Vulnerability-oriented Testing for RESTful APIs

Wenlong Du*[1], Jian Li*[1], Yanhao Wang[2], Libo Chen ✉[1], Ruijie Zhao[1],
Junmin Zhu[1], Zhengguang Han[3], Yijun Wang[1], and Zhi Xue[1]
[1]School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University
[2]Independent Researcher [3]QI-ANXIN Technology Group
{beet1e, vfenux483, bob777, ruijiezhao, junmin.zhu, ericwyj, zxue}@sjtu.edu.cn,
wangyanhao136@gmail.com, *hanzhengguang@qianxin.com*

Abstract

With the increasing popularity of APIs, ensuring their security has become a crucial concern. However, existing security testing methods for RESTful APIs usually lack targeted approaches to identify and detect security vulnerabilities. In this paper, we propose VoAPI[2], a vulnerability-oriented API inspection framework designed to directly expose vulnerabilities in RESTful APIs, based on our observation that the type of vulnerability hidden in an API interface is strongly associated with its functionality. By leveraging this insight, we first track commonly used strings as keywords to identify APIs' functionality. Then, we generate a stateful and suitable request sequence to inspect the candidate API function within a targeted payload. Finally, we verify whether vulnerabilities exist or not through feedback-based testing. Our experiments on real-world APIs demonstrate the effectiveness of our approach, with significant improvements in vulnerability detection compared to state-of-the-art methods. VoAPI[2] discovered 7 zero-day and 19 disclosed bugs on seven real-world RESTful APIs, and 23 of them have been assigned CVE IDs. Our findings highlight the importance of considering APIs' functionality when discovering their bugs, and our method provides a practical and efficient solution for securing RESTful APIs.

1 Introduction

Application Programming Interfaces (APIs), as the contract between information providers and information users, are gaining immense popularity as they effectively facilitate communication and data exchange between diverse web applications. It also enables businesses to harness their data and services well to support innovative scenarios and broaden their impact. Currently, it has been widely used and spanned across various scenarios, such as cloud services [1, 2], content management systems (CMS) [3–6], and Internet of Things (IoT) [7, 8] devices. Among different architectural styles for API development, REST [9] (REpresentational State Transfer) has emerged as a widely embraced approach, with APIs adhering to this style commonly known as RESTful APIs.

Unfortunately, numerous security issues [10, 11] have arisen in APIs in recent years, resulting in significant repercussions for providers and users. For instance, a vulnerability [10] in a Facebook API interface exposed millions of users' private information. As we see, API security has become a critical concern, especially for service providers. Meanwhile, APIs have increasingly become a focal point for attackers, making them a common target in various services. Hence, proactively identifying and addressing API flaws is vital in safeguarding services and their users. Detecting vulnerabilities early on is an essential strategy to ensure the protection of these systems.

Currently, API testing has gained significant popularity as a solution to enhance API security, and a considerable amount of research has been conducted in this field. However, existing testing methods for RESTful APIs often encounter limitations when it comes to identifying security vulnerabilities, especially in black-box testing scenarios [12–17]. These methods heavily rely on the API's status feedback, which means they may not detect certain logical vulnerabilities (e.g., SSRF) that do not affect the normal functioning of APIs. On the other hand, white-box testing methods [17–21] require source code security audits, which can be time-consuming, inefficient, and prone to generating false positives. Gray-box testing methods [22, 23] require access to the internal state of API execution, offering a more comprehensive view. Nevertheless, the diverse code frameworks used in API implementations present a challenge in developing a unified detection approach. Furthermore, testing methods tailored to specific architectures [24] have limited scope and are difficult to extend beyond their original application scenarios.

In addition, several other methods [12, 15, 16, 25–32] have been proposed that leverage OpenAPI[1] specifications to en-

*Co-leading authors. ✉Corresponding author.

[1]OpenAPI [33] (i.e., Swagger) stands as the most widely used specification for describing RESTful API interfaces. A Swagger-based specification outlines the usage guidelines for RESTful APIs, including the types of service requests accepted, the expected response format, and other relevant details.

able unified testing of APIs. These methods include functional testing [12, 15, 16, 25–29, 34], compliance testing [30], and security testing [31, 32]. Parts of these approaches [26–29] utilize OpenAPI specifications to parse HTTP requests and generate fuzzing test cases that adhere to syntax rules. Conversely, some methods [12, 15, 16, 25, 34] use OpenAPI specifications to infer the dependencies between requests and generate test messages that align with the API interaction context. This enables effective testing of API functionality, as well as the identification of exceptions and specific security vulnerabilities.

Despite the emergence of various testing approaches, it is noteworthy that most of these methods do not effectively utilize the specific characteristics of security vulnerabilities to implement vulnerability-targeted testing strategies. Instead, they primarily rely on the detection of "500 Internal Server Errors" as an indication of capturing bugs within API interfaces. Consequently, this approach leads to limited visibility and low testing efficiency. For example, when dealing with a large-scale API project similar in scale to GitLab, RESTler [12], on average, requires more than 5 hours to perform a comprehensive test. Furthermore, the security vulnerabilities uncovered by these tools, such as RestTestGen [16], represent only a small fraction of these bugs, with the majority being functional issues that primarily impact the reliability of APIs.

Based on our analysis of publicly disclosed API vulnerabilities in recent years, we observed that there exists a clear correlation between different API security vulnerabilities and the functionality of APIs. For instance, file upload interfaces can pose a risk of malware uploads, while proxy interfaces may harbor SSRF vulnerabilities. Building upon this observation, we propose the core concept of our method, which involves *identifying API functions associated with vulnerabilities within API specifications and conducting targeted security testing on those functions*. This approach aims to achieve **vulnerability-oriented API inspection**, thereby facilitating the effective discovery of bugs across a wide range of APIs.

There are three challenges in realizing vulnerability-oriented API inspection in practice. The first challenge is efficiently differentiating between various functional interfaces in an API and identifying functions that may have security vulnerabilities. The second challenge is effectively generating valid test cases based on different function interfaces and corresponding security vulnerability types. The last challenge is generating test sequences that comply with protocol states, incorporating the related test cases, and successfully executing the function interfaces.

To address these challenges, we propose VoAPI^2, a **V**ulnerability-**O**riented **API I**nspection framework designed to directly expose vulnerabilities in RESTful APIs. Our approach begins by identifying semantic keywords associated with potentially weak functions within the API specification. It then integrates a suitable corpus and employs a stateful request sequence that aligns with the execution context of the corresponding functions. By utilizing inspection payloads tailored to different vulnerability types, we can effectively assess these candidate functions for potential vulnerabilities.

We evaluated our tool on 7 real-world RESTful APIs, which resulted in the discovery of 7 zero-day vulnerabilities and 19 disclosed bugs. Out of these vulnerabilities, 23 have been verified and acknowledged by the respective vendors, and have been assigned CVE IDs. The experiment results demonstrate that VoAPI^2 surpasses the state-of-the-art methods by a significant margin, not only in terms of the number of identified bugs but also in terms of detection efficiency.

In summary, we make the following contributions:

- We propose and implement a novel method, VoAPI^2, to automatically reveal the vulnerability of RESTful APIs. It can perform vulnerability-oriented testing based on API specifications to effectively capture defects in RESTful APIs, and output the corresponding PoC. We will release the source code as well as the experiment data at https://github.com/NSSL-SJTU/VoAPI2.
- We evaluated our tool with several state-of-the-art methods. The experiment results demonstrate that VoAPI^2 surpasses all these methods by a significant margin, not only in terms of the number of identified bugs but also in terms of detection efficiency.
- We conducted an evaluation of VoAPI^2 on 7 real-world RESTful APIs, and VoAPI^2 identified 26 vulnerabilities, including 7 previously-unknown and 19 known security bugs. All zero-day bugs have been reported to vendors and fixed; four of them have been assigned CVE IDs.

2 Problem and Approach Overview

In this section, we first provide the background of vulnerabilities in RESTful API. Then, we present a motivation example, and our method to detect the hidden vulnerability in the sample. At last, we discuss the associated challenges and present an overview of our solution.

2.1 Vulnerability of RESTful API

REST, short for REpresentational State Transfer, is a widely adopted software architecture style for network applications. It is commonly utilized within the context of HTTP, and an API adhering to this style is referred to as a RESTful API. While existing tools are available for testing REST APIs, many of them primarily focus on assessing the reliability of these APIs rather than specifically targeting potential security vulnerabilities. However, similar to web applications, REST APIs are also vulnerable to various security threats, as demonstrated by the presence of such vulnerabilities in the CVE database. These vulnerabilities encompass issues like

```yaml
/Images/Remote:
  get:
    operationId: GetRemoteImage
    parameters:
      description: The image url.
      in: query
      name: imageUrl
      required: true
      schema:
        description: The image url.
        format: uri
        type: string
    responses:
      "200":
        content:
          image/*:
            schema:
              format: binary
              type: string
        description: Remote image returned.
      "404":
        content:
          # omitted
        description: Remote image not found.
    summary: Gets a remote image.
    tags:
      - RemoteImage
```

Figure 1: **Specification of an API endpoint.**

Server-Side Request Forgery, Unrestricted File Upload, OS Command Injection, Path Traversal, and more.

Motivation Example. OpenAPI, formerly known as Swagger, provides a specification for describing RESTful APIs. Typically, the specification document is presented in JSON or YAML format and encompasses the resources and operations of the RESTful API. It comprises a collection of Schema Objects, which encompass various elements such as API service endpoints (commonly referred to as paths), CRUD operations (such as `GET`, `POST`, `PUT` and `DELETE`) performed on an endpoint, input parameters, and expected responses. Each schema object follows a predefined structure with corresponding parameter types. Users can leverage the specification to generate valid API operations and communicate with API services by presenting them as HTTP requests.

