International Journal of Information Security (2025) 24:73
https://doi.org/10.1007/s10207-024-00979-w

REGULAR CONTRIBUTION

Fuzzing frameworks for server-side web applications: a survey

I Putu Arya Dharmaadi[1,2] · Elias Athanasopoulos[3] · Fatih Turkmen[1]

Abstract

There are around 5.3 billion Internet users, amounting to 65.7% of the global population, and web technology is the backbone of the services delivered via the Internet. To ensure web applications are free from security-related bugs, web developers test the server-side web applications before deploying them to production. The tests are commonly conducted through the interfaces (i.e., Web API) that the applications expose since they are the entry points to the application. Fuzzing is one of the most promising automated software testing techniques suitable for this task; however, the research on (server-side) web application fuzzing has been rather limited compared to binary fuzzing which is researched extensively. This study reviews the state-of-the-art fuzzing frameworks for testing web applications through web API, identifies open challenges, and gives potential future research. We collect papers from seven online repositories of peer-reviewed articles over the last ten years. Compared to other similar studies, our review focuses more deeply on revealing prior work strategies in generating valid HTTP requests for attack surface exploration, utilising security feedback from the Web Under Tests (WUTs), and expanding input spaces to uncover more security-related bugs. The findings of this survey indicate that several crucial challenges need to be solved, such as less effective web instrumentation and the complexity of handling microservice applications. Furthermore, some potential research directions are also provided, such as fuzzing for web client programming. Ultimately, this paper aims to give a good starting point for further research in web API fuzzer.

Keywords Fuzzing · Web application · Web API · Survey

1 Introduction

Fuzzing is an automated software testing technique that focuses on finding bugs, errors, or faults in the software under test (SUT) [93] by creating many test cases in the form of malformed/semi-malformed inputs and feed them into the SUT without requiring human intervention. The inputs are produced by employing a variety of techniques (e.g., mutation) with the idea of triggering software vulnerabilities that manifest themselves in the form of a crash. Since fuzzing has excellent potential to discover security-related vulnerabilities [103], fuzzing of binary applications has been studied extensively that led to the development of a plethora of binary fuzzers such as American Fuzzy Lop (AFL) [94] and libFuzzer [58], and a yearly competition [32].

While the research on binary fuzzing has made impressive strides, the research on fuzzing of web applications is only recently picking up and the existing work to date has been scattered. This is surprising because web applications are ubiquitous as most governments and companies provide their services through the World Wide Web that allows access from a plethora of devices including desktop computers, mobile phones and tablets. According to a study by Cloudflare in December 2021 [50], around 25% of all API traffic in its network is web related traffic and is twice as likely to be blocked than other API requests. More strikingly, the development and maintenance of web applications has a huge market size (56B$ in 2021 and is expected to rise up to 89B$ by 2027 [1]) so the testing of web applications for security vulnerabilities has a significant commercial value.

Web application fuzzing has its own challenges, and many techniques from binary fuzzing are not directly applicable.

✉ I Putu Arya Dharmaadi
arya.dharmaadi@rug.nl

Fatih Turkmen
f.turkmen@rug.nl

[1] University of Groningen, Groningen, The Netherlands

[2] Udayana University, Bali, Indonesia

[3] University of Cyprus, Nicosia, Cyprus

[1] https://www.businessresearchinsights.com/market-reports/web-development-market-109039

Published online: 05 February 2025

For instance, web APIs only accept valid HTTP requests to be executed, which means that the malformed test cases commonly produced by fuzzers will be rejected by web servers. This condition made some of the existing web API fuzzers (e.g., RESTler [12] and RestTestGen [21]) utilise OpenAPI specifications to create templates for HTTP requests which are then rendered with the correct values to form valid HTTP request sequences. However, there is a lack of research exploring novel techniques to uncover various web vulnerabilities as listed in the OWASP API Security Risks [74].

In this paper, we conduct a review to summarise the existing works on web API fuzzing while analysing their strengths and limitations. From a more practical point of view, our study reveals how prior works designed strategies to generate valid HTTP requests for attack surface exploration, utilise security feedback from the web under test (WUT), and expand relevant input spaces to uncover more security-related bugs, which are all crucial for the effectiveness of fuzzing. Finally, we identify five areas of challenges that need to be addressed in order to improve fuzzing effectiveness and efficiency in web applications and four potential research directions that can be massively studied in the future.

1.1 Research questions

In organizing our paper, we employed the following research questions (RQ), which are grouped according to various objectives.

