Beyond REST: Introducing APIF for Comprehensive API Vulnerability Fuzzing

Yu Wang
Tsinghua University
Beijing, China
wangyu18@mails.tsinghua.edu.cn

Yue Xu
PTLAB
Singapore, Singapore
TrustAI Pte.Ltd.
Singapore, Singapore
larryxu0226@outlook.com

ABSTRACT

In modern software development, APIs play a crucial role as they facilitate platform interoperability and serve as conduits for data transmission. API fuzzing has emerged to explore errors and vulnerabilities in web applications, cloud services, and IoT systems. Its effectiveness highly depends on parameter structure analysis and fuzzing request generation. However, existing methods focus more on RESTful APIs, lacking generalizability for other protocols. Additionally, shortcomings in the effectiveness of test payloads and testing efficiency have limited the large-scale application of these methods in real-world scenarios.

This paper introduces APIF, a novel API fuzzing framework that incorporates three innovative designs. Firstly, by adopting a tree-structured model for parsing and mutating parameters in different API protocols, APIF breaks the limitations of existing research that are only effective for RESTful APIs, thus broadening its applicability. Secondly, APIF utilizes a recursive decoder to tackle the complex encodings in API parameters, increasing the fuzzing effectiveness. Thirdly, APIF leverages a testing priority calculation algorithm together with a parameter independence analysis algorithm to enhance fuzzing efficiency, enabling this method to be widely applied in real-world, large-scale API vulnerability fuzzing.

We evaluate APIF against the state-of-the-art fuzzers on 7 open-source projects via 412 APIs. The results demonstrate APIF's superior precision, recall, and efficiency. Moreover, in real-world API vulnerability exploration, APIF discovered 188 bugs over 60 API projects, with 26 vulnerabilities confirmed by the software maintainers.

CCS CONCEPTS

• **Security and privacy → Software and application security**;
• **Software and its engineering →** *Software defect analysis.*

KEYWORDS

API Fuzzing, Vulnerability Testing, Application Security, Web Security

RAID 2024, September 30–October 02, 2024, Padua, Italy

ACM ISBN 979-8-4007-0959-3/24/09
https://doi.org/10.1145/3678890.3678928

ACM Reference Format:
Yu Wang and Yue Xu. 2024. Beyond REST: Introducing APIF for Comprehensive API Vulnerability Fuzzing. In *The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), September 30–October 02, 2024, Padua, Italy*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3678890.3678928

1 INTRODUCTION

With the widespread application of API infrastructure, attacks exploiting vulnerabilities in APIs, such as those listed in the OWASP API Security Top 10 [9], may result in service crashes, unauthorized access, and sensitive data exposure. Consequently, testing for security vulnerabilities in APIs has become a critical component in enterprise API service development.

Typically, the generation strategies of an API fuzzer handle two main problems: 1) how to generate/insert fuzz vector and 2) how to generate proper test sequences. Early works [19, 20, 22, 29, 34, 45, 48] obtaining information from API declaration files, and using predefined fuzzing libraries [19], random values [22, 29], constraint-based methods [34, 45], and historical data [20] to generate test vector and sequence, aim to discover test vectors that cause exceptions in APIs. However, in these methods, less attention is paid to the contextual relationships of API parameters, resulting in the generated test vectors having low validity. To better adapt to the needs of API fuzzing, some new approaches have been proposed to improve the process of generating test vectors and test sequences. These include using test coverage [44, 46], dependency constraint solving [28, 37, 43], data-driven metrics [21, 27, 38, 42], graph-based analysis [39], and context-aware learning based on historical API communication data [39, 40]. In the industry, API vulnerability fuzzing tools [4, 8] have adopted the pattern of web vulnerability fuzzing, relying on single-packet testing, and traversing all parameters.

Regarding the existing research, we have identified three main shortcomings in the approach to black-box API vulnerability fuzzing:

- **Generalizability**: Previous studies have predominantly focused on RESTful APIs, demonstrating a lack of generalizability in their methods. As the API infrastructure evolves, the security issues of various new types of APIs cannot be overlooked, such as SOAP and GraphQL APIs in web applications, MQTT APIs in IoT systems, and gRPC APIs in cloud-native applications. We aim to employ a universal API vulnerability fuzzing framework that can adapt to multiple API styles.

- **Effectiveness**: Previous studies paid little attention to the complexity of API parameter structures and encodings. In real-world web services, many API parameters are encapsulated and encoded in multiple layers. For example, a `base64` encoded parameter value that, once decoded, uncovers an XML structure. By blindly altering API parameters, testing without decoding often leads to format errors in API responses, thereby obstructing the discovery of vulnerabilities.
- **Efficiency**: Previous research has inadequately addressed the practical application scenario wherein enterprise security teams frequently conduct large-scale scans of hundreds to thousands of APIs. Theoretically, in payload-based security testing, the number of requests sent equals the total number of parameters in the APIs being tested multiplied by the number of payloads in the payload list. This typically represents hundreds of millions of test requests and a significant amount of time consumption. Unoptimized fuzz testing tools may generate an enormous volume of requests, leading to the following issues: 1) Extremely low testing efficiency, making it challenging to discover vulnerabilities within the limited testing timeframe. For example, in pre-release security testing scenarios, security teams often have very limited time to conduct security assessments. 2) Even in isolated testing environments, the substantial computational and network I/O load generated by a large number of requests typically translates to higher testing costs. Moreover, in real-world API security testing, effective authentication measures are usually required to test a broader range of APIs. For instance, valid cookies and sessions must be configured to test APIs that are accessible only after user login. However, session lifetimes are typically limited, necessitating that the testing tools identify vulnerabilities as quickly as possible within the valid session period.

To overcome the above challenges, this paper introduces APIF, a novel black-box API vulnerability fuzzing framework. Our research aims to design and implement the most practical API vulnerability fuzzer. It will have broad applicability across various types of APIs, feature a simple underlying principle, and have high efficiency, to fit the needs of conducting vulnerability testing on large-scale APIs in the real world.

Our innovation is in three primary phases: First, API communication patterns are obtained from real API traffic, and then API encoding and parameter structures are parsed and systematically stored as a unified tree structure through recursive decoding. This innovative representation enables an in-depth understanding of complex API parameters, ensuring that test vectors can be injected under the API's encoding format, enhancing the test effectiveness. Furthermore, using a unified tree structure to represent and mutate the content of API parameters is applicable to all mainstream API types, not limited to RESTful APIs, thereby enhancing the generalizability of this framework. Second, for each API in the testing list, we calculate the likelihood of vulnerabilities using featured indicators such as the number of parameters, complexity of parameter types, and variety of request methods. APIs with higher vulnerability probabilities are prioritized for testing to improve overall fuzzing efficiency. This approach is particularly beneficial in scenarios where session lifetimes are limited, as it helps selecting testing paths with higher effectiveness within a given timeframe. Third, an independent analysis algorithm allows the mutations of multiple API parameters to be tested in a single request, thereby reducing network requests and enhancing fuzzing efficiency.

The effectiveness of APIF is demonstrated through the empirical evaluation of 7 projects via 412 APIs. APIF outperforms leading black-box API fuzzing tools in precision, recall, and efficiency. Additionally, APIF's application to 60 real-world projects led to the discovery of 188 bugs, including 26 vulnerabilities.

In summary, we make the following main contributions.

- We propose a new tree-structured API vulnerability fuzzing framework, featuring several innovative optimizations including 1) using a unified tree structure to represent and mutate API parameters to increase the framework's generalizability across different types of APIs, 2) employing a recursive decoder to address the complexity of API parameters and enhance test effectiveness, and 3) utilizing a testing priority calculation algorithm along with a parameter independence analysis algorithm to improve testing efficiency.
- We compare the performance of APIF against other vulnerability scanners and API testing tools on seven projects and demonstrate APIF's superior precision, recall, and efficiency.
- We have made APIF's tool implementation publicly available [1]. It discovered 188 bugs over 60 real-world API projects and identified 26 vulnerabilities with 6 assigned CVE IDs and 12 assigned CNVD IDs. We responsibly disclose those vulnerabilities to the maintainers and all of the vulnerabilities are confirmed.

The rest of this article is organized as follows. Section 2 introduces the background. Sections 3 and 4 constitute the core of this work, introducing our framework and the optimizations. Section 5 presents the experimental evaluation. Section 6 discusses limitations. Section 7 positions our approach against related work. Section 8 introduces future work and concludes this article.

2 BACKGROUND AND PRELIMINARIES

2.1 APIs in Morden Services

2.1.1 Web Service API. Traditional web applications often employ GET/POST methods to retrieve user-input data, where 1) GET requests are used for loading web pages or fetching resources; 2) POST methods submit form data, like user login credentials or payment information. On the more advanced spectrum, WebSocket APIs provide real-time, bidirectional communication between clients and servers, ideal for chat applications or live updates. Furthermore, GraphQL APIs offer a more efficient and flexible way to query and manipulate data, allowing clients to specify exactly what data they need. By exploring these different APIs, developers can identify and address a broader range of vulnerabilities, such as injection attacks, session hijacking, or data over-fetching/under-fetching in web applications.

