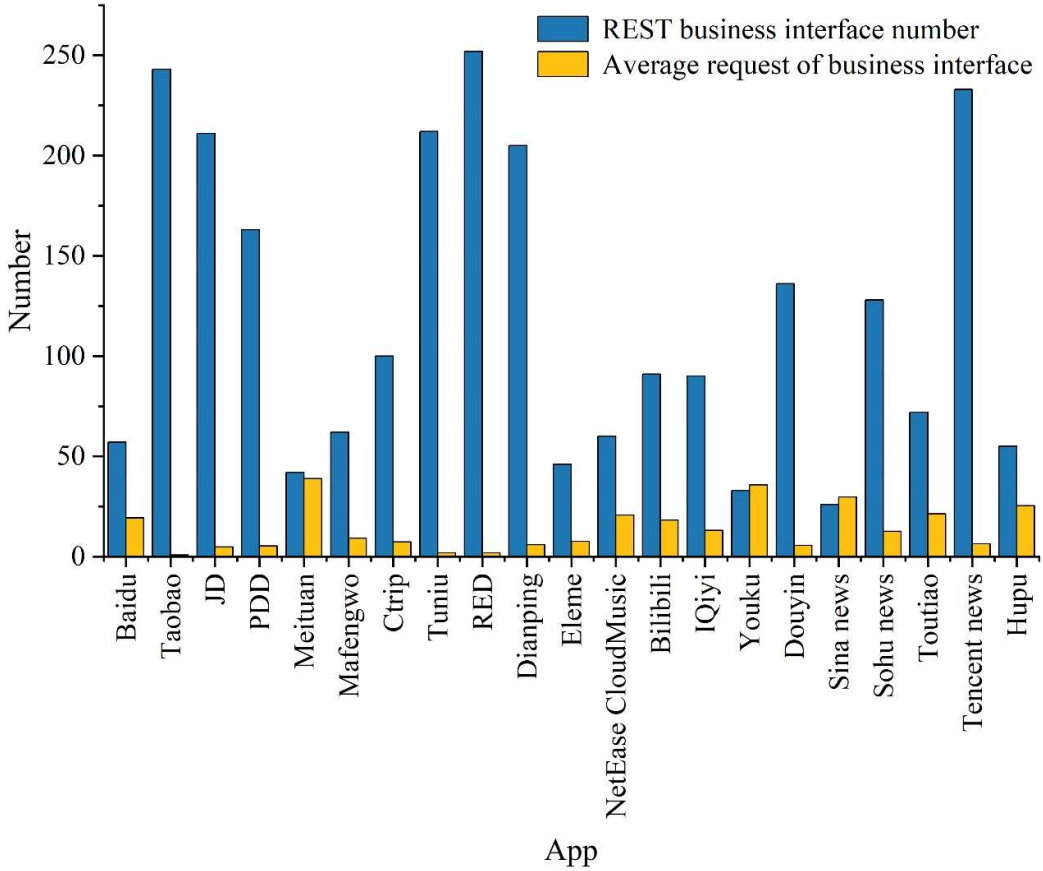

Fig. 5. Average call times of REST service API of App dataset

4.2. Comparative Experimental Analysis

4.2.1. Generating parameter validity analysis. For the effectiveness of the parameter generation strategy of the fuzzy test method in this paper, this section makes a comprehensive judgment by comparing the number of requests generated, the number of lines of code covered, and the number of errors found in this paper's test method with other tools within six hours. The experimental data is shown in Table 2.

As shown in Table 2, during the six-hour test, this paper's white-box fuzzy testing method for REST API based on graph resource nodes found more vulnerabilities than other testing techniques while generating an average of 49.93% fewer requests compared to other tools, which is a 157.14% increase in vulnerability discovery efficiency compared to Morest. This indicates that the test cases generated by this paper's method during fuzzy testing are more effective than other tools in detecting vulnerabilities and verifies the feasibility of the parameter generation strategy of this paper's fuzzy testing method in vulnerability identification.

Table 2. The number of requests generated by different tools in six hours and the number of errors

	RESTler		Morest		ZAP		W3AF		Ours	
	Generated request	Bug	Generated request	Bug	Generated request	Bug	Generated request	Bug	Generated request	Bug
JuiceShop	164253	1	98168	1	123523	2	132522	2	55243	4
NodeGoat	76853	0	57842	2	67458	1	75426	1	44824	3
Vampi	38468	1	17035	2	31256	0	18684	1	10388	4
Gitlab-projects	304153	2	113658	1	146285	1	176852	2	91064	4
WordPress	6745	1	7216	1	7078	1	6849	0	7582	3
Total	590472	5	293919	7	375600	5	410333	6	209101	18