- Producing valid HTTP requests for attack surface exploration
  - **RQ1**: What techniques are used to generate valid request templates?
  - **RQ2**: How are the request templates rendered?
- Observing security feedback
  - **RQ3**: What types of security-relevant feedback are collected from the WUT?
  - **RQ4**: How are WUTs instrumented to capture security-critical behaviors and responses?
  - **RQ5**: What vulnerabilities are observed?
- Input space expansion
  - **RQ6**: How are the existing input spaces expanded to uncover deeper or previously unseen security vulnerabilities?
- Security-focused evaluation
  - **RQ7**: What security benchmarks are used for empirical evaluations?
- Open challenges
  - **RQ8**: What open challenges are identified?

1.2 Scope, related works, and contributions

Web application fuzzing is similar to web vulnerability scanning in terms of their approaches. Since we aim to review fuzzing strategies, this section first clarifies the differences between the fuzzer and the scanner, in general, to make it clear. In addition, since the fuzzing-related review topic is packed, this section then stresses the uniqueness of this work compared to other survey papers. Finally, a summary of our contributions is provided in the last part.

1.2.1 Fuzzer vs vulnerability scanner

Both fuzzers and vulnerability scanners work automatically to find software vulnerabilities. However, there are certain differences between as we discuss in this section. While a fuzzer produces plenty of malformed/semi-malformed inputs to make a software crash and let the software developers identify any vulnerability including completely unknown, i.e., **zero-day vulnerability**, behind the crash [57] [48], a scanner injects web APIs with malicious payloads sourced from manually curated templates for finding **pre-defined security vulnerabilities**, such as SQL injection and XSS [3] [82]. Nevertheless, this distinction becomes increasingly blurred because, recently, there have also been vulnerability-driven fuzzers aiming to find certain vulnerabilities rather than just a crash (explained in Section 4.2.2). Another difference is that a fuzzer performs dynamic testing using either a black-box, grey-box, or white-box approach [17]; meanwhile, the (dynamic) scanner works from outside of the target (i.e., only uses the black-box approach) as per OWASP description [86] [92]. Therefore, scanners cannot identify the security vulnerabilities that cannot be uncovered by the black-box approach. Besides these limitations, the fuzzers incorporate more sophisticated strategies such as instrumentation than the scanners do, making the fuzzer can trigger more unexpected security vulnerabilities. This survey focuses more on fuzzing, which can be black-box, grey-box, or white-box, used for revealing web application vulnerabilities.

1.2.2 Related works

We identified several works that are similar to this study in the last ten years. Based on their focus, those works can be grouped into either general fuzzing or web API testing. First, some survey papers focused on the fuzzing approach in general, which is not explicitly intended for a specific platform, so their focus is different from ours. For example, the work of Chen et al. [17] and Li et al. [54] focused on exploring techniques for improving fuzzing in general. Another example is the work of Zhu et al. [103], which reviewed the knowl-

edge gaps of general fuzzing. Instead of reviewing fuzzing in general, our work explores more specific fuzzing approaches **tailored for web applications**, which utilises web resources that cannot be found in other application domains. Second, several works are done to review web API testing, such as the work of Martin-Lopez et al. [66], Zhang et al. [98], and Golmohammadi et al. [40]. Their works are quite similar to ours; however, their main review focuses are different because our survey focuses on revealing prior work strategies seen from the fuzzing approaches: **generating valid HTTP requests for attack surface exploration, utilising security feedback from the WUTs, and expanding relevant input spaces** (see Table 1).

1.2.3 Contribution

To conclude, our study's contributions are as follows.

1. We review existing fuzzing studies specially designed for **web application security testing through web API**. We investigate their strategies for **input generation, mutation, and feedback utilisation**.
2. We analyse security benchmarks used mostly for empirical evaluations.
3. We identify the remaining challenges that need to be solved.
4. We give some insights for future research directions.

2 Background

This section explains server-side web applications and the basic theory of fuzzing.

2.1 Server-side web application and web API

The server-side web application is the application running on the web server that executes any requests sent by the client [56] (see the illustration in the Figure 1). It works together with the web server to filter out broken or malicious request formats. We then only use the term "**web application**" to refer to server-side web applications for ease of reference.

The client sends requests to the web application through the **web API**, an interface enabling other users, like humans or programs, to access the web application functions through the computer network [46]. The most popular paradigm for accessing HTTP-based web API is Representational State Transfer (RESTful) [16]. RESTful is an architectural style in the web application to represent a standard interface that enables the client to interact with or manipulate specific resources [29]. RESTful APIs are supposed to be stateless; however, since they are connected to stored systems, such as databases or cache, they can be seen as stateful systems.