2.1.2 Cloud Service API. Access to most cloud services is typically provided through RESTful APIs, which facilitate a variety of functions. In practice, different request types can prompt varied responses from a cloud service. For example, in a cloud computing platform, a client can 1) use the GET method to retrieve a list of

```
/console/{id}:
 get:
  parameters:
   -name: id
    Required: yes
    type: string
   -name: cmd
    Required: yes
    type: string
```

(a) API Specification

```
Generated_Request =
Request(
 Static("GET /console/")
 + Consumer("{id}")
 + Static("?cmd=")
 + Fuzzable("string")
 + Static("HTTP/1.1")
 + ...
)
```

(b) Request template

```
GET /console/admin?cmd=
    cat%20/etc/passwd
    HTTP/1.1
Host: www.test.com
...
...
...
...
...
```

(c) Testing request

```
200 OK
Server: nginx
...
root:x:0:0:root:/root:/
    bin/bash
daemon:x:1:1:daemon:/usr/
    sbin:/usr/sbin/
    nologin
...
```

(d) Response

Figure 1: Example of converting the RESTful API specification into a request template, and what a generated request and its corresponding response look like.

services they are currently using; 2) use the POST method to create virtual machine instances, containers, and databases; 3) use the PUT method to update resource information; 4) use the DELETE method to remove a specific resource. These actions allow for the exploration of the cloud service's states. By automatically generating and sending request sequences via a cloud service's REST API, a black-box testing tool can explore errors hidden in different states and discover vulnerabilities such as command injection, data leakage, and improper access management.

2.1.3 IoT System API. In the IoT sector, MQTT is a widely used lightweight messaging protocol for small sensors and mobile devices. It enables efficient and reliable transactions between IoT devices and the server. For instance, within an MQTT API, a client can 1) subscribe to topics to receive updates or sensor data from devices; 2) publish messages to a topic to send commands or configuration changes to the devices; 3) use Quality of Service (QoS) levels to ensure message delivery according to the required assurance; 4) utilize retained messages for persisting the last relevant message for future subscribers. These MQTT API operations are integral to the real-time, event-driven nature of IoT communications. By rigorously testing MQTT API messages and topic subscriptions, security tools can identify vulnerabilities such as improper message handling, insecure topic subscriptions, and potential eavesdropping risks.

2.2 OpenAPI Specification and Swagger

Typically, the owners of APIs publish accessible API declarations to guide users in their utilization. OpenAPI Specification (OAS) [7] is a widely recognized standard for describing RESTful APIs, offering a language-agnostic approach to empower both humans and machines to understand the capabilities of an API without direct access to its source code, thereby facilitating easier integration and consumption. Concurrently, users can leverage tools like Swagger [16], which utilizes the OAS description to automatically generate code for calling the API. This code can then be seamlessly integrated into their projects.

2.3 API Vulnerability Fuzzing

Existing API fuzzing approaches [4, 8, 22, 40] are proposed to explore errors hidden in the reachable execution states of an API service. Initially, the fuzzing tool reads the information from the OAS file, parsing essential details such as the API's access paths, authentication mechanisms, and parameter structures, and generates a request template with fuzzable parameters. Subsequently, it selects specific test vectors from a predefined fuzzing library, tailored to the different parameters included in the API. These vectors are used to insert or replace existing parameter values, creating requests that conform to the API's parameter structure. Finally, the tool sends the complete test requests to the target API, retrieves the response data, and compares this data against predefined response checkers. This process helps in determining the presence of API errors or security vulnerabilities. The main modules of an API fuzzer are as follows.

2.3.1 API Parameter Parsing. API fuzzing tools necessitate the parsing of the API's request message structure, followed by the construction of request templates for subsequent fuzz testing procedures. The process unfolds as follows: Initially, the testing tool is required to read the OAS file of each API, which encompasses critical details such as the API request path, request methods, parameter names, and input constraints, as illustrated in Figure 1a. To obtain the API specification, users have two primary methods: 1) manually defining the API's path and parameter structure by reading the API documentation published by the vendor on a web page, and 2) automatically parsing it from the publicly available OAS URL or Swagger file provided by the API supplier. Then, based on the specification, the fuzzer performs a static analysis to construct request templates as shown in Figure 1b. In the template, different types of variables are utilized to assemble a complete API request. Here, the `Static` type represents immutable strings that ensure the legality of the request message and are not subject to alteration. The `Consumer` type requires the input of specified data with contextual dependencies. Values of the `Fuzzable` type are designated for the subsequent phase of test vector generation. In this phase, these values are replaced or mutated with diverse test vectors to conduct the request.

2.3.2 Testing Vector Generation. In this module, the fuzzer assigns a value for each parameter in the request and constructs a complete request that is ready to be sent in a sequence template. As shown in Figure 1b, the request template `GET /console/{id}` contains 2 parameters, `{id}` and `cmd`, that need to be set. To generate a ready-to-use request, there are two methods to obtain a parameter value, which are 1) selecting an alternative value from a pre-defined vulnerability testing dictionary (e.g., SecLists [13]) or 2) reading a

target object from the response of a previous request [22]. As for the parameter `id`, the fuzzer will read a target object value from the response of another API, which provides a valid `id` object after creation. The generated testing request is in Figure 1c, which sets a command injection vulnerability payload `cat /etc/passwd` for parameter `cmd`.

2.3.3 Vulnerability Verification. After completion of the test request generation, the fuzzer dispatches the request to the target API and examines its response to determine the presence of any vulnerabilities. For instance, as shown in Figure 1d, the API's response includes content from the `/etc/passwd` file (a Linux system file path), which indicates that the command `cat /etc/passwd` was successfully executed and suggests a command injection vulnerability.

By utilizing the above main modules, existing API fuzzers automatically generate requests to test different services via their APIs. However, they still have limitations in parameter parsing and request generation phase, resulting in slow state exploration progress, which is the main focus of this paper.

3 DESIGN AND IMPLEMENTATION

3.1 Challenges in API Vulnerability Testing

3.1.1 API Parameter Structure Parsing. Fuzzer first needs to understand the parameter structure of the API before it can perform mutation testing on different API parameters. The process of parameter parsing includes two issues:

- **Generalizability**: Existing research [20–23, 27–29, 32, 34, 38–40, 42, 45, 46, 48] is only applicable to RESTful APIs and cannot address other scenarios mentioned in Section 2.1, including API types such as SOAP and GraphQL in web applications, gRPC in cloud applications, and MQTT in IoT systems. The parameter structures of APIs with different styles and protocols have similarities (Figure 2), allowing us to abstract a unified expression method that enables fuzzing techniques of RESTful API to be widely applied to more scenarios.
- **Parameter Encodings**: Another challenge is recognizing complex encoding structures and parameter relationships within APIs. This ensures that test vectors accurately target a specific parameter without interfering with the functionality of others. For instance, common APIs encapsulate parameters in HTTP request bodies utilizing formats like JSON, XML, and various array-type objects that include nested encoding within the values of parameters.
  As illustrated in Listing 1, the `pfile` parameter value is in `base64` encoding, and the `info` parameter contains XML-structured data. Without proper encoding recognition and parameter parsing, blindly altering the content in a message for fuzz testing leads to format errors in API responses, hindering the discovery of vulnerabilities.

3.1.2 Testing Vector Generation. Testing APIs in a reasonable order within the same application is crucial. API parameters often depend on each other, where a value string in response data from one API might be a valid request parameter value in another API. Directly injecting data without considering these dependencies can result in meaningless outcomes and generate numerous invalid test requests, adversely affecting the efficiency of fuzzing. Additionally, in some large-scale applications, to complete testing within a limited timeframe, it is essential to calculate the priorities of these APIs to enhance the efficiency of the testing process.

3.1.3 Priority Testing Path Selection. API security testing typically faces a large base number of APIs and the accompanying number of API call dependency paths. This often requires numerous payload injection attempts on a single API or a specific API call path, resulting in security testing results taking a considerable amount of time to converge. This issue is especially pronounced in scenarios where session lifetimes are limited, potentially allowing only a limited number of path tests to be completed before the session expires. By evaluating testing priority, we can prioritize testing of APIs and API call paths with higher probabilities of vulnerabilities within a given timeframe.

Listing 1: Encodings in the API request parameters

```
POST /user/update HTTP/1.1
HOST: 127.0.0.1

{
  "uid": 14175246,
  "timestap": 1668933016,
  "userdata":{
    "pfile": "dV8xNDE3NTIONi5wbmc="
    "info": "<?xml version=1.0 encoding=UTF-8?><
        note><age>18</age><name>jack</name><gender
        >male</gender></note>"
  }
}
```

Listing 2: Interdependence in API parameters

```
POST /api/forum/posts HTTP/1.1
Host: exampleforum.com

{
  "category": "Technology",
  "title": "The Future of Open Source",
  "authorName": "Alex Doe",
}
```

3.1.4 API Parameter Interdependency. API parameter fuzzing is a key method to detect API vulnerabilities, as any position within an API's parameters could potentially trigger a vulnerability. Typically, fuzzers change just one parameter with each API request, leading to massive testing requests and slowing down the fuzzing process. If we can identify the interdependencies of parameters, we can test multiple parameters at once in a single request, thereby enhancing the efficacy of the fuzzing process.