4.2.2. Complex Structural Body Mutation Experiments. In order to measure whether this paper's fuzzy test method is able to accomplish variations of complex nested structs, this subsection compares the number of lines of code covered by this paper's method with four other tools on different web services in a six-hour test. The number of lines of code covered is a measure of whether a test case is able to cover every line in the source code. In API fuzzing tests, if the test cases are able to cover the branches, logical paths, and boundary cases of complex nested constructs in the source code, then the number of lines of code covered will increase accordingly. When the test cases generated by the API fuzzy tester are able to successfully mutate complex nested structures and these mutated test cases are able to trigger different code paths and logical branches in the target system, it will result in more lines of code being covered. Therefore, the increase in the number of lines of code covered is a direct indication that the API fuzzing tester is able to effectively mutate complex nested constructs.

The number of lines of code covered by different testing tools within six hours is shown in Figure 6. Within six hours, the test method of this paper is higher than all other tools in terms of the number of lines of code covered, with an average improvement of 53.86%, compared to RESTler, the improvement rate is as high as 90.88%, and compared to Morest, the improvement rate is 21.38%, which proves the effectiveness of the strategy of this paper.

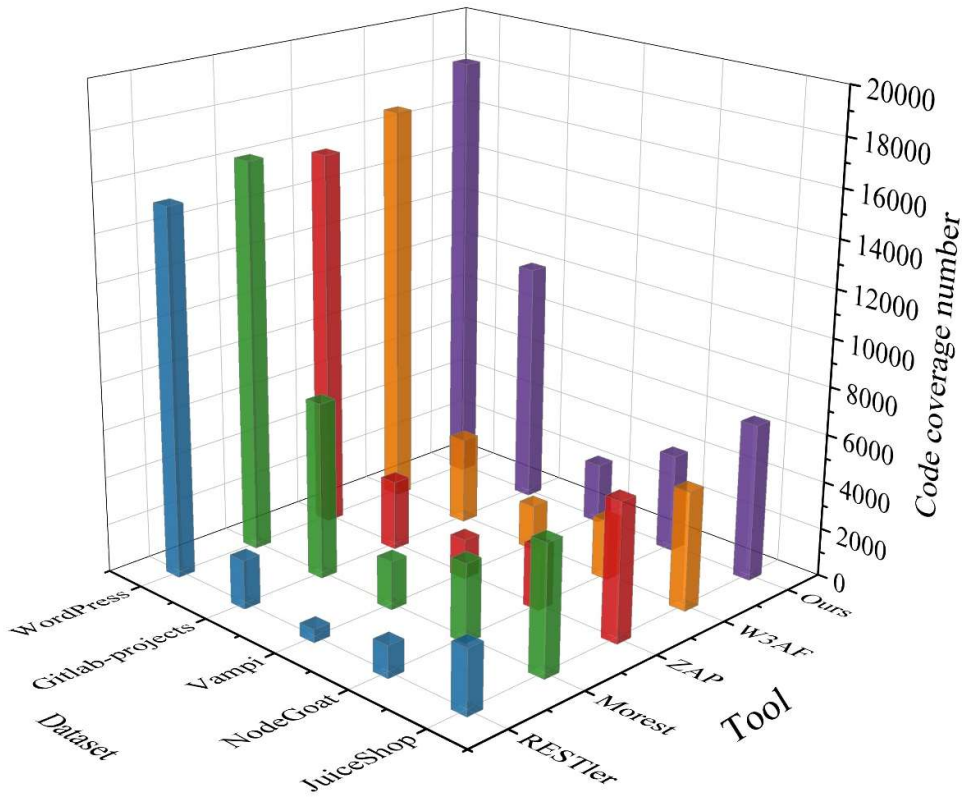

Fig. 6. The different test tools cover the number of lines in six hours

4.2.3. Functional experiments. Current testing techniques have challenges in generating effective long request sequences. Due to the inability to generate request sequences that are complex enough to trigger deep errors in APIs, the corresponding vulnerabilities cannot be detected. This limits the ability of automated testing to identify API vulnerabilities. In addition, existing techniques have difficulties in accommodating complex nested API requests. When requests contain other structures or use specific encodings, it is not possible to effectively mutate the structure and encoding of these requests to cover all possible vulnerability scenarios. Finally, existing techniques usually focus only on 5xx errors triggered during testing, without having specialized rule checkers for specific vulnerabilities. This means that some vulnerabilities may be missed because they do not necessarily lead to 5xx errors, but may produce anomalous behavior under other state codes.