Figure 1 illustrates a simplified fragment of the OpenAPI specification for Jellyfin [3], an open-source media management system. This fragment specifies an API endpoint, `/Images/Remote`, which supports a single CRUD operation, `GET`. It defines the input parameters and expected response for this endpoint. Specifically, the input parameter is named "`imageUrl`" (Line 7), which is required and used to specify the URL of the image (Line 10). The corresponding schema (Line 9) designates the parameter type as a string (Line 12). The response section includes the HTTP status code and response message (Line 13). Furthermore, this example includes a "`summary`" field (Line 25), which typically provides a brief description of the API's functionality (similar to the "`description`" field). Additionally, there is a "`tags`" field

Figure 2: **Exploitation process of CVE-2021-29490.**

(Line 26) that offers category tags to classify the API.

In Jellyfin (version 10.7.3 and earlier), there exists a Server-Side Request Forgery (SSRF) vulnerability (CVE-2021-29490 [35]) in the aforementioned API endpoint. Exploiting this vulnerability requires the attacker to send a GET request to the `/Images/Remote` API with testing payloads (e.g., `http://LAN_IP/ssrf`) in the query parameter "`imageUrl`". The root cause of this vulnerability is the lack of filtering for the remote resource address specified by the "`imageUrl`" parameter in the backend of the API service. This oversight allows attackers to freely access internal resources by sending an HTTP request (i.e., the testing payload) from the Jellyfin server, as depicted in Figure 2.

As we see, the vulnerability in the motivation sample is closely related to the functionality of the corresponding API endpoint. Specifically, the SSRF bug typically arises in the API path responsible for retrieving a remote resource. As demonstrated in this example, the purpose of this API endpoint is to fetch a remote image, as indicated in its summary (Line 25). The parameter "`imageUrl`" is utilized to specify the address of the resource (Line 5). Consequently, if the backend of this API interface fails to rigorously validate input parameters, it may result in an SSRF vulnerability.

2.2 Observation and Our Method

Current security testing methods for APIs suffer from two primary efficiency issues and most of them cannot detect the vulnerability hidden in the motivation sample. *Firstly, they are time-consuming.* For example, RESTler [12] required over five hours to test four groups of API interfaces in the open-source code management software GitLab, with the longest effective request sequence length tested remaining at just three. This inefficiency can be attributed to testing tools needing to traverse all interfaces indiscriminately, rather than focusing on a specific API interface in-depth. *Secondly, existing testing tools heavily rely on service status feedback from API endpoints to gauge testing effectiveness, resulting in the identification of mostly service-disrupting bugs.* For instance, RESTler discovered 28 bugs causing GitLab [36] service unavailability, while RestCT [25] identified eight similar issues in Bing Maps [37] and GitLab.

Table 1: **Statistical Data of Known APIs' Vulnerabilities.** Here we list the disclosed API bugs of six common types that we collected from the CVE database. **#Bug** indicates the number of disclosed bugs. **#Ratio** indicates the proportion of common functionality in all vulnerable functions.

CWE ID	Vulnerability Type	#Bug	Functionality Summary of Vulnerable APIs	#Ratio	Keywords
CWE-434	Unrestricted Upload of File	42	Most of them focus on uploading.	83%(35/42)	"upload","submit","import", etc.
CWE-918	Server-Side Request Forgery	21	Most of them need to request a remote source, like Proxy interfaces.	81%(17/21)	"remote","proxy","URL", etc.
CWE-22	Path Traversal	31	They are responsible for handling file through a "Path" variable.	52%(16/31)	"path","dir","file", etc.
CWE-78	OS Command Injection	55	Most of them are used to set configurations through commands executed in OS shell.	40%(22/55)	"CMD","command","system", etc.
CWE-89	SQL Injection	70	They are responsible for handling SQL database.	53%(37/70)	"SQL","database","select", etc.
CWE-79	Cross-site Scripting	55	They present front-end pages that show text.	35%(19/55)	"display","content","view", etc.

Furthermore, these testing methods often lack a targeted strategy and approach based on specific security vulnerabilities. Instead, they primarily rely on detecting generic bugs through feedback such as "500 Internal Server Errors" from API endpoints. Consequently, this leads to an indiscriminate and inefficient testing approach, with the majority of detected bugs being functional issues rather than genuine security vulnerabilities.

Observation. According to the motivation sample, we **observe** that we can assess the functionality of APIs by examining certain feature strings commonly used to define API features in the specification, such as the presence of `remote` in the current example. Meanwhile, the keyword `remote` is also strongly associated with the type of vulnerability (SSRF). To validate our observation that the category of vulnerability present in an API interface is strongly linked to the functionality of that interface via some keywords, we gathered data from the CVE database on the most common types of revealed API vulnerabilities (e.g., Unrestricted Upload and SSRF) and selected vulnerabilities that had been disclosed in CVE records with specific details, such as the exact location of the bug within the API interface.

Based on our collected data[2], we first conduct a word frequency analysis on the API paths and parameters of API interfaces according to these vulnerabilities' CVE descriptions. Then, combining with expert knowledge (analyze the root cause of these vulnerabilities), we extract high-frequency words that also reflect the functionality of these vulnerable APIs, such as "upload" and "submit," and mapped these for the functionalities (e.g., File Upload function) and vulnerabilities (e.g., Unrestricted Upload of File). Finally, we try to construct a mapping from the vulnerability type to the corresponding function functionality and its feature strings frequently applied in context.

Our analysis results are presented in Table 1, which shows that, on average, 57% of vulnerable API interfaces affected by a specific type of bug belonged to the same functionality. These findings confirm the validity of our intuition for these common vulnerabilities. Additionally, we identified a group of commonly used strings in specific functionality as keywords. For example, in an upload interface, we may observe frequently used strings such as "`upload`", "`submit`", and "`import`" which can be used as keywords to identify corresponding interfaces from API specifications and support subsequent bug detection, as shown in Table 1. It is also important to note that these six types of vulnerabilities are the mainstream of web vulnerabilities, collectively accounting for over 70% of the total [38].

Our Method. Building upon our observation and verification result, we propose the core concept of our method, which involves *identifying API functions associated with vulnerabilities within API specifications and conducting targeted security testing on those functions*. This approach aims to achieve *vulnerability-oriented API inspection*, thereby facilitating the effective discovery of bugs across a wide range of RESTful APIs. Different from the previous method, it considers the API functionality when identifying and testing for vulnerabilities.

2.3 Challenges and Our Approach

There are three challenges in realizing vulnerability-oriented API inspection in practice.

C1: How to efficiently differentiate between various functional interfaces in an API and identify functions that may have security vulnerabilities? API documentations usually contain a large number of APIs, but only a small portion of them may have security vulnerabilities. Hence, it is crucial to employ a method that can identify candidate APIs, enabling us to enhance testing efficiency and minimize ineffective test requests.

C2: How to generate test case sequences that comply with protocol states and trigger the vulnerable functional interfaces? API invocations are state-based and often require a series of pre-request actions (request sequences) to fulfill the execution

[2]All the vulnerability context information and keywords gathered during this construction process can be found at the link: https://github.com/NSSL-SJTU/VoAPI2/CVE-Information.xlsx.

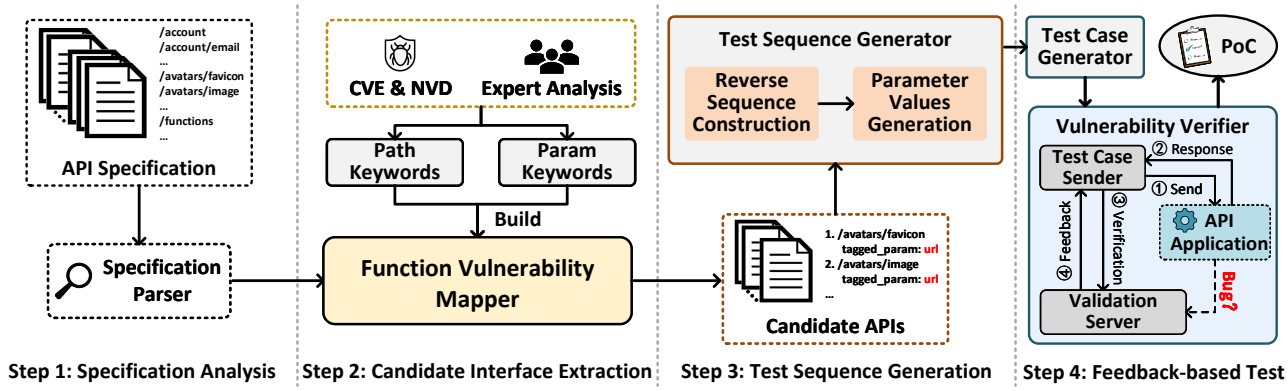

Figure 3: **System Architecture of VoAPI^2.** VoAPI^2 first analyzes the API specification and identifies semantic keywords associated with potentially weak functions within the API specification. It then integrates a suitable corpus and employs a stateful request sequence that aligns with the execution context of the corresponding functions. By utilizing inspection payloads tailored to different vulnerability types, VoAPI^2 can effectively assess these candidate functions for potential vulnerabilities.

dependencies of the API, such as providing required input parameters, in order to ensure the proper execution of the API. Making standalone API calls is usually considered invalid.

C3: How to effectively generate valid test cases based on different functional interfaces and corresponding security vulnerability types? (1) Different types of vulnerabilities have distinct principles and require different detection methods. (2) Overly complex test messages can significantly increase the cost of testing while compromising efficiency and accuracy. (3) Verifying the presence of vulnerabilities may necessitate additional triggering requests. Therefore, the design of test messages and checkers should strive to encompass a comprehensive range of real-world scenarios.