Listing 2 shows an API request for posting an article on a forum. When submitted, the system checks if the given `category` value already exists, returning an error message if it hasn't been found. Fuzzers testing both `category` and `title` parameters concurrently in a single request will find that test vectors of `title` are ineffective.

```
GET /console/command/
    get_status/12345 HTTP
    /1.1
Host: test.com
...
...
...
```

(a) RESTful API request

```
<soapenv:Body>
<con:GetConsoleCommand>
 <con:id>12345</con:id>
 <con:cmd>getStatus</con:
     cmd>
</con:GetConsoleCommand>
</soapenv:Body>
```

(b) SOAP API request

```
query {
  getConsoleCommand(
    id: "12345",
    cmd: "getStatus"
  )
  {result}
}
```

(c) GraphQL API request

```
console/commands/get
...
{
  "id": "12345",
  "cmd": "getStatus"
}
...
```

(d) MQTT API request

Figure 2: Examples of API request messages for a query containing two parameters, `id=12345` and `cmd=getStatus`, under different API protocols.

Therefore, they require two separate requests to validate these parameters. However, since `title` and `authorName` are independent, fuzzers can make mutations for both in just one request.

3.2 Design of APIF

Our design goal is to create a practical API vulnerability fuzzing framework that exhibits enhanced generalizability, effectiveness, and efficiency. It can be broadly applicable across various API types, addressing the challenges of complex parameter encoding, sequence constraints, and parameter constraints to ensure the effective injection of test vectors. Moreover, it can prioritize test targets based on the likelihood of vulnerability occurrence. and concurrently mutate multiple parameters in a single request, minimizing the consumption of computational and network I/O resources, thus providing efficiency in real-world scenarios of large-scale, batch fuzzing of APIs.

Building upon these design objectives, we introduce APIF, a comprehensive black-box API vulnerability fuzzing framework. It enhances the fuzzing process in three ways. Firstly, it decodes encoded API parameter values and parses them to a unified tree-based structure. Secondly, it calculates the likelihood of vulnerabilities and identifies the dependencies across APIs to create a proper testing sequence. Thirdly, it checks the interrelations among API parameters to allow simultaneous injection of multiple test vectors in a single request. These improvements come from the shortcomings of existing API fuzzing works. The overall process is structured as Figure 3.

3.2.1 API Acquisition. The process begins by deploying a well-known proxy server MitmProxy [6] on the client side, acting as a man-in-the-middle to intercept API communication data. The intercepted message provides insight into the API structure and parameters, forming the basis for further analysis and testing within the APIF framework.

Contrasting with the analysis of API parameters through OAS files, parsing API parameter structures from real API communication traffic offers two main advantages: 1) The feasibility of obtaining API parameter structures via OAS depends on the type of API and the maintenance of API declaration files. In real-world scenarios, not all APIs have accessible declaration files. However, interactions for APIs of all protocols can be obtained through traffic capture. 2) By analyzing API traffic, we can observe not only the API's parameter structure but also acquire valid parameter values. This is beneficial in addressing the context-dependency issues between different APIs, thereby facilitating the determination of reasonable testing sequences and enhancing test coverage.

3.2.2 API Parameter Parsing. Our proposed API parameter parsing algorithm addresses the following two issues: 1) Different types of APIs have different protocols and parameter formats. To make the fuzzer more universally applicable, we need a unified data structure to represent API parameters. 2) As mentioned earlier, the decoding work of API parameter values affects the implantation of subsequent test vector generation and parameter mutation phase, which is important for the effectiveness of API vulnerability fuzzing.

We implemented parsers for extracting parameter content from API communications across multiple protocols. Initially, we determine the protocol of the API request message through protocol feature matching. We developed a protocol-type detection module based on feature recognition for common API communication protocols. For instance, RESTful APIs can be effectively identified by features such as URL patterns, version parameters, and the ACCEPT header. GraphQL APIs can be distinguished by their data structures and specific operation fields like "query," "mutation," or "subscription." SOAP APIs can be recognized through the XML data format and distinctive nodes such as Envelope, Header, Body, and Fault.

Subsequently, for various API protocols, we employ the corresponding parsing libraries to capture the parameters and their values. This process involves recursive decoding to organize the API parameters and values into a unified tree structure, as detailed in Algorithm 1.

The recursive decoder will attempt to decode parameters that are encoded, if the presence of structured objects (e.g., JSON, XML, array-type data, etc.) is detected, the recursive decoder transforms the encoded API parameter values in the intercepted API request parameters (Listing 1) into a tree structure (Figure 4). The parameter parsing algorithm can effectively convert all key-value parameters into a comprehensive tree structure. This facilitates a more in-depth analysis and understanding of the API's encoding and parameter structures, which is crucial for further vulnerability fuzzing. The significant innovation compared to other tools is that the injection of test payloads will be conducted at every node within the tree structure. This elevates the granularity of fuzz testing from the parameter level to each node within the structured objects of the parameter, enabling us to perform very deep fuzz testing and significantly increasing the likelihood of discovering previously hard-to-detect security vulnerabilities. Besides, it exhibits a certain

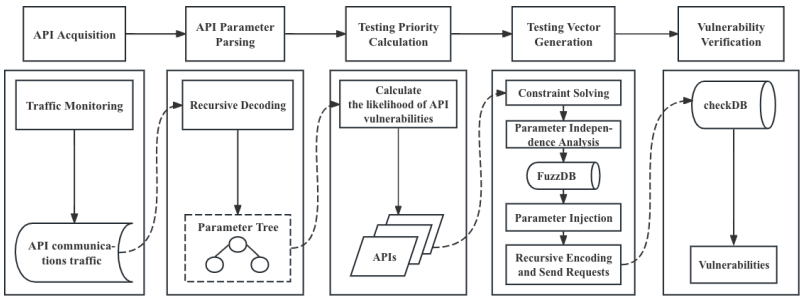


Figure 3: APIF API vulnerability fuzzing process.

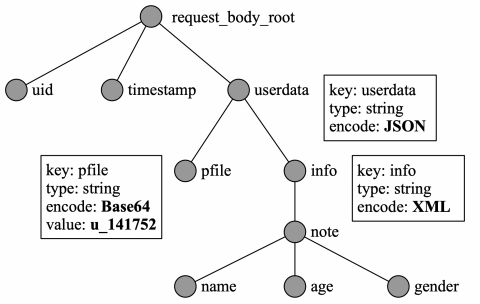


Figure 4: API parameters in tree structure.

degree of generalizability across different API communication protocols, which is more universally applicable compared to previous studies where a mutation template was generated for each target API (Figure 1b).

Algorithm 1 API Parameter Parsing Process

```
Input: Origin API request message request_data
Output: Tree-structured API parameters arg_tree
arg_tree ← PARSE_API(request_data)
function PARSE_API(request_data)
    arg_tree ← {}
    arg_tree.add_node(request_data)
    arg_tree ← RECURSIVE_DECODE(arg_tree)
    return arg_tree
end function
function RECURSIVE_DECODE(arg_tree)
    for node in arg_tree.child_nodes() do
        type ← GET_DATATYPE(node.value)
        if type in (XML, JSON, ...) then
            tree ← DECODE(node.value)
            node ← RECURSIVE_DECODE(tree)
        end if
    end for
    return arg_tree
end function
```

3.2.3 Testing Priority Calculation. APIF calculates the probability of vulnerabilities in APIs by utilizing API request elements. This calculation is used to prioritize APIs for testing and maximize the discovery of vulnerabilities within a set time frame, thereby enhancing testing efficiency. In this field, the common priority calculation methods of fuzz testing [44, 46] encounter significant issues when applied to APIs, such as 1) The factors leading to vulnerabilities in APIs differ from those causing errors; 2) Black-box testing makes it difficult to obtain detailed test coverage metrics, such as code or function-level execution coverage. Drawing on the experience in API-related vulnerability mining and the verification of public vulnerability datasets, we identified three metrics for assessing the likelihood of API vulnerabilities:

- The more parameters a user can input, the greater the probability of vulnerabilities. This is because a significant portion of API vulnerabilities are caused by improper handling of user inputs.
- The more complex the parameter types in an API, the higher the likelihood of discovering vulnerabilities. Complex parameter types imply that there are queries and functionalities associated with various types of data in the API. The complexity of the datatype leads to an increased possibility of security vulnerabilities.
- The more request methods an API supports, the higher the probability of finding vulnerabilities. For example, an API access path that supports GET, POST, UPDATE, and DELETE typically involves more complex operational functionalities and code logic, frequently leading to risky entity operations, as well as permission issues.

APIF defines API testing priority by a fast risk-evaluate algorithm with three indicators. The higher these scores, the more complex the functionality of the API and the higher the probability of vulnerabilities occurring.