In order to highlight the capability of the fuzzy testing approach of this paper in terms of vulnerability detection, a dedicated capability test was conducted and three test benchmark programs, NodeGoat, Juice Shop and VAmPI, were chosen. These benchmark programs were developed for educational purposes and contain known vulnerabilities, the number and type of which are defined. By testing against these vulnerabilities, the vulnerability detection capabilities of the methods in this paper can be better evaluated. The specific test results are shown in Table 3, where A~F represent authentication bypass vulnerabilities, directory traversal vulnerabilities, reflective XSS, stored XSS, SQL injection, and other injections, respectively.

The data below shows the number of unique vulnerabilities identified by different solutions. It can be seen that the fuzz testers used for comparison all perform similarly in terms of vulnerability testing. This is because they use the same payload dictionary and follow a similar testing strategy to identify vulnerabilities for each endpoint individually. In particular, ZAP and W3AF have better performance compared to the two Restful API testing solutions because they have built-in XSS execution detection modules to identify such vulnerabilities. In contrast, this paper's approach achieves significantly better performance in terms of both vulnerability types and number of vulnerability capabilities. The method in this paper is able to find different types of vulnerabilities including SQL injection, command

injection, XSS, and privilege management. On average, it identifies more vulnerabilities compared to other solutions. In particular, the method of this paper can cover all the vulnerabilities found by these solutions.

Table 3. Test comparison results

	JuiceShop						NodeGoat						Vampi					
	A	B	C	D	E	F	A	B	C	D	E	F	A	B	C	D	E	F
RESTler	0	0	0	0	2	0	0	0	1	1	1	0	1	0	0	0	2	0
Morest	0	1	0	0	1	1	0	1	1	0	2	2	1	0	0	0	1	0
ZAP	0	0	0	0	0	2	0	1	1	0	1	1	1	0	0	0	1	1
W3AF	0	1	1	0	1	1	0	0	1	1	1	1	1	0	0	0	1	1
Ours	2	2	1	2	4	3	0	1	2	4	3	3	4	1	2	3	4	3

5. Conclusion

The article designs interface testing methods for RESTful API software interfaces, improves them based on traditional fuzzy testing techniques and their general problems, proposes a white-box fuzzy testing method for REST API based on graph resource nodes, and tests the effectiveness of the method in this paper.

More than 65,000 network request data and more than 8.5GB HAR files are obtained from 21 apps with more than one million downloads on the mobile application market, with an average of 2966 network request data collected for each app.The REST interface filtering method greatly accurately targets the interface objects for fuzzy testing, and the average number of REST business interfaces is less than 5% of the previous number. The number of invocations of each REST business interface in the web request data ranged from 0.8 to 38.9 times.

The number of requests generated by this paper's white-box fuzzy testing method for REST APIs based on graph resource nodes is 49.93% lower than the average number of requests of other tools, and the efficiency of vulnerability discovery is greatly improved. The test method in this paper has the highest number of lines of code covered in six hours, which is 53.86% higher than the average of other tools, which shows the effectiveness of the strategy in this paper. The test method in this paper is able to identify more vulnerabilities and can cover all the vulnerabilities found by these solutions.

References

[1] Lamothe, M., Guéhéneuc, Y. G., & Shang, W. (2021). A systematic review of API evolution literature. ACM Computing Surveys (CSUR), 54(8), 1-36.

[2] Adnyana, I. K. W., Antara, I. G. M. Y., & Wulandari, D. A. P. (2021). Pemanfaatan Application Programming Interface Midtrans dan Raja Ongkir Untuk Membangun Enterprise Application Integration. Jutisi: Jurnal Ilmiah Teknik Informatika dan Sistem Informasi, 10(1), 13-22.

[3] Ofoeda, J., Boateng, R., & Effah, J. (2025). An institutional perspective on application programming interface development and integration. Information Technology & People, 38(2), 984-1016.

[4] Arianto, M. A. (2016). Analisis dan Perancangan Representational State Transfer (REST) Web Service Sistem Informasi Akademik STT Terpadu Nurul Fikri Menggunakan YII Framework. Jurnal Teknologi Terpadu, 2(2).

[5] Ong, S. P., Cholia, S., Jain, A., Brafman, M., Gunter, D., Ceder, G., & Persson, K. A. (2015). The Materials Application Programming Interface (API): A simple, flexible and efficient API for materials data based on REpresentational State Transfer (REST) principles. Computational Materials Science, 97, 209-215.

[6] Ehsan, A., Abuhaliqa, M. A. M., Catal, C., & Mishra, D. (2022). RESTful API testing methodologies: Rationale, challenges, and solution directions. Applied Sciences, 12(9), 4369.

[7] Li, L., Chou, W., Zhou, W., & Luo, M. (2016). Design patterns and extensibility of REST API for networking applications. IEEE Transactions on Network and Service Management, 13(1), 154-167.