To solve these challenges, we design VoAPI^2 to detect various vulnerabilities in REST APIs effectively. To solve the first challenge, we identify and extract the relevant functional APIs from the API specifications by utilizing semantic keywords (e.g., path keywords and parameter keywords). With a focus on vulnerability analysis, we identify potential vulnerability types and their possible locations within these functional APIs. And then, we employ reverse sequence construction and parameter values generation methods, which is expanding on the techniques used in RESTler and RestCT, to generate test requests that adhere to the protocol's state transitions. This ensures the proper context for executing the candidate APIs. At last, to address different vulnerability types, we create specific test corpora. By combining the test sequences developed for candidate APIs and the marked potential vulnerability locations, we incorporate tailored test data into the corresponding parameters, resulting in comprehensive test cases. The presence of potential vulnerabilities in candidate APIs is assessed by the feedback received from the backend checker, enabling timely adjustments to the testing strategy when necessary.

3 Design

Overview. As shown in Figure 3, our approach begins by identifying semantic keywords associated with potentially weak functions within the API specification. It then integrates a suitable corpus and employs a stateful request sequence that aligns with the execution context of the corresponding functions. By utilizing inspection payloads tailored to different vulnerability types, we can effectively assess these candidate functions for potential vulnerabilities.

3.1 Semantic Keyword Collection

VoAPI^2 utilizes semantic keywords to identify APIs that exhibit specific functionalities. As shown in Table 1, APIs related to file uploads are often vulnerable to arbitrary file uploads. The functionality of an API is typically indicated by its access path, as demonstrated by `/Images/Remote` in Figure 1. On the other hand, the parameters of an API provide specific details about its intended function. For instance, a file upload API is likely to include the keyword "`upload`" in its access path, while the parameters are expected to involve the keyword "`file`".

In this part, we follow a two-step process to extract API access path and parameter keywords. First, we collected 544 vulnerabilities related to APIs with publicly available detailed descriptions from sources such as CVE [39] and NVD [40] to build a dataset. We then clustered these vulnerabilities based on their CWE (Common Weakness Enumeration) IDs and ranked the CWE IDs according to the number of vulnerabilities they encompassed. We selected the six clusters of CWE IDs with the highest proportion. Using this dataset as a foundation, we analyzed the types of API functions associated with these CWE IDs and created mappings between the CWE ID and the corresponding API functionality. Specifically, we

Table 2: **Keywords of API paths and parameters.** Here we list the mappings between Vulnerability Type and API Type as well as the corresponding number of keywords collected in the Semantic Keyword Collection module.

Vulnerability Type	#Keywords		API Type
	API path	API parameter	
SSRF	10	22	Resource Request APIs
Unrestricted Upload	12	8	File Upload APIs
Path Traversal	12	3	Path Processing APIs
Command Injection	12	7	System Configuration APIs
SQL Injection	11	4	Database Operation APIs
XSS	15	12	Text Display APIs

analyzed vulnerabilities within these six CWE clusters and obtained functional descriptions for the vulnerable APIs from various sources, including source code, API documentation comments, and vulnerability descriptions. We conducted word frequency analysis and leveraged expert experience to identify the frequently occurring API functionalities within these six CWE clusters. Finally, we established the mappings as shown in Table 2 (i.e., Vulnerability Type and API Type).

Next, we conducted a statistical analysis of the words found in the API access paths and parameters, and gathered two distinct clusters of semantic keywords: one for API paths and another for parameters. In detail, we extracted the paths and parameters from each vulnerable API in the dataset. We then performed word frequency analysis on these paths and parameters, and sought input from human experts to identify the keyword collections. By incorporating the additional insights and expertise from the experts, we refined and expanded the existing keyword collections. This process allowed us to establish meaningful connections between the keywords, the corresponding API functionalities, and the vulnerability types. These strings serve as semantic keywords that reflect the function of the respective category. The number of each type's keywords is listed in Table 2.

3.2 Candidate Interface Extraction

In this module, we analyze the API specification to identify interface functions that may potentially have vulnerabilities by searching for semantic keywords in the API documentation. The process involves the following steps. Firstly, VoAPI^2 utilizes the RESTler Compile Module to analyze the API specification and generate a grammar file (shown in Listing 1). This grammar file contains information about parameters, responses, and the dependencies between individual requests.

And then, VoAPI^2 parses grammar files based on the following three patterns: (1) the API access path is identified by locating the string "`Endpoint:`"; (2) the API request method is identified by locating the string "`method:`"; (3) The API parameters are identified by finding the line above the `restler_fuzzable_string` and checking if they are surrounded by parentheses "`()`". Taking the grammar file in

```
1  # Endpoint: /Images/Remote, method: Get
2  request = requests.Request([
3      primitives.restler_static_string("GET "),
4      primitives.restler_basepath(""),
5      primitives.restler_static_string("/"),
6      primitives.restler_static_string("Images"),
7      primitives.restler_static_string("/"),
8      primitives.restler_static_string("Remote"),
9      primitives.restler_static_string("?"),
10     primitives.restler_static_string("imageUrl="),
11     primitives.restler_fuzzable_string("fuzzstring", quoted=
           False),
12     primitives.restler_static_string(" HTTP/1.1\r\n"),
13     primitives.restler_static_string("Accept: application/
           json\r\n"),
14     primitives.restler_static_string("Host: localhost\r\n"),
15     primitives.restler_refreshable_authentication_token("
           authentication_token_tag"),
16     primitives.restler_static_string("\r\n"),
17 ],
18 requestId="/Images/Remote"
19 )
```

Listing 1: **Sample of Candidate Interface Extraction.**

Listing 1 as an example, we obtained the API access path `/Images/Remote` (Line 1), the API request method `Get` (Line 1) and the unique parameter of this API `imageUrl` (Line 10).

By applying these patterns, VoAPI^2 is able to extract relevant information from the grammar file and help identify the API functions that are associated with potential vulnerabilities. VoAPI^2 utilizes the semantic keywords generated by the previous module to check for their presence in the paths and parameters of a given API. Based on this analysis, VoAPI^2 categorizes the API functions and assigns them potential vulnerability types (one or more). Our statistical analysis of past API vulnerabilities indicates that all parameters could be vulnerable when an API includes path keywords but lacks parameter keywords. As a result, we employ the following strategy to extract and classify API functions: (1) APIs that solely consist of path keywords without any parameter keywords are identified as candidate APIs, and all parameters of these APIs are marked as test parameters; (2) APIs that contain parameter keywords are also recognized as Candidate APIs, but only the parameters corresponding to the parameter keywords are designated as test parameters.

The extracted candidate APIs are categorized into corresponding functional categories based on the keywords, and potential vulnerability types are mapped, which guides the selection of testing corpus and methods. VoAPI^2 categorizes the candidate API functions and their corresponding potential vulnerability types as shown in Table 2.

3.3 Test Sequence Generation

This module is designed to create a sequence of API requests that accurately reflect the state transitions of the API under testing. Since API calls often depend on the current state, a single request is often inadequate to thoroughly test an API. To address this, VoAPI^2 employs reverse sequence construction techniques to identify other APIs that one candidate API depends on, thereby forming a series of requests that precisely

Algorithm 1 Reverse Sequence Construction Algorithm

```
1:  function GENERATE_SEQUENCE(candidate_api)
2:      S ← candidate_api
3:      i ← -1
4:      while True do
5:          producers ← FIND_PRODUCERS(S[i])
6:          for producer ∈ producers do
7:              if IS_VALID_PRODUCER(producer, S) && not
    IS_DUPLICATE(producer, S) then
8:                  S.INSERT(0, producer)
9:          if S[i] == S[0] then
10:             Break
11:         else
12:             i = i − 1
13:     return S
```

capture the state transitions of the API being tested.

3.3.1 Reverse Sequence Construction

We list the reverse sequence construction algorithm in Algorithm 1 and illustrate the algorithm by using `GET /database/collections/{collectionId}/documents /{documentId}`, an API of Appwrite [5]. It initializes a sequence set `S` with the candidate API (`candidate_api`) and proceeds to reverse-traverse the set *S*, starting from the end (`S[-1]` is `candidate_api`). This process involves identifying all producer APIs of the candidate API. For example, for the sample API, it searches the `responses` field in the API documentation for potential producers. It uses string fuzzy matching to match the producers of the parameters (i.e., `collectionId`, `documentId`) of `candidate_api`. Among these producer APIs, only those that meet the *producer-consumer relationship*, *CRUD semantic* constraints and *resource hierarchy* constraints (we explain the three key definitions and their corresponding constraints used in this algorithm in the following segments.) are retained and appended to the front end of the sequence set `S` (Line 6-8). The algorithm then continues the reverse-traversal process, identifying the producers of the preceding API until it reaches the beginning of the sequence `S` (Line 9-10).

Producer-Consumer Relationship. In the context of the API testing framework, the Producer-Consumer Relationship refers to a rule that governs the execution order of API requests based on their input parameters. If the input parameter of request `A`, which can be located in the Path, Query, Header, Form, or Body, is derived from the response description of request B using heuristic string matching, then request `B` must be executed before request `A`. In other words, request `B` acts as the producer of the input parameter for request `A`. In cases where the same input parameter corresponds to multiple producers with the same path but different request methods, a *priority system* is used to select and add a single producer to the producer set. The priority is determined as follows: `POST` takes precedence over `PUT`, `GET`, and `PATCH`. For the sample API, its all potential producers contains:
P_1="`GET /account`" (P represents Producer),
P_{n_1}="`GET /database/collections`",
P_{n_2}="`POST /database/collections`",
P_{n_3}="`GET /database/collections/{collectionId}`",
P_{n_4}="`POST /database/collections/{collectionId}/documents`",
and so on. And according to the strategy of the *priority system*, the producer P_{n_1} will be removed because it shares the same path as P_{n_2} but has a lower priority.

CRUD Semantics. CRUD is an acronym that stands for `CREATE`, `READ`, `UPDATE`, and `DELETE`. According to the CRUD semantics, a resource (and all its sub-resources) should not be accessed before its creation or after its deletion. In reverse sequence construction, the CRUD semantic constraints only apply to a pair of producers and consumers. For example, because the consumer's request method is `GET`, if there is a producer with a request method of `DELETE`, we need to remove it from the producer set.