- **Parameter Coverage Rate.** This is calculated by dividing the number of parameters in a single API by the total number of parameters across all APIs in the test scope.
- **Parameter Value Coverage Rate.** Considering the variety in parameter types (e.g., `int`, `float`, `str`, `null`, `bool`), this rate is calculated by dividing the number of parameter value types in a single API by the total number of parameter value types in all APIs within the test scope.
- **Operation Method Coverage Rate.** For an API with operation methods like GET, POST, PUT, and DELETE, we calculate this rate by dividing the number of operation types in a single API by the total number of operation types in all APIs within the test scope.

To neutralize the impact of scale and distribution differences between calculation indicators, we employ the Z-score method [36] for data normalization. This method standardizes data based on the mean μ and standard deviation σ of the original dataset, aiming to unify different magnitude data onto the same scale.

$$x' = \frac{x - \mu}{\sigma} \tag{1}$$

The transformation function for the Z-score is represented above (1), where x represents each observation value of the calculation criteria. For instance, if the operation method coverage results in

a test API are $x_1, x_2, x_3, \ldots, x_n$ then after applying the Z-score formula, the new sequence for the operation method standard becomes $y_1, y_2, y_3, \ldots, y_n$, each with a mean of 0 and a variance of 1.

A linear regression [26] function is performed on these variables: vulnerability probability $h_\theta(x)$, operation method coverage rate x_1, parameter coverage rate x_2, and parameter value coverage rate x_3. We analyzed 206 API vulnerabilities in open-source systems reported in CVEs from February 2022 to June 2023 as our dataset. We calculated these three coverage rates and the associated probabilities of vulnerabilities to determine their weights. The results are presented as (2).

$$h_\theta(x) = 0.735x_1 + 0.461x_2 + 0.223x_3 + 0.551 \tag{2}$$

3.2.4 Testing Vector Generation. After calculating the vulnerability probabilities and confirming the priority of the APIs to be tested, APIF will use constraint solving method to generate test vectors that have a proper sequence to fit the contextual dependencies (Section 4.1). Subsequently, APIF will perform a parameter independence analysis, and try to mutate multiple parameters in one test to reduce the total number of tests and improve overall test efficiency (Section 4.2). After confirming the concurrent testing strategy for parameters, APIF will retrieve payloads from the predefined fuzzing library and conduct mutation based on the previously generated unified tree structure (Section 4.3). Finally, it will re-encode the payloads according to the parameter's encoding type and send the test requests (Section 4.4).

3.2.5 Vulnerability Verification. Each test vector corresponds to a specific verification method to determine the presence of security vulnerabilities in the current API and identify their types. We summarize these methods into three types:

- **Content Matching in Response Messages.** For instance, in the case of using a fuzzing vector for Cross-Site Scripting (XSS) vulnerabilities, we verify if the API response content includes specific strings like JavaScript payloads or injected DOM elements. Similar approaches are also applicable to vulnerabilities such as data exposure, file uploads, command execution, and various other types.
- **Verification Based on Response Status Codes.** During the constraint-solving phase for the API sequence, a valid request typically has a 200 status code. This model is advantageous for detecting vulnerabilities with distinct status codes. For example, if a response to a Denial of Service (DoS) attack vector returns status codes 503 or 504, which confirms the presence of such a vulnerability in the API.
- **Verification Based on Response Time.** A prime example is identifying SQL time-based blind injection vulnerabilities. Here, if the attack vector includes SQL's sleep function, we monitor changes in response times before and after the vector is embedded.

Based on the three verification methods above, we have delineated 13 common types of API vulnerabilities, which include: Data Exposure, Command Injection, Broken Object Level Authorization, File Read Vulnerability, Broken Authorization, Server-Side Request Forgery (SSRF), SQL Injection, Security Misconfiguration, Denial of Service (DoS), Improper Error Handling, Cross-Site Scripting (XSS), Content Security Policy (CSP) Not Implemented, and Unvalidated Redirects.

4 OPTIMIZATION OF TEST VECTOR GENERATION PHASE

4.1 Constraint Solving

It is necessary to test different APIs sequentially in the appropriate order. For instance, in an e-commerce payment system [15], after a product purchase order is created, it can be paid for. Therefore, we cannot directly initiate tests on the payment interface; instead, we must first create an order.

In APIF, we designed an algorithm that calculates a set of request sequences based on API traffic (Algorithm 2).

Algorithm 2 API Request Constraint Solving

```
    Input: All API request set req_set
    Output: Valid test sequence seq
 1: sequence ← CALC_SEQ(req_set)
 2: function CALC_SEQ(req)
 3:     K ← 1
 4:     seq ← []
 5:     while K < length(req) do
 6:         for i ← 1 to N do
 7:             for j ← 1 to M do
 8:                 if DEPENDS(req[i], req[j]) then
 9:                     seq.add(req[i], req[j])
10:                     seq ← RENDER(seq)
11:                 end if
12:             end for
13:         end for
14:         K ← K + 1
15:     end while
16:     return seq
17: end function
18: function RENDER(seq)
19:     for i ← 1 to N do
20:         req ← LAST_REQ(seq[i])
21:         for j ← 1 to L do
22:             newseq ← CONCAT(seq[j], req)
23:             resp ← SEND(newseq)
24:             if resp has no valid error then
25:                 seq.add(newseq)
26:             end if
27:         end for
28:     end for
29:     return seq
30: end function
```

The algorithm initially includes an empty sequence and considers a sequence valid if each request in it returns a valid response code, defined as any code within the 200 range. The algorithm iteratively calculates request sequences of increasing length, starting from $n = 1$. For each set of valid sequences of length $n-1$, it creates new sequences of length n by adding requests with satisfied dependencies to the end of each sequence. The function `render` checks if

all dependencies of a specified request are met. A sequence is valid if every dynamic object required as a request parameter is produced by a response represented earlier in the sequence. New sequences of length n are retained if all relations are satisfied, otherwise, they are discarded. If a dynamic object, used as a parameter in a subsequent request, is destroyed after that request, the algorithm detects this by receiving an invalid status code (outside the 200 range) when attempting to reuse the object and discard that request sequence.

4.2 Parameter Independence Analysis

Current API fuzzing methods typically mutate one parameter at a time to ensure comprehensive testing, leading to a high number of tests. To address this inefficiency, we developed a parameter independence algorithm. This algorithm analyzes inter-parameter correlations, identifying independent parameters of APIs. In black-box fuzzing, this allows for simultaneous mutation of multiple uncorrelated parameters in each request, significantly reducing the number of tests and improving vulnerability discovery efficiency.

For an API with n optional parameters in (3), assuming there are $P_1, P_2, \ldots, P_n$ valid payloads for each parameter and Q invalid payloads, a total of S tests are needed.

$$S = Q_n + \prod_{k=1}^{n} P_k \tag{3}$$

Typically, bugs (or vulnerabilities) are triggered by a single or a pair of inputs, bugs resulting from combinations of three or more factors are rare. Therefore, reducing test cases to the minimum effective input changes needed to trigger bugs is crucial. If each test case not only changes one parameter in the API, R test cases are generated in (4), substantially reducing the number of test cases.

$$R = Q_n + \sum_{k=1}^{n} P_k \tag{4}$$

Building on this, if changing parameters $n_1, n_2, \ldots, n_k$ results in a response structure identical to the original, and changing n_i individually results in a different response, subsequent tests can embed payloads for $n_1, n_2, \ldots, n_k$ simultaneously to further improve test efficiency. This algorithm is described in pseudocode in Algorithm 3.

For example, an API with three distinct parameters A, B, and C, having 3, 4, and 5 valid payloads respectively, would typically require $60(3 \times 4 \times 5)$ tests for comprehensive testing. Additionally, testing with invalid payloads, such as using a string for an `int` parameter, is necessary. Considering APIs usually transmit data in JSON format with basic types like `int`, `float`, `str`, `null`, and `bool`, this adds $15(3 \times 5)$ more tests for invalid formats. However, if changing parameters A and B individually results in the same response structure as the original, while changing C yields a different response, then in subsequent tests, payloads for A and B can be embedded simultaneously, reducing the total tests to $26(3{+}3{+}5{+}5{\times}3)$. This significantly lowers the number of tests required.

In the above process, the independence of parameter A and parameter B is analyzed and ensured through bidirectional validation. The algorithm, in the first loop, first mutates parameter A to obtain `resp`, and then, based on the already mutated parameter A, continues to mutate parameter B to obtain `resp1`. In the second loop, the algorithm first mutates parameter B to obtain `resp`, and then, based on the already mutated parameter B, continues to mutate parameter A to obtain `resp1`. Only when the structures of `resp` and `resp1` completely match in both loops do we mark parameters A and B as independent parameters and proceed with simultaneous testing in subsequent steps.