[8] Neumann, A., Laranjeiro, N., & Bernardino, J. (2018). An analysis of public REST web service APIs. IEEE Transactions on Services Computing, 14(4), 957-970.

[9] Ali, S., Nasim, F., & Haider, K. (2025). SECURE MIDDLEWARE MODEL FOR PUBLIC RESTFUL APIS. Al-Aasar, 2(1), 50-62.

[10] Alharbi, S. J., & Moulahi, T. (2023). API Security Testing: The Challenges of SecurityTesting for Restful APIs. International Journal of Innovative Science and Research Technology, 8(5), 1485-1499.

[11] Khan, M. S., Siam, R. S. F., & Adnan, M. A. (2024). A framework for checking and mitigating the security vulnerabilities of cloud service RESTful APIs. Service Oriented Computing and Applications, 1-22.

[12] Wang, Y., & Xu, Y. (2024, September). Beyond REST: Introducing APIF for Comprehensive API Vulnerability Fuzzing. In Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses (pp. 435-449).

[13] Beer, M. I., & Hassan, M. F. (2018). Adaptive security architecture for protecting RESTful web services in enterprise computing environment. Service Oriented Computing and Applications, 12(2), 111-121.

[14] Gowda, P., & Gowda, A. N. (2024). Best Practices in REST API Design for Enhanced Scalability and Security. Journal of Artificial Intelligence, Machine Learning and Data Science, 2(1), 827-830.

[15] Zhang, M., & Arcuri, A. (2023). Open problems in fuzzing restful apis: A comparison of tools. ACM Transactions on Software Engineering and Methodology, 32(6), 1-45.

[16] Chaganti, K. C. (2024). Securing Enterprise Java Applications: A Comprehensive Approach. International Journal of Science And Engineering, 10(2), 18-27.

[17] Munsch, A., & Munsch, P. (2020). The Future of API Security: The Adoption of APIs for Digital Communications and the Implications for Cyber Security Vulnerabilities. Journal of International Technology & Information Management, 29(3).

[18] Zolotas, C., Chatzidimitriou, K. C., & Symeonidis, A. L. (2018). RESTsec: a low-code platform for generating secure by design enterprise services. Enterprise Information Systems, 12(8-9), 1007-1033.

[19] Joshi, N. Y. (2023). Developing Robust APIs and Web Applications for Enterprise Applications: Automation Frameworks and Testing Strategies. Journal of Basic Science and Engineering, 20(1).

[20] Kaul, D., & Khurana, R. (2021). AI to detect and mitigate security vulnerabilities in APIs: encryption, authentication, and anomaly detection in enterprise-level distributed systems. Eigenpub Review of Science and Technology, 5(1), 34-62.

[21] Marculescu, B., Zhang, M., & Arcuri, A. (2022). On the faults found in REST APIs by automated test generation. ACM Transactions on Software Engineering and Methodology (TOSEM), 31(3), 1-43.

[22] Li, H., Li, J., Wang, Y., Zhou, C., & Yin, M. (2023). Leaving the Business Security Burden to LiSEA: A Low-Intervention Security Embedding Architecture for Business APIs. Applied Sciences, 13(21), 11784.

[23] Padmanaban, R., Thirumaran, M., Anitha, P., & Moshika, A. (2022). Computability evaluation of RESTful API using Primitive Recursive Function. Journal of King Saud University-Computer and Information Sciences, 34(2), 457-467.

[24] Kornienko D V,Mishina S V,Shcherbatykh S V & Melnikov M O. (2021). Principles of securing RESTful API web services developed with python frameworks. Journal of Physics: Conference Series,2094(3).

[25] Zheng Zhangqi,Liu Yongshan,Zhang Bing,Liu Xinqian,He Hongyan & Gong Xiang. (2023). A multitype software buffer overflow vulnerability prediction method based on a software graph structure and a self-attentive graph neural network. Information and Software Technology,160.

[26] Vakhrushev I.A.,Kaushan V.V.,Padaryan V.A. & Fedotov A.N. (2018). Search method for format string vulnerabilities. Proceedings of the Institute for System Programming of the RAS,28(4),23-38.

[27] Guoyun Duan,Hai Zhao,Minjie Cai,Jianhua Sun & Hao Chen. (2025). DFL: A DOM sample generation oriented fuzzing framework for browser rendering engines. Information and Software Technology,177,107591-107591.

[28] Andrea Arcuri,Man Zhang,Susruthan Seran,Juan Pablo Galeotti,Amid Golmohammadi,Onur Duman... & Hernan Ghianni. (2024). Tool report: EvoMaster—black and white box search-based fuzzing for REST, GraphQL and RPC APIs. Automated Software Engineering,32(1),4-4.