Resource Hierarchy. In RESTful APIs, the hierarchical relationships between resources are indicated by the forward slash "/" in the URLs of the resources. For example, `/user/{id}` is a direct sub-resource of `/user`, whereas there is no direct relationship between `/user` and `/team`. We need to remove producers that do not satisfy the resource hierarchy relationship. For example, the producer P_1 that does not have a resource relationship with `candidate_api` will be removed.

After applying these three constraints to filter out invalid producers from the producer set, we are left with three valid producers: "`POST /database/collections`", "`GET /database/collections/{collectionId}`", "`POST /database/collections/{collectionId}/documents`".

3.3.2 Parameter Values Generation

In order to create a concrete request, each input-parameter, such as `imageUrl` (Line 7) in Figure 1, in the sequence needs to be assigned a value domain. VOAPI[2] utilizes one of the following strategies to determine the value domain for each input-parameter based on RestCT [25]:

- CONSUMER. This strategy uses the resources from the producer's response message (i.e., http response message for one request). If an input-parameter `p` corresponds to a resource in the producer's response message, then the resource is used as the value domain for `p` during runtime.
- SPECIFICATION. This strategy uses the values described in the API specification. If an input-parameter `p` has an enum or default field, then the corresponding value is directly used as the value domain for `p`. Otherwise, the strategy searches for all input parameters with the same name as p and selects sample values at random from their example fields to use as the value domain for `p`.
- FORMAT. This strategy uses the values from the preset formatting dictionary. If an input-parameter `p`'s name can

Table 3: **Sample Testing Corpus for Different API Types.**

API Type	Sample in Testing Corpus
Resource Request API	http://IP:PORT/ssrf/{0}
File Upload API	evil files (evil.jsp, evil.asp, evil.php, etc)
Path Processing API	/etc/passwd; C://Windows//win.ini
System Configuration API	curl http://IP:PORT/command/{0}
Database Operation API	1" or "1"="1; SQLMap
Text Display API	<img src='http://IP:PORT/xss/{0}'>

be matched to a preset formatting dictionary, then the corresponding value is used as the value domain for `p`. The VoAPI^2 preset formatting dictionary is mainly designed for parameters that have specific formatting requirements, such as email (which requires a valid email format) and password (which requires a length of at least 8 characters).

- SUCCESS. This strategy uses the values from the previous successful requests (identified by a 200 HTTP status code). If an input-parameter `p` has the same name as an input-parameter from a previous successful request, then the value for that input-parameter is used as the value domain for `p`. If the value for the same input-parameter was generated using the RANDOM strategy, then the value is mutated to increase the diversity of test inputs and avoid creating duplicates.
- RANDOM. This strategy generates a random value to use as the value domain for an input-parameter `p` if none of the above strategies can determine a value domain for `p`.

VoAPI^2 applies these strategies in decreasing priority until a value domain is determined for each input-parameter.

3.4 Feedback-based Testing and Verification

This module consists of two parts: Test Case Generator and Feedback-based Vulnerability Verifier. For each candidate API in the test sequence, Test Case Generator generates valid test cases with the assistance of the test sequence generation module. These test cases are then sent to the API application by the Feedback-based Vulnerability Verifier to detect the presence of corresponding vulnerabilities.

3.4.1 Test Case Generator

The test case generator leverages the output of the Test Sequence Generator module to generate test cases that target potential vulnerabilities in the APIs. Specifically, it generates test cases by inserting the testing corpus into the marked testing parameter positions. For the unmarked parameters, the values assigned by the Test Sequence Generation module are utilized. As shown in Table 3, we collected some test cases and built the testing corpus for different categories of APIs. For instance, to address variations in server-side architectures, we gathered and designed malicious file upload test cases (e.g., `evil.jsp` and `evil.php`) tailored to different programming languages. For another example, we developed a comprehensive set of payloads (e.g., `"../"*9+"etc/passwd"`) that attempt to bypass various types of path checks to explore path traversal vulnerabilities in APIs that handle file paths. More importantly, we design an extensible interface to scale the test corpus. For example, in addition to employing conventional SQL injection test cases (e.g., 1" or "1"="1), we augment the Test Case Generator by integrating one existing powerful detection tool (i.e., SQLMap [41]) to help determine the presence of vulnerabilities with sophisticated corpus.

3.4.2 Feedback-based Vulnerability Verifier

The Feedback-based Vulnerability Verifier primarily comprises two key components: the Test Case Sender and the Validation Server. The Test Case Sender is tasked with transmitting test cases to the target API application and receiving the corresponding responses. It subsequently forwards the received response, candidate API information, and testing corpus data to the Validation Server for in-depth analysis and verification. Meanwhile, the Validation Server is responsible for confirming the presence of vulnerabilities.

For SSRF, XSS, and Command Injection testing corpus, when a vulnerability is present, the vulnerable behavior produced by the testing corpus prompts the API application to send a vulnerability verification request to the Validation Server. The Validation Server determines the existence of the vulnerability based on whether it receives this request. If the request is received, the Validation Server infers that the vulnerability exists. For the remaining types of testing corpus, the Validation Server determines the existence of vulnerabilities based on the verification information sent by the Test Case Sender. For example, if the response content of an upload API test case contains the term "`success`" or the response status code is in the 2xx range, the Validation Server concludes that the API is vulnerable to Unrestricted Upload of File with Dangerous Type. Following this, the Validation Server further checks if the returned information includes the path of the uploaded file. If it does, the Validation Server accesses the file and conducts relevant tests to determine if the malicious file can be successfully executed.

The Validation Server provides feedback on the verification results to the Test Case Generator. When a vulnerability is identified, Test Case Generator proceeds to the next test case. When Validation Server reports that no vulnerability exists, Test Case Generator makes the following determinations. (1) The request method in the test case is POST. (2) The testing corpus corresponds to one of the following: SSRF, XSS, or Command Injection testing corpus. (3) The testing request containing the testing corpus has been executed (determined by a 2xx response status code).

When all of these conditions are met, the Test Case Generator infers that a triggering request is required to further verify

the existence of the vulnerability. The triggering request is sent to further test the existence of the vulnerability. To illustrate this point, consider CVE-2023-27161 as an example. The SSRF vulnerability is located in the `/Repositories` API and can be exploited by embedding a payload in the `url` parameter using a POST request. However, merely sending the POST request does not trigger the vulnerability. In such cases, an additional triggering request, specifically `GET /Packages`, needs to be sent to activate the vulnerable behavior. This triggers the API application to send a vulnerability verification request to the Validation Server, which ultimately leads to the discovery of this vulnerability.

The triggering requests consist of two types: (1) an API with the same path and request method as GET, and (2) all APIs that do not require parameters and have a request method of GET. The selection of these two types of triggering requests is not only aimed at cost savings but also based on historical vulnerability analysis, which has shown a high correlation between these types of APIs and triggering vulnerabilities. If the vulnerability cannot be triggered through these requests, it is recorded as a type of vulnerability that requires manual triggering. By introducing a triggering mechanism, VoAPI^2 can more accurately identify API vulnerabilities that require specific triggering and improve the overall effectiveness of vulnerability testing.

4 Implementation

We have developed a VoAPI^2 prototype system based on RESTler, which consists of 2,700 lines of Python code. The *Semantic Keyword Collection* module is implemented based on Scrapy [42] and Beautiful Soup [43]. The *Candidate Interface Extraction* module is extended from the RESTler's Compile module to support parsing API documentation of multiple formats. In the *Test Sequence Generation* module, the reverse sequence construction is improved based on RESTler's producer search and sequence construction strategy, and the parameter values generation is extended from RestCT's input-parameter value rendering module. In the *Feedback-based Testing and Verification* module, the test case generator is built by modifying the test corpus of RESTler, while the feedback-based vulnerability verifier is implemented using the socket library and by expanding response handling. More importantly, to support various vulnerabilities (e.g., Unrestricted Upload), we have extended the `multipart_formdata` primitive in RESTler Compile. After all, the current prototype system supports multiple API documentations adhering to OpenAPI v2/v3 and can detect six types of API vulnerabilities.

5 Evaluation

We conduct experiments to evaluate the performance of VoAPI^2, and our evaluation targets the following questions:

RQ1 (Vulnerability Detection): How well are the vulnerability identification capability (§5.2) and accuracy (§5.3) of VoAPI^2? Can VoAPI^2 discover vulnerabilities in real-world APIs? (§5.4)
RQ2 (Efficiency): Can VoAPI^2 efficiently generate test cases and explore operations, and how does it perform compared with the state-of-the-art tools? (§5.5)
RQ3 (Ablation Study): How does the vulnerability-oriented strategy of VoAPI^2 affect the testing results? (§5.6)

5.1 Experiment Setup

Compared Tools. We compared our prototype system with several popular vulnerability scanners and state-of-the-art tools in RESTful API testing. It is important to note that these tools are designed for different purposes. First, vulnerability scanners are primarily used to evaluate web applications. Furthermore, these selected vulnerability scanners must support API testing by sending mutated requests to these endpoints and reporting potential vulnerabilities based on the response results. On the other hand, RESTful API testing tools aim to generate better API test cases and discover more bugs (i.e., 500 Internal Server Error). However, they are not designed to reveal in-depth vulnerabilities. We focus on different aspects of VoAPI^2 and compare them with corresponding tools to answer the questions we are concerned about, for example, whether our method can find more bugs than vulnerability scanners. Finally, we select the following two vulnerability scanners and three RESTful API testing tools:

- **Zed Attack Proxy (ZAP)** [44] is an open-source black-box web vulnerability scanner developed by OWASP, primarily used for web vulnerability assessment and penetration testing. We leverage ZAP's resolver to parse API specifications and inspect API services directly.
- **Astra** [45] is a scanner specifically designed for vulnerability scanning of RESTful APIs, testing various vulnerabilities by loading payloads into parameters, HTTP headers, and other locations.
- **RESTler** [12] is an open-source black-box RESTful API testing tool developed by Microsoft. It generates stateful test cases by inferring producer-consumer dependencies between operations and feedback at runtime.
- **RestTestGen** [16] uses data dependencies between operations to generate test cases and deploy two distinct oracles to test RESTful API, that test cases can reveal implementation defects.
- **MINER** [46] uses length-oriented strategies to generate sequence templates and a neural network model to predict key request parameters and provide appropriate parameter values to get a long sequence request in RESTful API testing.