Algorithm 3 API Parameter Independence Analysis

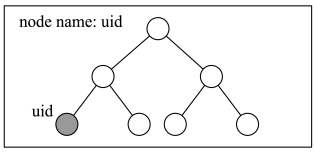
```
Input: API request parameters tree req_tree
Output: API parameters can be mutated in one request params
params ← CHECK_INDEPENDENCE(req_tree)
function CHECK_INDEPENDENCE(req_tree)
    params ← []
    args ← req_tree.iterate_nodes()
    for arg in args do
        flag ← True
        req ← MUTATE(arg)
        resp ← SEND_REQUEST(req)
        for arg_others in req_tree.nodes() do
            req_1 ← MUTATE(arg, arg_others)
            resp_1 ← SEND_REQUEST(req_1)
            if CHECK_SIMILARITY(resp, resp_1) then
                flag ← False
            end if
        end for
        if flag == True then
            params.add(arg)
        end if
    end for
    return params
end function
```

4.3 Parameter Tree Mutation

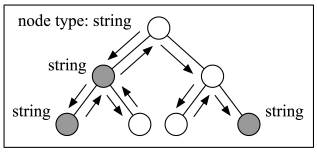
After the initial steps, parameters in the API test sequence are marked as fuzzable primitives. Test vectors are inserted into these primitives, replacing their original values, based on a user-configured vulnerability dictionary [13]. For API parameters identified as "independent" in the analysis phase, multiple testing vectors are embedded in a single test request.

APIF introduces four fundamental mutation strategies to enhance API vulnerability fuzzing, as illustrated in Figure 5. Firstly, it allows for the mutation of either the name or value of a specific parameter node. For instance, injecting a payload specifically into the `uid` parameter (Figure 5a). Secondly, the framework supports the traversal and mutation of all nodes satisfying specific filter criteria. An example is applying a command injection vulnerability test vector to all nodes with a `string` data type (Figure 5b). Thirdly, APIF facilitates the addition of new nodes to the request parameters, such as inserting a node named `admin` with the value `true` could help bypass permission checks (Figure 5c). Lastly, the deletion of nodes is also supported; removing parameters related to the identity might expose vulnerabilities in the authentication process due to inadequate validation policies (Figure 5d). These mutation methods are capable of identifying various types of API vulnerabilities and

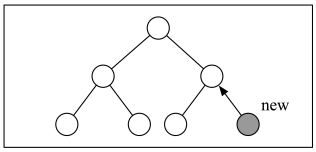
are adaptable to different API formats such as RESTful, GraphQL, SOAP, and gRPC, demonstrating a high level of generalizability.



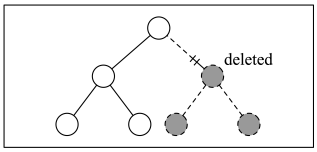
(a) Modify the value of a specific node.



(b) Modify all nodes that meet the specified criteria.



(c) Mutation by creating new leaf nodes.



(d) Mutation by deleting target nodes.

Figure 5: Parameter tree mutation methods of APIF.

4.4 Recursive Encoding and Send Requests

After replacing original values with auto-generated test vectors based on SecLists [13], we use a recursive encoder to re-encode the test vectors into their original format. This process restores the tree structure to the API communication format, altering only the parameter values of the fuzzable primitives into test vectors, while keeping the overall request structure and syntax intact. This step maximizes the consistency of parameter encoding and decoding, reducing the likelihood of parameter handling anomalies. Consequently, it increases the chances of payloads becoming effective. After encoding, the modified API requests are sent out, and the responses received then move on to the result verification phase.

5 EVALUATIONS AND RESULTS

5.1 Dataset

To validate our research, we selected 4 API vulnerability sandboxes widely used in API fuzzing approaches and 3 real-world API projects as fuzzing targets. This dataset not only encompasses the comprehensiveness of API vulnerabilities but also offers practicality in real-world environments.

The API vulnerability sandboxes are:

- crAPI [10]: A project aimed at helping testers understand key API security risks. It is designed to be easily attacked, providing a practical testbed for API fuzzers.
- vAPI [18]: Another API vulnerability playground simulates OWASP API Security Top10 [9].
- APISandbox [2]: It offers a broader range of API attack scenarios, including issues under 4A authentication systems, GraphQL-based message boards, classic API vulnerabilities, WSDL leaks, and unauthorized server access.
- VAmPI [17]: A collection of vulnerable APIs made with Flask, featuring vulnerabilities from the OWASP API Security Top10 [9], aimed at evaluating the efficiency of tools in detecting API security issues.

For these API vulnerability sandboxes, we obtained both the vulnerable APIs and the normal APIs from the project documentation as labels for positive and negative samples.

To measure APIF's performance in real-world API applications, we tested three open-source projects: Spree [15], GitLab-CE [5], and SilverStripe [14], which are widely used in real business environments. We obtained vulnerabilities from the official security updates for these three projects over the last three years, which serve as positive examples in our dataset.

The final dataset encompasses 7 projects with a total of 412 APIs and 112 security vulnerabilities. The types of these vulnerabilities were also cataloged as shown in Table 1.

5.2 Fuzzers

To validate our research, we compared APIF with 3 state-of-the-art API fuzzers:

- RESTler [12]: The first stateful RESTful API fuzz testing tool, designed to test errors in services through REST APIs automatically. It was later improved by researchers for the detection of security vulnerabilities [23] and is considered a representative research achievement in the field of API vulnerability fuzzing.
- Fuzzapi [4]: The most popular API vulnerability fuzzer on GitHub, allows security experts to discover vulnerabilities in APIs by conducting fuzz tests with various attack payloads, and is widely applied in real-world vulnerability testing scenarios.
- OpenAPI-Fuzzer [8, 30]: Another popular open-source API vulnerability fuzzer in the industry that has already discovered several API vulnerabilities in public systems such as Kubernetes, Vault, and Gitea.
- APIF-A: The complete technical implementation based on this APIF framework, developed in Golang. It includes API parameter tree structure parsing, vulnerability probability calculation, and concurrent testing based on parameter interdependency analysis.
- APIF-B: A partial implementation of our proposed theoretical framework. Unlike APIF-A, it doesn't perform vulnerability probability calculation and independence analysis. Each request undergoes a single mutation.

5.3 Testing Vector Library

The effectiveness of API fuzzing tools depends not only on the framework but also on the crafted test vector library. To avoid bias from expert experience, we used the well-known security vulnerability testing library SecLists [13], which contains different types of vulnerability testing payloads as our test vector library. By standardizing the vector library across tools, we can more accurately compare the strengths and weaknesses of different tools in terms of framework and algorithmic design. All tools above were configured to use only vectors from SecLists [13] for automated testing without manual intervention.

5.4 Setup and Metrics

Our experiments were conducted on an Ubuntu 20.04 system with an Intel I7 processor and 16GB of RAM. We ran tests using RESTler,

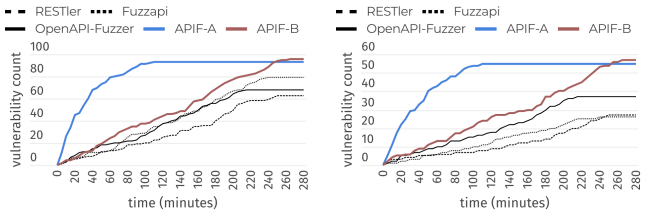
Fuzzapi, OpenAPI-Fuzzer, APIF-A, and APIF-B. We recorded metrics including total testing time, time to reach 50% and 90% API coverage, network metrics, and the vulnerability output indicators:

- Vulnerability Reported (TP+FP): Refers to the automatic calculation and reporting of vulnerability results by each fuzzer. A piece of vulnerability information contains three key elements: 1) The type of vulnerability, 2) the API endpoint where the vulnerability is located, and 3) the parameter position that triggers the vulnerability.
- Vulnerability Verified (TP): Refers to the process of manually verifying that the vulnerabilities produced by fuzzers (the three elements mentioned above) are completely the same as the records in the dataset, we mark the vulnerability as "Verified".

Since the tools RESTler, Fuzzapi, and OpenAPI-Fuzzer require parsing API parameter structures from OpenAPI Specification (OAS) files, we generated OAS files for 412 target APIs uniformly. Additionally, we used Postman (an API debugging tool capable of sending messages to APIs and receiving responses) [11] to trigger access behavior for each API. This allowed the MITM modules of APIF-A and APIF-B to capture the API communication traffic and thereby deduce the structure of the API parameters. Furthermore, both the target programs and testing tools were configured with valid authentication sessions. These setups ensure that all fuzzing tools are applicable to all target APIs, eliminating the impact on testing effectiveness due to the inability to obtain the structure of API parameters (e.g., missing OAS files or lack of API access traffic) and addressing the issue of missing authentication mechanisms.

5.5 Results

We conduct API vulnerability testing on all target APIs in the dataset. Figure 6a illustrates the number of vulnerabilities reported by each tool. After comparing the results against the dataset, the actual number of API vulnerabilities is shown in Figure 6b. The detailed vulnerabilities identified by each tool are listed in Table 1.



(a) Vulnerabilities reported by each fuzzer (TP+FP).

(b) Vulnerabilities confirmed after check (TP).

Figure 6: Number of vulnerabilities discovered by each fuzzer.

The detailed metrics for each tool, including total testing time, time to achieve 50% and 90% API coverage, network I/O counts, network traffic, and vulnerability count, as shown in Table 2.

We compare the experimental results of different tools from several perspectives: effectiveness, efficiency, and generalizability. We also conducted ablation testing and evaluated the effectiveness of vulnerability detection in real-world scenarios.