Table 4: **Benchmark Applications**. **#Endpoint** represents the number of endpoints in each Application. **#Download** represents the number that each application is downloaded in the Docker hub [47].

Applications	#Endpoint	Version	Description	#Download
GitLab	358	8.17.0	Code Repository	100M+
Jellyfin	405	10.7.1	CMS	100M+
Appwrite	95	0.9.3	CMS	5M+
Microcks	44	1.17.1	CMS	600K+
Casdoor	121	1.13.0	CMS	20K+
Gitea	325	1.16.7	Code Repository	20K+
Rbaskets	22	1.2.3	Web Service	10K+

Evaluation Benchmarks. We use the following three criteria to select real-world RESTful API applications as our benchmark for evaluation. (1) Having a complete API specification document that can serve as input for all comparative testing tools. (2) The application should have APIs covering multiple orders of magnitude, ranging from small to large-scale. (3) The application should be open-source and widely used in different scenarios. Table 4 illustrates the seven applications we have chosen as our testing subjects. Based on the number of API endpoints, Rbaskets [48], and Microcks [6] are categorized as small-scale API applications, while Appwrite [5] and Casdoor [4] belong to medium-scale API applications. Gitea [49], Jellyfin [3] and GitLab [36], on the other hand, are considered large-scale API applications. All of these applications are easily deployable and supply API services well.

Evaluation Settings. For the target services, we install them with a basic configuration to ensure the proper functioning of their API services. Moreover, we employ the default settings for all compared tools and run each with a maximum time limit (i.e., 5 hours). After each round of testing, we restore the service environment to ensure the consistency of the target service. We conduct all experiments on a server with 4 CPU cores, 8 GB RAM, and the Ubuntu 20.04 LTS operating system, which provides adequate resources for conducting the experiments. In papers such as RESTler, a `bug` is identified by the reception of a 500 HTTP status code following the execution of a request sequence. However, it's important to note that 5xx errors typically refer to server errors during request processing, and not all 5xx errors are necessarily related to security vulnerabilities, such as Denial of Service (DoS). Hence, in our paper, we classify a genuine security vulnerability as a `bug`, distinct from a mere 5xx error.

5.2 Vulnerability Detection (RQ1)

VoAPI^2 identified 26 vulnerabilities, including 7 previously unknown and 19 known security bugs, across seven API applications, encompassing six different types of vulnerabilities. All zero-day bugs have been reported to vendors and fixed; four of them have been assigned CVE IDs. In comparison, the two vulnerability scanning tools discovered fewer vulnerabilities. The three RESTful API testing tools primarily uncovered bugs that resulted in `HTTP 500` errors, with a tiny proportion being security vulnerabilities, as shown in Figure 4. It can be observed that VoAPI^2 detected vulnerabilities in all seven applications, with the highest number and variety of vulnerabilities. In contrast, the vulnerability scanning tools (i.e., ZAP and Astra) did not detect any vulnerabilities in Gitea, Rbaskets and GitLab. The capabilities of these two tools were similar, with slight differences observed only in the Appwrite. A comparison reveals that VoAPI^2 has the strongest ability to identify vulnerabilities compared to ZAP and Astra, particularly in SSRF and XSS vulnerabilities. The other three RESTful API tools discovered only a small number of security vulnerabilities in four applications (i.e., Appwrite, Jellyfin, Microcks and GitLab), and they required manual payload insertion to validate the security vulnerabilities associated with the identified `HTTP 500` errors.

The three RESTful API testing tools were able to discover multiple bugs that resulted in `HTTP 500` errors when detecting targets, as shown in Table 5. The average proportion of vulnerabilities among the `HTTP 500` errors discovered by the other three tools is very low. Through manual analysis, we determined that the percentage of vulnerabilities triggered by these errors is only 5.4% (RESTler), 3% (MINER), and 2.5% (RestTestGen), respectively. Because paths with security vulnerabilities may not necessarily result in internal errors in real-world API applications. The mentioned tools randomly modify parameters' syntax format and content, leading to back-end parameter parsing issues rather than posing a significant security threat. Additionally, incorporating payloads for all types of vulnerabilities without distinguishing the API functionalities would result in low efficiency, as observed in our subsequent ablation experiments (§5.6). Meanwhile, VoAPI^2 found fewer instances of `HTTP 500` errors, mainly due to the targeted nature of VoAPI^2 testing, with fewer API paths being tested, which can be observed from the number of messages. Moreover, VoAPI^2 strives to generate messages that comply with grammatical rules to trigger API behavior and successfully execute the payload that confirms the bug.

We collected a summary of all the security vulnerabilities discovered by various tools, including the vulnerability type, vulnerability position, vulnerable endpoint, and vulnerable parameters, as shown in Table 6. It can be observed that vulnerability-oriented testing methods allow VoAPI^2 to discover corresponding types of security vulnerabilities in APIs with different functionalities. More importantly, VoAPI^2 has discovered the highest number of vulnerabilities and the widest range of vulnerability types compared to other tools. Specifically, VoAPI^2 has identified four types of vulnerabilities, 100% higher than the average of other tools. Meanwhile, VoAPI^2 found 26 vulnerabilities, which was 420% more than the average of other tools.

Among these vulnerabilities, our tool uniquely uncovers

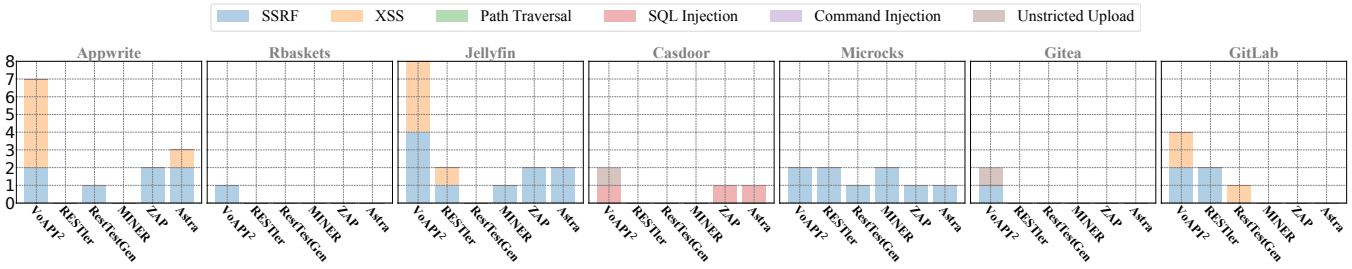

Figure 4: **The vulnerabilities and their types uncovered by different tools on evaluation benchmarks .**

Table 5: **Compared with three RESTful API testing tools. #500** means the number of `HTTP 500` errors found by tools. **#Packet** means the number of all packets sent in testing. **#Ratio** is equal to total #500 divided by total #Packet.

Compare	Appwrite			Casdoor			Gitea			Jellyfin			Microcks			Rbaskets			GitLab			#Ratio
	#500	#Packet	Time	#500	#Packet	Time	#500	#Packet	Time	#500	#Packet	Time	#500	#Packet	Time	#500	#Packet	Time	#500	#Packet	Time	
RESTler	5	88,558	5h	0	165,470	5h	1	297,419	5h	54	2,175	12m07s	19	67,128	5h	0	20,435	5h	32	105,931	5h	0.015%
MINER	4	64,944	5h	0	104,352	5h	1	31,668	5h	68	23,002	5h	13	143,830	5h	0	17,422	5h	13	48,267	5h	0.023%
RestTestGen	1	9,030	19m49s	0	16,340	34m13s	2	58,200	4h37m	63	71,440	4h43m	15	7,340	8m5s	0	2,140	1m32s	41	57,550	1h52m	0.055%
RestTestGen+V	1	6,334	12m42s	0	8,920	13m09s	0	32,550	1h28m	55	48,740	3h57m	11	4,246	5m51s	0	1,550	1m03s	36	43,110	1h33m	0.071%
VoAPI²	1	2,123	1m51s	0	6,987	6m27s	0	9,137	7m05s	23	13,578	10m53s	2	509	24s	0	238	13s	3	1,558	5m03s	0.085%

14 bugs that elude detection by other tools. Based on our analysis, this phenomenon can be attributed to twofold factors. First, in the case of RESTful API testing tools (e.g., RESTler), they often lack a comprehensive testing corpus required for verifying various types of vulnerabilities (e.g., SSRF) within their corresponding endpoints. Thus, these tools often fail to detect a vulnerability if it does not lead to a `HTTP 500` error. Second, scanning tools (e.g., ZAP) face challenges in constructing an appropriate sequence encompassing multiple requests that align with the data dependencies existing among API endpoints. Thus, these tools fail to identify the respective vulnerabilities (e.g., XSS) concealed behind intricate interactions, as exemplified in §5.4.

5.3 Accuracy (RQ1)

We further analyzed the accuracy of VoAPI² and vulnerability scanners. All alerts were manually verified to determine if they were true vulnerabilities, thereby identifying false positives (FP). The false discovery rate (FDR = FP/(FP+TP)) was calculated and presented in Table 7. It can be observed that VoAPI² has lower false positive rates compared to the scanners, which can be attributed to VoAPI²'s vulnerability-oriented strategy and more accurate vulnerability validation strategy.

For XSS vulnerabilities, ZAP and Astra conduct tests on all APIs and determine the existence of an XSS vulnerability solely based on the presence of the XSS payload in the response. However, a significant number of these APIs do not contain display functions. Consequently, the payload would not be displayed on any particular page, which results in a high incidence of false positives. Correspondingly, the vulnerability-oriented testing strategy helps VoAPI² find more API paths related to XSS, which leads to a lower false positive rate.

For path traversal vulnerabilities, ZAP lacks further validation. It solely bases its determination of a vulnerability's existence on whether the test request returns a 2xx status code. On the other hand, VoAPI² goes a step further by analyzing the response content, checking for the presence of content corresponding to the test payload to confirm the existence of the vulnerability. For instance, when the test payload is `"/etc/passwd"`, VoAPI² will match the response content for characteristic strings (e.g., root) to validate the vulnerability.

In terms of VoAPI², false positives may occur when checking a particular XSS (i.e., stored XSS) and the implicit unrestricted upload vulnerability. In these scenarios, we need to trigger vulnerability manually and check whether vulnerability exists or not. Moreover, we thoroughly discuss the root causes and the corresponding improvement ways in §6.

5.4 Real-world Vulnerabilities (RQ1)

We applied VoAPI² to discover security vulnerabilities in real API applications and found various vulnerabilities. As shown in Table 6, we identified more types of vulnerabilities compared to scanning tools, especially on XSS and SSRF. The main reason is that our method can generate effective request sequences that access multiple endpoints in proper order, allowing us to trigger deeper vulnerabilities.

Case Study: XSS. The XSS vulnerability (CVE-2022-2925) was discovered in the Appwrite application. As shown in Figure 5, this bug exists in five API endpoints (`/teams`, `/users`, `/functions`, `/database/collections`, `/teams/{teamId}/memberships`), and the vulnerable arguments in these endpoints are `"name"` marked in blue color. In our experiments, both ZAP and Astra face significant difficulties. While they may encounter problems in gen-

Table 6: **RESTful API vulnerabilities identified by all tools.** For **Producer**, ✔ indicates this API endpoint has a producer, while ✗ indicates it doesn't have; For **0-day**, ✔ indicates that this vulnerability is a zero-day vulnerability while ✗ indicates it is not; For **Vulnerability Identification Tools**, ✔ indicates that this tool can identify the vulnerability, ✔ indicates only this tool can identify the vulnerability, while ✗ indicates it cannot.

Application	Version	Path	Parameter	Type	Producer	0-day	Bug-IDs	Vulnerability Identification Tools					
								VoAPI^2	RESTler	RestTestGen	MINER	ZAP	Astra
Appwrite	0.9.3	/avatars/favicon	url	SSRF	✗	✔	CVE-2023-27159	✔	✗	✗	✗	✔	✔
	0.9.3	/avatars/image	url	SSRF	✗	✔	CVE-2023-27159	✔	✗	✔	✗	✔	✔
	0.9.3	/teams	name	XSS	✗	✗	CVE-2022-2925	✔	✗	✗	✗	✗	✔
	0.9.3	/teams/{teamId}/memberships	name	XSS	✔	✗	CVE-2022-2925	✔	✗	✗	✗	✗	✗
	0.9.3	/database/collections	name	XSS	✗	✗	CVE-2022-2925	✔	✗	✗	✗	✗	✗
	0.9.3	/functions	name	XSS	✗	✗	CVE-2022-2925	✔	✗	✗	✗	✗	✗
	0.9.3	/usrs	name	XSS	✗	✗	CVE-2022-2925	✔	✗	✗	✗	✗	✗
Rbaskets	1.2.3	/api/baskets/{name}	forward_url	SSRF	✗	✔	CVE-2023-27163	✔	✗	✗	✗	✗	✗
Jellyfin	10.7.1	/Images/Remote	imageUrl	SSRF	✗	✗	CVE-2021-29490	✔	✔	✗	✔	✔	✔
	10.7.1	/Items/RemoteSearch/Image	imageUrl	SSRF	✗	✗	CVE-2021-29490	✔	✗	✗	✗	✔	✔
	10.7.1	/Items/{itemId}/RemoteImages/Download	imageUrl	SSRF	✔	✗	CVE-2021-29490	✔	✗	✗	✗	✗	✗
	10.7.1	/Repositories	Url	SSRF	✗	✔	CVE-2023-27161	✔	✗	✗	✗	✗	✗
	10.7.1	/Playlists	name	XSS	✗	✗	CVE-2023-23636	✔	✗	✗	✗	✗	✗
	10.7.1	/Repositories	name	XSS	✗	✗	CVE-2022-35910	✔	✗	✗	✗	✗	✗
	10.7.1	/Collections	name	XSS	✗	✗	CVE-2023-23635	✔	✔	✗	✗	✗	✗
	10.7.1	/Startup/User	Name	XSS	✗	✔	1 unassigned	✔	✗	✗	✗	✗	✗
Casdoor	1.13.0	/api/get-organizations	field	SQL Injection	✗	✗	CVE-2022-24124	✔	✗	✗	✗	✔	✔
	1.13.0	/api/upload-resource	fullFilePath	Unrestricted Upload	✗	✗	CVE-2022-38638	✔	✗	✗	✗	✗	✗
Microcks	1.17.1	/jobs	repositoryUrl	SSRF	✗	✔	1 unassigned	✔	✔	✔	✔	✗	✗
	1.17.1	/artifact/download	url	SSRF	✗	✔	1 unassigned	✔	✔	✗	✔	✔	✔
Gitea	1.16.7	/repos/{owner}/{repo}/contents/{filepath}	content	Unrestricted Upload	✔	✗	CVE-2022-1928	✔	✗	✗	✗	✗	✗
	1.16.7	/repos/{owner}/{repo}/hooks	url	SSRF	✔	✗	CVE-2018-15192	✔	✗	✗	✗	✗	✗
GitLab	8.17.0	/v3/hooks	url	SSRF	✗	✗	CVE-2018-8801	✔	✔	✗	✗	✗	✗
	8.17.0	/v3/projects	import_url	SSRF	✗	✗	CVE-2022-0249	✔	✔	✗	✗	✗	✗
	8.17.0	/v3/projects/{id}/deploy_keys	title	XSS	✔	✗	CVE-2022-2230	✔	✗	✗	✗	✗	✗
	8.17.0	/v3/projects/{id}/milestone	title	XSS	✔	✗	CVE-2022-1190	✔	✗	✔	✗	✗	✗

Table 7: **FDR identified by different tools.**

	Appwrite	Rbaskets	Jellyfin	Casdoor	Microcks	Gitea	GitLab	**FP Rate**
VoAPI^2	4/11	1/2	5/13	2/4	1/3	3/5	3/7	**42.22%**
ZAP	6/8	2/2	9/11	5/6	7/8	5/5	2/2	**85.71%**
Astra	5/8	0/0	4/6	3/4	2/3	1/1	1/1	**69.57%**

```
1. /teams:
2.   POST:
3.     RequestBody:
4.       name
5.     responses:
6.       $id:
7.         description: Team ID.
8. /teams/{teamId}/memberships:
9.   POST:
10.    parameters:
11.      description: Team unique ID.
12.      in: path
13.      name: teamId
14.    RequestBody:
15.      name
```

```
16. /databases/collections:
18.   POST:
19.     RequestBody:
20.       name
21. /users:
22.   POST:
23.     RequestBody:
24.       name
25. /functions:
26.   POST:
27.     RequestBody:
28.       name
```

Figure 5: **XSS vulnerabilities in Appwrite.**

erating appropriate parameter values for some APIs, the more critical issue is their inherent inability to request `/teams/{teamId}/memberships` (Line 8). Because the presence of this XSS vulnerability on this endpoint requires accessing `/teams` (Line 1) with the POST method before, then parsing the `id` value (Line 6) from the response, and assigning it to the path parameter `teamId` (Line 13). Obviously, ZAP and Astra cannot infer this data dependency and construct a suitable sequence that includes these requests. In contrast, VoAPI^2 can extract the context in which the target API endpoint operates and generate similar request sequences to discover such vulnerabilities.

5.5 Efficiency (RQ2)

We check the efficiency of VoAPI^2 in two respects: (1) whether VoAPI^2 can efficiently generate request sequences for testing API endpoints, compared with state-of-the-art RESTful testing methods (e.g., RestTestGen); (2) whether VoAPI^2 is more efficient than web scanners (e.g., ZAP) in finding vulnerabilities.

Operation Coverage. In this experiment, we used VoAPI^2's Test Sequence Generation (TSG) module to generate a request sequence for each endpoint. We selected three approaches to compare the API's operation coverage with TSG. If a tool is able to generate at least one valid request sequence for an

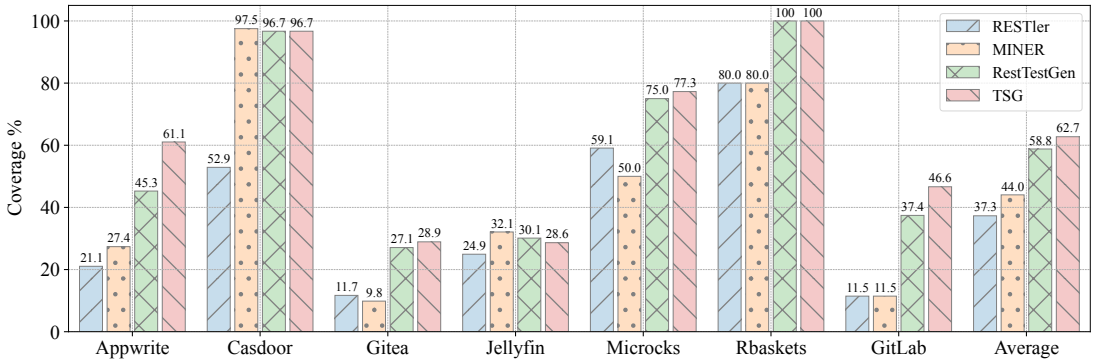

Figure 6: **Compared with RESTful API testing tools on coverage.**

Table 8: **Compared with scanners on testing time consume.**

	Appwrite	Casdoor	Gitea	Jellyfin	Microcks	Rbaskets	GitLab	#Total
ASTRA	4:30m	18:07m	40:32m	103:28m	8:02m	4:34m	56:16m	235:19m
ZAP	10:23m	22:02m	51:27m	88:02m	10:22m	7:57m	182:41m	372:54m
ZAP+V	6:39m	8:05m	16:33m	6:11m	1:31m	0:40m	24:07m	63:46m
VoAPI^2	1:51m	6:27m	7:05m	10:53m	0:24m	0:13m	5:03m	31:05m

API endpoint that results in a 2xx status code response, we consider that endpoint to be covered.