Table 1: The true positive result of each fuzzer

Vulnerabilities in Dataset		Discovery of Each Fuzzer (TP)				
Vulnerability Type	Count	RESTler	Fuzzapi	OpenAPI-Fuzzer	APIF-A	APIF-B
Data Exposure	11	4	6	3	6	6
Command Injection	9	0	3	6	6	6
Broken Object Authorization	13	3	1	4	8	8
File Read Vulnerability	8	2	2	4	4	4
Broken Authentication	12	3	2	6	6	6
Server-side request forgery	9	3	2	4	5	5
SQL Injection	9	1	0	0	5	6
Security Misconfiguration	5	2	0	1	0	0
Denial of Service	7	2	2	4	4	4
Improper Error Handling	4	1	0	0	0	0
Cross-site scripting	8	2	4	2	7	7
CSP Not Implemented	3	1	1	1	1	1
Unvalidated Redirects	5	1	0	0	1	1
Others	9	2	3	2	2	3
Total	112	27	26	37	55	57
Recall (%)	-	24.1	23.2	33.0	**49.1**	**50.9**

Table 2: Metrics and comparison

Test Items	RESTler	Fuzzapi	OpenAPI-Fuzzer	APIF-A	APIF-B
Total testing time (minute)	248	238	214	**109**	265
50% API coverage time (minute)	123	117	102	59	131
90% API coverage time (minute)	219	211	188	97	233
Network I/O counts (thousands)	430	387	366	163	482
Network traffic (MB)	242.1	219.9	206.5	96.8	275.4
Vulnerabilities reported	63	79	68	93	96
Vulnerabilities verified	27	26	37	55	57
Recall (%)	24.1	23.2	33.0	**49.1**	**50.9**
Precision (%)	42.9	32.9	54.4	**59.1**	**59.4**

Table 3: Real-world vulnerabilities discovered by APIF

Vulnerabilities Discovered by APIF		Detected (•)		
ID	Type	RESTler	Fuzzapi	OpenAPI-Fuzzer
CVE-2022-35509	Cross-site Scripting			
CVE-2022-39054	Cross-site Scripting		•	•
CVE-2022-41471	Broken Authorization			
CVE-2022-41472	Cross-site Scripting			
CVE-2022-42154	Unrestricted File Upload			•
CVE-2022-42735	Broken Authorization	•	•	
CNVD-2022-56311	Cross-site Scripting			
CNVD-2022-67082	Sensitive Data Exposure			
CNVD-2022-70325	Sensitive Data Exposure		•	
CNVD-2022-70707	SQL Injection			
CNVD-2022-70700	Broken Authorization			•
CNVD-2022-71314	Broken Authorization	•	•	•
CNVD-2022-73094	Command Injection		•	•
CNVD-2022-73085	Unrestricted File Read	•		•
CNVD-2022-73214	Cross-site Scripting			
CNVD-2022-73378	Broken Authorization			
CNVD-2022-73407	Unrestricted File Upload		•	•
CNVD-2022-74556	Broken Authorization			

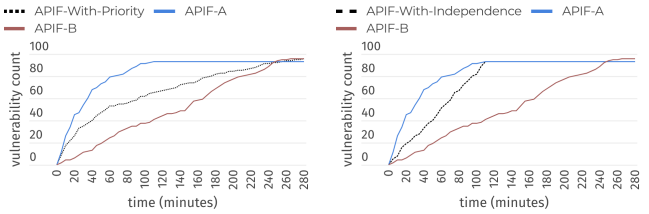
5.5.1 The Comparison of Effectiveness. Table 2 reveals that APIF-A and APIF-B outperform the other tools in terms of vulnerability detection recall and precision. In addition to the fact that real API communications typically contain more information, many parameters can only be effectively tested after undergoing recursive decoding and encoding. With a consistent test payload library, this indicates that our API parameter decoding and tree-structured mutation significantly contribute to enhancing the coverage of vulnerability testing, enabling the discovery of more hidden issues.

5.5.2 The Comparison of Efficiency. While producing similar vulnerability results, APIF-A saves 58.9% of the testing time compared to APIF-B, and also significantly reduces the amount of network I/O requests and network traffic consumption. This demonstrates that the API parameter independence analysis algorithm we employed in the test vector generation phase effectively reduces the number of tests and enhances overall testing efficiency. The concurrent testing of parameters resulted in APIF-A identifying 2 fewer vulnerabilities than APIF-B, indicating that the accuracy of the algorithm requires further optimization.

Additionally, based on the vulnerability discovery curve of APIF-A in Figure 6b, we observe that APIF-A reported 76.4% of its vulnerability detection in the first half of the testing time. The diminishing slope of APIF-A's data curve indicates that the testing priority calculation algorithm effectively prioritizes high-risk APIs. For real-world large-scale application testing, there is a prevalent objective to discover a maximal number of security vulnerabilities within a constrained time. This imperative lends considerable credence to the feasibility of deploying APIF in extensive, real-world applications.

5.5.3 The Comparison of Generalizability. We observed that the SilverStripe [14] project contains four GraphQL API vulnerabilities: CVE-2023-44401, CVE-2023-40180, CVE-2023-28104, and CVE-2021-28661. As mentioned earlier, existing research on API vulnerabilities has made optimizations for RESTful APIs but has not taken into account other types of APIs. Therefore, the tools RESTler, Fuzzapi and OpenAPI-Fuzzer did not identify the four vulnerabilities. Meanwhile, APIF-A and APIF-B successfully identified two of those vulnerabilities, CVE-2023-28104 and CVE-2023-40180. Triggering these vulnerabilities requires further decoding of the GraphQL messages before inserting test vectors. Our APIF framework's recursive decoder and tree structure mutation accomplished this task, indicating that the theoretical approach of APIF is effective across different API protocols, offering greater generalizability.

5.5.4 Ablation Study. To investigate how the API parameter independence analysis and priority calculation strategy contribute to improving the efficiency of APIF, we conducted an ablation study on these two main components. We implemented different variants of APIF: 1) APIF-A, which enabled both parameter independence analysis and priority calculation, 2) APIF-B, which removed both parameter independence analysis and priority calculation, 3) APIF-With-Priority, which removed the implementation of API parameter independence analysis, and 4) APIF-With-Independence, which removed the implementation of testing priority calculation. The results are shown in Figure 7. As in previous studies, the testing payload set was Seclists[13], and the four tests received the same API communication data.



(a) Vulnerabilities reported by each variant (TP+FP).

(b) Vulnerabilities reported by each variant (TP+FP).

Figure 7: Number of vulnerabilities discovered by each tool.

In Figure 7a, compared to the unoptimized APIF-B, APIF-With-Priority can discover more vulnerabilities within a relatively short time frame without affecting the effectiveness of vulnerability detection, while the overall fuzzing duration remains nearly the same. This is because the parameter priority analysis primarily optimizes the testing sequence. In Figure 7b, the curve of APIF-With-Independence significantly shortens the overall fuzzing duration, greatly improving efficiency. This improvement is due to the concurrent insertion of multiple payloads in a single request, allowing more test cases to be completed in a given time. However, due to incorrect judgments of API parameter independence in some cases, the number of detected vulnerabilities slightly decreases. This indicates that there is still room for further optimization of the parameter independence analysis method.

5.6 Real-World Vulnerability Testing

We applied the implementation of the APIF tool to real-world vulnerability testing and discovered 188 bugs and 26 vulnerabilities across 60 open-source API projects. These vulnerable applications include open-source applications such as EyouCMS and cloud-native API gateway services like Apache ShenYu. We reported these vulnerabilities to the respective service vendors, and 18 of them have been addressed and publicly disclosed, as shown in Table 3.

5.6.1 Test Environment Setup. During the test setup process, our goal was to ensure that each testing tool could fully obtain the API information of the target programs. First, we deployed all target testing programs in the test environment. Then, by configuring proxies and manually triggering interactions through Postman, we captured the communication data required by APIF. Additionally, we generated corresponding OAS files for each target testing program. The payload set for testing uniformly adopted SecLists[13], and the maximum runtime for a single test target was set to 240 minutes.

5.6.2 Results Comparison. For the vulnerabilities discovered by APIF, we conducted supplementary tests using three other tools within the same time limit. The results (as shown in Table 3) indicated that RESTler identified 3 of these vulnerabilities, Fuzzapi discovered 6, and OpenAPI-Fuzzer found 7. This further demonstrates that the optimizations in APIF effectively enhance the outcomes of vulnerability fuzzing.

Through comparative experiments and testing in real-world evaluation, we have validated the effectiveness of APIF. The optimizations implemented in our proposed fuzzing framework and

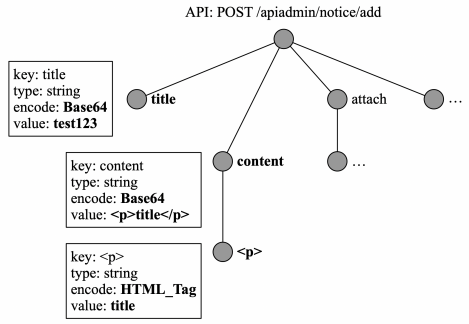


Figure 8: Parsing and decoding API request parameters into the tree structure.

process, including API parameter parsing, testing priority calculation, and test vector generation optimizations have collectively enhanced both the coverage and efficiency of API vulnerability fuzzing.