Figure 6 exhibits our experimental results, which evidently demonstrates that TSG performs better than the current state-of-the-art RESTful testing tools in terms of generating valid request sequence for target API endpoint. More specifically, TSG achieves the highest endpoint coverage on Appwrite, Gitea, Microcks, Rbaskets and GitLab. Specifically, the average coverage of TSG on these products can reach 62.7%, which is more than RESTler's 37.3%, MINER's 44.0%, RestTestGen's 58.8%. We attribute TSG's outstanding performance to its superior ability to construct dependencies for target API endpoints compared to other tools. For the sample API in §3.3.1, we found that none of the tools except VoAPI^2 can construct the correct dependency relation for this API.

Compared With Web Scanners. Table 8 shows time consumed by the web vulnerability scanners and VoAPI^2. The test time of VoAPI^2 is much less than the other two tools. Specifically, it took VoAPI^2 31min05s to test the seven products, while Astra took 235min19s, and ZAP took 372min54s. This is because of VoAPI^2's vulnerability-oriented strategy, which just tests candidate API endpoint within targeted payloads based on the feature of its path and parameters, greatly enhancing testing efficiency. In contrast, tools like ZAP and Astra test parameters and paths indiscriminately within all payloads, resulting in significantly longer testing times.

5.6 Ablation Study (RQ3)

To investigate how the vulnerability-oriented strategy of VoAPI^2 impacts the result of testing, we perform an ablation study. Thus, we remove the Candidate Interface Extraction (§3.2) module from VoAPI^2 to ensure all API paths and parameters are checked with equal priority. We refer to this modified tool as VoAPI^2-V, which serves as a benchmark for comparative analysis against VoAPI^2.

In testing, no new vulnerabilities were found by VoAPI^2-V. Table 9 shows the time VoAPI^2 and VoAPI^2-V need to expose each vulnerability. Obviously, VoAPI^2-V takes several orders of magnitude more time, because VoAPI^2-V must indiscriminately check every endpoint of API. When a potentially vulnerable path is located near the end of the specification document, VoAPI^2-V takes much longer to reach it. For example, in the case revealed in `/Startup/User` of Jellyfin, VoAPI^2 found this vulnerability on this API in 10min30s, while VoAPI^2-V takes 495min28s. This is because VoAPI^2 can aim this endpoint directly from a large number of APIs, while VoAPI^2-V must test all APIs sequentially. Furthermore, VoAPI^2-V tests every parameter of the API using all available test payloads, which consumes a significant amount of time. In contrast, VoAPI^2 selects targeted payloads for testing the portion parameters marked with directed strategy, which saves a lot of time.

Furthermore, we integrate the keyword shortlisting process into existing tools (i.e., RestTestGen and ZAP). Specifically, these tools are now only applied to candidate APIs extracted from VoAPI^2. The integrated models resulting from this integration are named RestTestGen+V and ZAP+V. In order to compare the performance of these models, we conduct experiments focusing on efficiency and bug discovery.

In terms of test efficiency, ZAP+V inspects solely the API endpoints that are filtered by keywords. As a result, the testing time is significantly reduced compared to the original ZAP, as showcased in Table 8. This highlights the effectiveness of keyword lists when used with ZAP. On the other hand, for RestTestGen+V, we expanded the testing scope to include not only the API endpoints filtered by keywords but also those that have data dependencies with them. The testing results presented in Table 5 demonstrate that these integrated models exhibit lower time consumption and send fewer packets compared to their respective original tools.

Table 9: **Compared with VoAPI^2-V on time consume.**

Application	Bug-IDs	Path	Time-to-Exposure	
			VoAPI^2	VoAPI^2-V
Appwrite	CVE-2023-27159	/avatars/favicon	2.81s	1min31s
	CVE-2023-27159	/avatars/image	2.90s	1min39s
	CVE-2022-2925	/teams	53.42s	26min30s
	CVE-2022-2925	.../memberships	1min03s	34min36s
	CVE-2022-2925	/database/collections	15.28s	17min18s
	CVE-2022-2925	/functions	1min39s	60min16s
	CVE-2022-2925	/users	1min30s	48min8s
Rbaskets	CVE-2023-27163	/api/baskets/{name}	2.08s	38s
Jellyfin	CVE-2021-29490	/Images/Remote	5.79s	225min46s
	CVE-2021-29490	/Items/.../Image	7.26s	246min29s
	CVE-2021-29490	/Items/.../Download	6.68s	228min13s
	CVE-2023-27161	/Repositories	10.24s	322min38s
	CVE-2023-23636	/Playlists	9min09s	368min04s
	CVE-2022-35910	/Repositories	8min57s	322min47s
	CVE-2023-23635	/Collections	8min25s	35min11s
	1 unassigned	/Startup/User	10min30s	495min28s
Casdoor	CVE-2022-24124	/api/get-organizations	5min04s	43min28s
	CVE-2022-38638	/api/upload-resource	2min33s	75min50s
Microcks	1 unassigned	/jobs	1.90s	11min19s
	1 unassigned	/artifact/download	2min06	90min55s
Gitea	CVE-2022-1928	/repos/.../{filepath}	3.61s	94min41s
	CVE-2018-15192	/repos/.../hooks	2.49s	36min05s
GitLab	CVE-2018-8801	/v3/hooks	1min12s	236min05s
	CVE-2022-0249	/v3/projects	1min17s	20min17s
	CVE-2022-2230	/v3/.../deploy_keys	1min49s	53min33s
	CVE-2022-1190	/v3/.../milestone	2min36s	145min41s

Regarding bug discovery, RestTestGen+V identified fewer `HTTP 500` errors, as indicated in Table 5. This can be attributed to the fact that the number of candidate APIs is significantly smaller than the original APIs, and our vulnerability-oriented strategies are not specifically designed to detect `HTTP 500` errors. Furthermore, RestTestGen+V lacks a suitable test corpus that is necessary to trigger potential vulnerabilities in the selected APIs. Therefore, it cannot benefit from keyword shortlisting. Similarly, ZAP+V does not discover more vulnerabilities than ZAP because it is unable to construct a suitable sequence comprising valid requests to satisfy dependencies among candidate APIs.

In conclusion, although keyword selection can enhance testing efficiency, it does not lead to better effectiveness when directly applied to traditional RESTful API testing tools and vulnerability scanners.

6 Discussion

In this section, we discuss the limitations of VoAPI^2 and explore the improvement direction in the future.

Scale Ability of Bug Verification. The current state of feedback-based vulnerability verification is limited by the concise validation methods based on the testing corpus for different bugs. That brings false positives when a bug must be triggered interactively, like the stored XSS and implicit upload bug we mentioned before (§5.3). Consequently, VoAPI^2 may

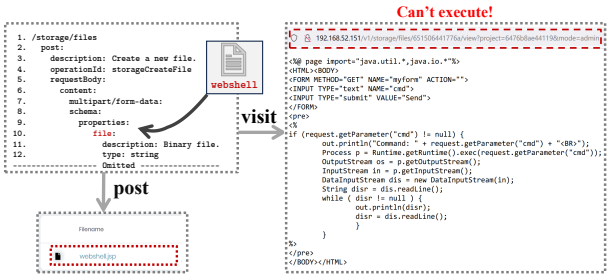

Figure 7: **False positive of unrestricted upload.**

fail to judge these vulnerabilities correctly and require manual efforts. To address this issue, we plan to design an extensible interface of the Feedback-based Vulnerability Verifier (§3.4.2) for specific vulnerabilities. Furthermore, we can continuously extend vulnerability verification methods, for example, to heuristically generate upload paths based on the context of implicit unrestricted upload of dangerous files, thereby enabling VoAPI^2 to discover vulnerabilities more generally and automatically.

Manual Effort. In identifying and validating vulnerabilities, VoAPI^2 requires human intervention in two scenarios: implicit unrestricted upload vulnerabilities and stored Cross-Site Scripting (XSS).

In the case of unrestricted upload vulnerabilities, VoAPI^2 submits a malicious file to check whether the bug exists or not. If the server's response omits the file's access path, VoAPI^2 is unable to automatically retrieve this path, which hinders further tests to assess whether the file can be executed. We present an example of a false positive of unrestricted upload in Figure 7. It can be seen that VoAPI^2 automatically deduces that the arbitrary file upload API interface permits unrestricted uploads of malicious files. However, it lacks the prior knowledge necessary to automatically retrieve the unique access paths of these uploaded files, such as identifying the location of the file access interface or how to obtain random file IDs like "`651506441776a`". This prior knowledge is not standardized and tends to vary across different API services, making automatic retrieval of access paths a complex challenge. As a result, manual intervention is currently necessary to visit relevant pages, identify the page containing the file access interface, and manually test whether the server can parse the malicious file. If the server restricts the parsing of such malicious files, the upload vulnerability remains untriggered, leading to a false positive.

In terms of stored XSS (also known as persistent XSS), VoAPI^2 takes a POST request path as a potential XSS vulnerability after getting a response packet with a 2xx status code, which also contains an XSS payload. However, similar to the challenges unrestricted upload checking faces, VoAPI^2 cannot automatically determine which storage page stores

XSS payloads. Thus, this XSS payload can only be triggered by manually browsing the relevant pages guided by expert knowledge. If the XSS behavior is not triggered even after browsing all relevant pages, it is considered a false positive.

Support More Specifications. Currently, VoAPI^2 only supports OpenAPI-formatted API specifications as input, while many applications do not provide such documentation and instead offer API usage instructions in other files (e.g., HTML). In order to expand VoAPI^2's capabilities, we propose to use machine learning or natural language processing methods to identify and extract API information from these usage instructions. Finally, VoAPI^2 can inspect more API applications.

Machine Learning Method. VoAPI^2 classifies API interfaces based on statistics of vulnerability characteristics and human expertise. Although it is a reliable method to support our prototype system in discovering real-world security vulnerabilities, we believe there is a trend for artificial intelligence technology to replace these expert experiences in the future. For example, advances in natural language processing (e.g., large language model) have demonstrated powerful text analysis capabilities, promising to audit API documents for interface classification more effectively.

7 Related Work

RESTful API Security. The security operations teams of popular API services can discover attacks against their services. The Facebook team [10] identified that attackers could exploit authorization vulnerabilities in a specific API endpoint, allowing third-party applications to access users' private photos. The Twitter team [11] discovered that attackers could utilize a specific API endpoint that associates phone numbers with account names to systematically launch widespread attacks and infer the target users' phone numbers and account names. Researchers note that there are typically two categories of bugs in RESTful APIs, including that result in service unavailability and those related to web security [50]. The former is often caused by syntactically incorrect parameters in API request data, leading to the backend being unable to process them and resulting in `HTTP 500` errors. Most of these bugs typically indicate that the server is unable to handle the current request properly. The latter category arises due to similarities between the backend processing of RESTful APIs and traditional web services, resulting in web bugs when requests are mishandled.

RESTful Service Testing Methods. Regarding the potential bugs in RESTful services, researchers have proposed numerous automated testing methods that can generate sequence requests, assess various endpoints of the API, and determine if there are any issues based on the service's responses. RESTler [12] is a stateful API black-box testing technique that generates stateful test cases by inferring producer-consumer dependencies between different API endpoints, allowing it to reach the "deep" states of the target API service. RestTestGen [16] utilizes data dependencies between operations to generate test cases, while RESTest [15] considers dependencies among parameters and generates test cases that satisfy specified dependency relationships using constraint solving and random input generation. MINER [46] uses a neural network model to predict critical request parameters. However, the aforementioned black-box testing tools all focus on better parsing API specifications, generating request sequences that comply with the API interface protocol state, and generating effective test cases that cover more API functionalities. They aim at the usability of API functions based on whether the API service returns `HTTP 500` errors but do not specifically target security vulnerabilities.

Penetration Techniques. Due to the close similarity in implementation between the backend of RESTful APIs and traditional web services, researchers aim to expand web penetration techniques to conduct vulnerability detection in RESTful APIs. Zed Attack Proxy (ZAP) [10] is primarily used for web vulnerability assessment and penetration testing. Its current OpenAPI extension supports parsing API specifications, enabling web vulnerability testing for API services. Astra [45] is a tool specifically designed to scan vulnerabilities in RESTful APIs. It verifies common web vulnerabilities by injecting payloads into parameters, HTTP headers, and other locations. However, these tools have significant limitations that they lack the ability to interpret the state of the API protocol. They can only generate requests for individual API endpoints, limiting the scope to test each endpoint separately and preventing the detection of vulnerabilities in API services composed of multiple endpoints. NAUTILUS [50] incorporates comment policies into API specifications to handle operation relationships and parameter generation, resulting in meaningful sequences of operations and the discovery of corresponding API security vulnerabilities. However, due to its oversight of vulnerability characteristics, NAUTILUS focuses on a limited range of vulnerability types, mainly handling injection vulnerabilities.

8 Conclusion

In summary, we propose VoAPI^2, a novel inspection framework, to apply a vulnerability-oriented strategy to inspect RESTful APIs. Based on the insight that the type of vulnerability hidden in an API interface is strongly associated with its functionality, VoAPI^2 can directly expose vulnerabilities in RESTful APIs. We first track commonly used strings as keywords to identify APIs' functionality. Then, we generate a stateful and suitable request sequence to inspect the candidate API function within a targeted payload. Finally, we verify whether vulnerabilities exist or not through feedback-based testing. Our evaluation result shows that VoAPI^2 demonstrates higher efficiency and effectiveness in bug discovery than state-of-the-art tools, including RESTful API testing and penetration methods.

Acknowledgments

We thank the anonymous reviewers of this work for their helpful feedback. This research was supported, in part, by National Natural Science Foundation of China under Grant No. 62372297, Science and Technology Commission of Shanghai Municipality Research Program under Grant No. 20511102002, National Radio and Television Administration Laboratory Program (TXX20220001ZSB002).

References

[1] Amazon. AWS. https://aws.amazon.com/.

[2] Microsoft. Azure. https://azure.microsoft.com/en-us/.

[3] Jellyfin. https://jellyfin.org/.

[4] Casdoor. https://casdoor.org/.

[5] Appwrite. https://appwrite.io/.

[6] Microcks. https://microcks.io/.

[7] CVE-2021-3044. https://nvd.nist.gov/vuln/detail/CVE-2021-3044.

[8] CVE-2019-12643. https://nvd.nist.gov/vuln/detail/CVE-2019-12643.

[9] Representational state transfer. https://en.wikipedia.org/wiki/Representational_state_transfer.

[10] Tomer Bar. Notifying our Developer Ecosystem about a Photo API Bug. https://developers.facebook.com/blog/post/2018/12/14/notifying-our-developer-ecosystem-about-a-photo-api-bug/, 2018.

[11] Twitter. An Incident Impacting Your Account Identity. https://privacy.twitter.com/en/blog/2020/an-incident-impacting-your-account-identity, 2020.

[12] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758, Montreal, QC, Canada, 2019. IEEE.

[13] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Automatic generation of test cases for REST APIs: a specification-based approach. In *International Enterprise Distributed Object Computing Conference*, pages 181–190, 2018.

[14] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. QuickREST: property-based test generation of OpenAPI-described RESTful APIs. In *International Conference on Software Testing, Validation and Verification (ICST)*, pages 131–141, 2020.

[15] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. RESTest: black-box constraint-based testing of RESTful web APIs. In *International Conference on Service-Oriented Computing (ICSOC)*, pages 459–475, 2020.

[16] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. RestTestGen: automated black-box testing of RESTful APIs. In *International Conference on Software Testing, Validation and Verification (ICST)*, pages 142–152, 2020.

[17] Tobias Fertig and Peter Braun. Model-driven testing of RESTful APIs. In *International Conference on World Wide Web: Companion*, pages 1497–1502, 2015.

[18] Andrea Arcuri. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1):1–37, 2019.

[19] Andrea Arcuri. Automated blackbox and whitebox testing of RESTful APIs with EvoMaster. *IEEE Software*, 2020.

[20] Andrea Arcuri and Juan P Galeotti. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–31, 2020.

[21] Andrea Arcuri and Juan P Galeotti. Enhancing search-based testing with testability transformations for existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–34, 2021.

[22] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations. *arXiv preprint arXiv:2005.11498 [cs.SE]*, 2020.

[23] S. T. Liu. Coverage guided fuzzing in python-based web server. Master's thesis, Institute of Computer Science and Engineering, National Chiao Tung University, Hsinchu, Taiwan, 2019.

[24] Runtime application self-protection. https://en.wikipedia.org/wiki/Runtime_application_self-protection.

[25] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. Combinatorial testing of RESTful APIs. In *Proceedings*

of the 44th International Conference on Software Engineering (ICSE '22), pages 426–437, New York, NY, USA, 2022. Association for Computing Machinery.

[26] AppSpider. https://www.rapid7.com/products/appspider.

[27] Qualys Web Application Scanning (WAS). https://www.qualys.com/apps/web-app-scanning/.

[28] TnT-Fuzzer. https://github.com/Teebytes/TnT-Fuzzer.

[29] APIFuzzer. https://github.com/KissPeter/APIFuzzer.

[30] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. Differential regression testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*, pages 312–323, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Y. Luo et al. RestSep: Towards a Test-Oriented Privilege Partitioning Approach for RESTful APIs. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 548–555, Honolulu, HI, USA, 2017.

[32] Y. Luo, H. Zhou, Q. Shen, A. Ruan, and Z. Wu. RestPL: Towards a Request-Oriented Policy Language for Arbitrary RESTful APIs. In *2016 IEEE International Conference on Web Services (ICWS)*, pages 666–671, San Francisco, CA, USA, 2016.

[33] Swagger. https://swagger.io/.

[34] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Checking Security Properties of Cloud Service REST APIs. In *13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 387–397, Porto, Portugal, 2020. IEEE.

[35] CVE-2021-29490. https://nvd.nist.gov/vuln/detail/CVE-2021-29490.

[36] Gitlab. https://gitlab.com/.

[37] Bing Maps. https://www.bing.com/maps.

[38] Cybersecurity Vulnerability Landscape Report. https://www.h3c.com/cn/d_202303/1796824_30003_0.htm.

[39] CVE. https://cve.mitre.org/.

[40] NVD. https://nvd.nist.gov/.

[41] Bernardo Damele A. G. and Miroslav Stampar. sqlmap. https://github.com/sqlmapproject/sqlmap.

[42] Scrapy. https://scrapy.org/.

[43] Beautiful Soup. https://www.crummy.com/software/BeautifulSoup/.

[44] OWASP Zed Attack Proxy. https://www.zaproxy.org/.

[45] ASTRA. https://github.com/flipkart-incubator/Astra.

[46] Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Chen, and Raheem Beyah. Miner: A hybrid data-driven approach for rest api fuzzing. In *32th USENIX Security Symposium (USENIX Security 23)*, 2023.

[47] Docker Hub. https://hub.docker.com/.

[48] Rbaskets. https://rbaskets.in/.

[49] Gitea. https://gitea.io/.

[50] Deng Gelei, Zhang Zhiyi, Li Yuekang, Liu Yi, Zhang Tianwei, Liu Yang, Yu Guo, and Wang Dongjin. Nautilus: Automated restful api vulnerability detection. In *32th USENIX Security Symposium (USENIX Security 23)*, 2023.