Listing 3: Request message captured in API communication

```
POST /apiadmin/notice/add HTTP/1.1

{
    "title": "dGVzdDEyMw==",
    "content": "PHA+dGVzdDxwPg==",
    ...
}
```

5.7 Case Study: Discovery of CVE-2022-41472

Among the real-world CVE vulnerabilities discovered by APIF, CVE-2022-41472 [3] was identified as a Cross-site Scripting (XSS) vulnerability within a JSON structure. This approach involved several steps.

5.7.1 API Acquisition. First, we deploy the service under test within an internal environment and simulate normal traffic flows, storing the traffic in the form of raw HTTP messages, which are then sent to APIF for fuzzing. The original request message is shown in Listing 3. APIF identified API `/apiadmin/notice/add`, which `title` and `content` parameters in the API request message have been encoded with `base64`.

5.7.2 API Parameter Parsing. The captured API traffic was recursively decoded and structurally parsed into a tree structure. In Figure 8, during the decoding process of parameter values, APIF identifies that the `content` parameter contains a segment of HTML code. Consequently, a corresponding decoded node <p> is added downstream. Generally, when a program renders HTML code snippets submitted by users, it becomes susceptible to Cross-Site Scripting (XSS) vulnerabilities.

5.7.3 Testing Priority Calculation. The system calculated the potential for vulnerabilities in all API endpoints to prioritize high-risk APIs for testing. It was calculated that `/apiadmin/notice/add` API had a high priority for vulnerability testing (Table 4).

5.7.4 Testing Vectors Generation. After identifying high-risk APIs, APIF used constraint solving and independence analysis to select appropriate payloads from the testing vector library and injected

Table 4: Calucate testing priority for all APIs in the project

Testing Priority	API Endpoint	Vuln Score
1	`/apiadmin/order/check`	0.8172
2	`/apiadmin/notice/add`	0.8106
3	`/member/login/company/{id}`	0.7852
4	`/member/login/personal/{id}`	0.7683
5	`/apiadmin/help/add`	0.7289
...	...	...

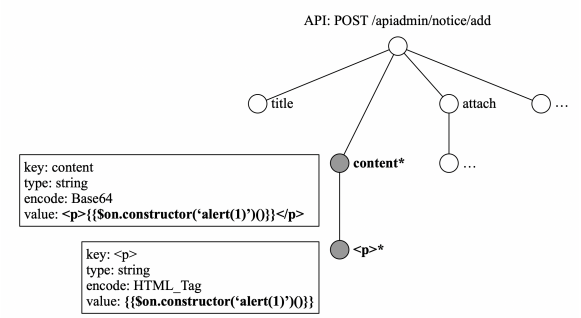


Figure 9: Mutate node value with XSS payloads.

them into the content node (Figure 9). Then the tree-structured data was restored and encoded, generated a testing request, and sent to the target server.

5.7.5 Vulnerability Verification. After getting a remote API response content, the built-in verification rules identified the API with path `/apiadmin/notice/add` was vulnerable to XSS attacks through the `title` parameter in its JSON body when using the POST method. Following the automated detection, we manually reproduced the vulnerability to confirm its existence and reported it to the relevant vendor. The vendor acknowledged the vulnerability and provided its CVE-2022-41472 identifier.

6 DISCUSSION AND LIMITATIONS

6.1 API Communication Traffic Analysis

APIF employs passive traffic analysis to parse the parameter structure of APIs. However, in the experimental process mentioned above, other tools parsed OpenAPI Specification (OAS) files. To ensure the validity of our experimental results, we generated uniform OAS files for every API in the dataset, ensuring that the control group tools would not fail to discover vulnerabilities due to missing API documentation.

In API parameter parsing, traffic analysis and OAS parsing are complementary methods with their pros and cons. Traffic analysis excels in capturing extensive API context and works for APIs lacking OAS files, applicable to various API types. Its downside is needing a man-in-the-middle service to intercept API traffic. OAS parsing stands out for its simplicity, does not require traffic interception, and suits RESTful APIs well. However, its effectiveness is limited when OAS documents are unavailable, necessitating manual OAS document generation.

6.2 API Authentication

For online services API, certain functionalities require user authentication. For instance, a typical scenario might allow a regular user to access information using the GET method, but only an administrator can modify information using the PUT method. To enhance test coverage, the API fuzzer needs to comprehend the authentication mechanisms of different APIs.

In our experiments, we configured valid authentication sessions for the testing tools and the applications under test, enabling the fuzzing tools to access content that requires authorization. In API vulnerability fuzzing, failing to obtain API authentication details significantly impacts API test coverage. Therefore, we developed an authentication module for APIF, which can carry various authentication information in testing requests based on predefined configurations. Users can submit their API access credentials to APIF through a configuration file. Once APIF recognizes an authentication request, it will automatically carry these credentials for subsequent testing. However, in real-world scenarios, not all target applications support long-lasting authentication sessions. In such cases, testing tools can only perform time-limited security testing using the short-term sessions they have obtained.

6.3 The Universality of Parameter Priority Calculation

Before performing parameter priority calculation, we used historical CVE vulnerabilities as the dataset and validated the metrics based on TCL-based fuzzing [44]. Drawing on our experience in the field of API vulnerability mining and the practical characteristics of black-box API security testing scenarios, we ultimately selected three of the most representative feature dimensions in security testing scenarios to calculate the priority. Through experiments, we successfully demonstrated that this improvement can enhance the efficiency of API vulnerability fuzzing based on parameter injection, parameter modification, and parameter deletion, thereby encouraging more subsequent research.

However, API vulnerabilities also include many other categories, such as logical vulnerabilities. Due to the typically more complex and highly case-specific testing methods required for such vulnerabilities, this work did not address efficiency optimization methods for these types of vulnerabilities. In future work, we will further optimize priority-guided methods for different types of vulnerabilities.

7 RELATED WORK

With the rise of cloud computing, RESTful APIs have seen widespread adoption, spurring research in RESTful API fuzzing. Eddouibi et al. proposed an automated testing method that begins by parsing OpenAPI description files to extract API models from JSON or YAML files, creating a test model that is then transformed into executable JUnit code [29]. This method relies on random values to generate parameters testing payloads.

RESTler [22], open-sourced and proposed by Atlidakis et al. in 2019, marks the first instance of a stateful tool designed to automatically fuzz cloud services through their RESTful APIs [12]. Following this, the same team integrated a series of security rule checkers within RESTler [23], enabling the automatic identification of security vulnerabilities. Furthering this field, Godefroid and associates developed a methodology for differential regression testing specifically for RESTful APIs [31]. This approach utilized RESTler to create network logs across different versions of specified RESTful APIs, facilitating the detection of both service and specification regressions through the analysis of these logs.

Meanwhile, researchers focusing on fuzzing RESTful APIs have adopted different methods to optimize the generation of test vectors to improve test coverage. HsuanFuzz [44] uses the Test Coverage Levels (TCL) algorithm and grey-boxed feedback analysis to identify whether mutations and vectors are effective, enabling meaningful mutations to generate new values. Inspired by this research, we have simplified the TCL algorithm in the scenario of black-box API fuzzing, leading to the vulnerability prioritization algorithm.

Additionally, we have taken note of studies that utilize data-driven methods to optimize the process of API fuzzing. Pythia [21] incorporates a machine-learning model to determine mutation strategies for different segments of a request. It also employs code coverage analysis to direct the fuzzing process. Miner [40] leverages the historical data to guide the sequence construction and improves the fuzzing request generation. These methods require a long duration to acquire real API communication traffic for perfect learning, in order to understand the parameter correlations of APIs and generate appropriate test vectors. For large-scale fuzzing tasks with limited time, the rapid vulnerability prioritization algorithm and parameter correlation analysis algorithm we propose will be more efficient.

As mentioned above, existing black-box API fuzzing research mainly focuses on RESTful APIs. Methods for testing other types of APIs have also been proposed, including fuzzers for GraphQL API [24, 25, 35, 47], SOAP API [33, 41], and gRPC API [49, 50]. However, most of these studies start from the perspective of software robustness, using white-box or gray-box testing methods, and cannot achieve black-box vulnerability detection like the experimental objects selected in this article.

The RESTful API fuzzing studies above are contingent on providing accurate API Specification files by API providers. This reliance significantly limits the application of these techniques to vulnerability testing in various other types of APIs. To address this limitation and enhance the generalizability and practicality, our research is based on analyzing the API communication traffic. We employ a unified tree structure for storing and mutating parameters of different API types, making our research applicable to other mainstream APIs such as SOAP, GraphQL, gRPC, MQTT, etc.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced an API vulnerability fuzzing framework. Our innovative approach employs a tree structure for structural analysis of API parameters, effectively addressing complex encoding issues and enabling the discovery of deeper API vulnerabilities, while also improving the generalizability of the framework to cope with various types of APIs in real-world batch testing scenarios. Subsequently, we optimized the API vulnerability testing efficiency by calculating the likelihood of vulnerabilities in individual APIs for prioritization and analyzing parameter independence within each API. This allowed embedding multiple test vectors in a single

request, thereby significantly reducing the number of tests and improving testing efficiency.

Our experimental results validate the effectiveness of our framework, demonstrating its advantages over existing API testing technologies in terms of efficiency and capability to uncover more vulnerabilities in black-box testing. We applied this framework to real-world vulnerability fuzzing, discovering 188 bugs and 26 vulnerabilities, including 6 CVEs and 12 CNVDs in 60 open-sourced API projects. Overall, our approach can serve as a practical direction to improve the API vulnerability fuzzing techniques.

Future work will focus on refining the vulnerability likelihood assessment by incorporating additional factors that correlate to vulnerabilities and exploring the interrelations between these factors to optimize our evaluation methods. Additionally, we will further improve the parameter independence analysis to achieve more precise and effective results. These enhancements will further improve the efficiency of black-box API fuzzing.

REFERENCES

[1] [n. d.]. *APIF*. Retrieved April 15, 2024 from https://github.com/apif-tool/APIF_tool_2024

[2] [n. d.]. *APISandbox*. Retrieved February 28, 2024 from https://github.com/API-Security/APISandbox

[3] [n. d.]. *CVE-2022-41472 Vulnerability Description*. Retrieved February 28, 2024 from https://www.cve.org/CVERecord?id=CVE-2022-41472

[4] [n. d.]. *Fuzzapi*. Retrieved February 28, 2024 from https://github.com/Fuzzapi/fuzzapi

[5] [n. d.]. *GitLab-CE Download Page*. Retrieved February 28, 2024 from https://packages.gitlab.com/gitlab/gitlab-ce

[6] [n. d.]. *MitmProxy Homepage*. Retrieved February 28, 2024 from https://mitmproxy.org/

[7] [n. d.]. *Open API Specification*. Retrieved February 28, 2024 from https://swagger.io/specification/

[8] [n. d.]. *OpenAPI-Fuzzer*. Retrieved February 28, 2024 from https://github.com/matusf/openapi-fuzzer

[9] [n. d.]. *OWASP API Security Risk List*. Retrieved February 28, 2024 from https://owasp.org/API-Security/

[10] [n. d.]. *OWASP crAPI API Vulnerability Sandbox*. Retrieved February 28, 2024 from https://github.com/OWASP/crAPI

[11] [n. d.]. *Postman*. Retrieved February 28, 2024 from https://www.postman.com/downloads/

[12] [n. d.]. *Restler Homepage*. Retrieved February 28, 2024 from https://github.com/microsoft/restler-fuzzer

[13] [n. d.]. *SecLists*. Retrieved February 28, 2024 from https://github.com/danielmiessler/SecLists

[14] [n. d.]. *SilverStripe CMS Homepage*. Retrieved February 28, 2024 from https://www.silverstripe.org/

[15] [n. d.]. *Spree*. Retrieved February 28, 2024 from https://github.com/spree/spree

[16] [n. d.]. *Swagger Homepage*. Retrieved February 28, 2024 from https://swagger.io/

[17] [n. d.]. *VAmPI*. Retrieved February 28, 2024 from https://github.com/erev0s/VAmPI

[18] [n. d.]. *vapi*. Retrieved February 28, 2024 from https://github.com/roottusk/vapi

[19] Juan C. Alonso, Alberto Martin-Lopez, Sergio Segura, José María García, and Antonio Ruiz-Cortés. 2023. ARTE: Automated Generation of Realistic Test Inputs for Web APIs. *IEEE Transactions on Software Engineering* 49, 1 (2023), 348–363. https://doi.org/10.1109/TSE.2022.3150618

[20] Juan C. Alonso, Sergio Segura, and Antonio Ruiz-Cortés. 2023. AGORA: Automated Generation of Test Oracles for REST APIs. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1018–1030. https://doi.org/10.1145/3597926.3598114

[21] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations. *CoRR* abs/2005.11498 (2020). arXiv:2005.11498 https://arxiv.org/abs/2005.11498

[22] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 748–758. https://doi.org/10.1109/ICSE.2019.00083

[23] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020. Checking Security Properties of Cloud Service REST APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 387–397. https://doi.org/10.1109/ICST46399.2020.00046

[24] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. Evolutionary-based automated testing for GraphQL APIs. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 778–781. https://doi.org/10.1145/3520304.3528952

[25] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. White-Box and Black-Box Fuzzing for GraphQL APIs. arXiv:2209.05833 [cs.SE]

[26] R. Dennis Cook. 1977. Detection of Influential Observation in Linear Regression. *Technometrics* 19, 1 (1977), 15–18. http://www.jstor.org/stable/1268249

[27] Davide Corradini, Michele Pasqua, and Mariano Ceccato. 2023. Automated Black-Box Testing of Mass Assignment Vulnerabilities in RESTful APIs. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2553–2564. https://doi.org/10.1109/ICSE48619.2023.00213

[28] Gelei Deng, Zhiyi Zhang, Yuekang Li, Yi Liu, Tianwei Zhang, Yang Liu, Guo Yu, and Dongjin Wang. 2023. NAUTILUS: automated RESTful API vulnerability detection. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) *(SEC '23)*. USENIX Association, USA, Article 313, 17 pages.

[29] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. 181–190. https://doi.org/10.1109/EDOC.2018.00031

[30] Matúš Ferech and Pavel Tvrdík. 2023. Efficient fuzz testing of web services. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. 291–300. https://doi.org/10.1109/QRS60937.2023.00037

[31] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 725–736. https://doi.org/10.1145/3368089.3409719

[32] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 312–323. https://doi.org/10.1145/3395363.3397374

[33] Sadeeq Jan, Cu D. Nguyen, and Lionel C. Briand. 2016. Automated and effective testing of web services for XML injection attacks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2931037.2931042

[34] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. 2019. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)* (2019), 131–141. https://api.semanticscholar.org/CorpusID:209439495

[35] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2021. Automatic Property-based Testing of GraphQL APIs. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. 1–10. https://doi.org/10.1109/AST52587.2021.00009

[36] V Kathiresan and P Sumathi. 2012. An efficient clustering algorithm based on Z-Score ranking method. In *2012 International Conference on Computer Communication and Informatics*. 1–4. https://doi.org/10.1109/ICCCI.2012.6158779

[37] Yi Liu. 2022. RESTInfer: automated inferring parameter constraints from natural language RESTful API descriptions. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1816–1818. https://doi.org/10.1145/3540250.3559078

[38] Yi Liu. 2023. RESTCluster: Automated Crash Clustering for RESTful API. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 198, 3 pages. https://doi.org/10.1145/3551349.3559511

[39] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: model-based RESTful API testing with execution feedback. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1406–1417. https://doi.org/10.1145/3510003.3510133

[40] Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Chen, and Raheem Beyah. 2023. MINER: A Hybrid Data-Driven Approach for REST API Fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4517–4534.

[41] Riyadh Mahmood, Jay Pennington, Danny Tsang, Tan Tran, and Andrea Bogle. 2022. A Framework for Automated API Fuzzing at Enterprise Scale. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 377–388. https://doi.org/10.1109/ICST53961.2022.00018

[42] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. 2022. On the Faults Found in REST APIs by Automated Test Generation. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 41 (mar 2022), 43 pages. https://doi.org/10.1145/3491038

[43] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. A Catalogue of Inter-parameter Dependencies in RESTful Web APIs. In *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 399–414. https://doi.org/10.1007/978-3-030-33702-5_31

[44] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. Test coverage criteria for RESTful web APIs. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Tallinn, Estonia) *(A-TEST 2019)*. Association for Computing Machinery, New York, NY, USA, 15–21. https://doi.org/10.1145/3340433.3342822

[45] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: automated black-box testing of RESTful web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 682–685. https://doi.org/10.1145/3460319.3469082

[46] Chung-Hsuan Tsai, Shi-Chun Tsai, and Shih-Kun Huang. 2021. REST API Fuzzing by Coverage Level Guided Blackbox Testing. *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)* (2021), 291–300. https://api.semanticscholar.org/CorpusID:245634367

[47] Daniela Meneses Vargas, Alison Fernandez Blanco, Andreina Cota Vidaurre, Juan Pablo Sandoval Alcocer, Milton Mamani Torres, Alexandre Bergel, and Stéphane Ducasse. 2018. Deviation Testing: A Test Case Generation Technique for GraphQL APIs. https://api.semanticscholar.org/CorpusID:220494731

[48] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 142–152. https://doi.org/10.1109/ICST46399.2020.00024

[49] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2023. White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 122 (Jul 2023), 38 pages. https://doi.org/10.1145/3585009

[50] Man Zhang, Andrea Arcuri, Yonggang Li, Kaiming Xue, Zhao Wang, Jian Huo, and Weiwei Huang. 2022. Fuzzing Microservices In Industry: Experience of Applying EvoMaster at Meituan. arXiv:2208.03988 [cs.SE]