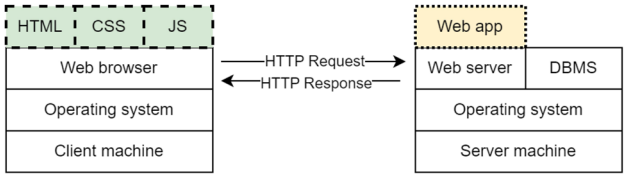

Fig. 1 Web application overview. The dashed boxes are called client-side web applications, while the dotted ones are called server-side web applications or only "web applications" for ease of reference

As explained below, researchers utilised several web API attributes to develop a web API fuzzing framework.

2.1.1 HTTP methods and response codes

Clients request the web API using various HTTP methods defined in the HTTP standard. Consequently, fuzzing researchers considered this rule to develop a test case generator producing valid input data. The four most commonly used methods related to web resource management are `GET`, `POST`, `PUT`, and `DELETE` [79]. After sending the request using one of the methods, the web server replies to the client with an HTTP response code. It is a three-digit code produced by the web application to help a client generally know what happened after processing the client request [79]. Most web fuzzers utilise this information to decide whether their mutated input triggered a particular web behaviour. The most common response codes are explained as follows.

- 2xx (OK): describing successful processing of the request and the web replies with intended resources.
- 4xx (Bad Request): describing the client's request may be incorrect or malformed.
- 5xx (Internal Server Error): describing a problem happening in the web application caused by the client's request.

2.1.2 Open-API specification

To help software developers comprehend web API usage, web API developers should provide standard documentation. Many tools and technologies related to API documentation are available, making the developer easy in preparing the documentation, such as RESTful API Modeling Language (RAML)[2], Swagger [3], API Blueprint[4], and others [22]. OpenAPI Specification, formerly known as Swagger, has been the official standard for documenting web RESTful API because of its mass adoption [34]. Therefore, most fuzzing

[2] http://raml.org/

[3] http://swagger.io/

[4] https://apiblueprint.org/

Table 1 Comparison of this survey paper to other similar reviews in web API testing

Paper	Main focus	Samples of the results
[66]	Black-box API testing (e.g., failure detection capability)	Experimental results on failure detection and fault detection capabilities in 13 online APIs under test
[40]	REST API testing approaches	Metrics for evaluating API testing effectiveness (e.g., coverage), testing techniques (e.g., black-box and white-box), kind of testing (e.g., system testing)
[98]	Empirical assessment and open technical problem analysis	Empirical comparison and technical analysis of seven state-of-the-art web API fuzzers
This paper	Adjusted fuzzing strategies for Web API	Diverse techniques for generating valid HTTP requests (Section 5.1 and 5.2), utilising feedback from the WUTs (Section 5.3), and expanding relevant input spaces (Section 5.4)

frameworks focused on applications with an OpenAPI specification to generate valid test cases. The example of the OpenAPI specification with some essential components in version 3.1.0 written in YAML format is listed in Appendix A.

The crucial points most concerned by the web API fuzzer are the endpoint, method, and parameter. Endpoint (line 5 of Appendix A) is a path string pointing to a specific URL of the API that can be followed by a token and/or an argument [59]. Method (line 6) refers to the HTTP methods like GET or POST, and the Parameter (lines 7-15) is data formed key-value pairs sent to the API server. One endpoint can be called using different methods, so the API server treats each endpoint-method combination differently.

2.2 Fuzzing

Fuzzing, standing for fuzz testing, is an automatic software testing intended to find vulnerabilities or bugs, first proposed by Miller *et al.* [72] in 1988. The main ideas of this testing are automatically producing a huge amount of input data, injecting them into the software under test (SUT), and then watching the software's behaviour, whether it results in a crash, fault, or hang [57]. Algorithm 1 illustrates the fuzzer process using the **mutation-based approach** (explained in Section 2.2.1). First, a fuzzer calls a mutation method to pick a random seed from the corpus and to produce an input (line 5). Then, the fuzzer injects the input into SUT and gets feedback (line 6). Lastly, the fuzzer stores the input if the feedback contains an error or is interesting (lines 7-13). The original fuzzing approach proposed in 1988 has become extremely competent with recent developments. Instead of producing genuinely random inputs, additional techniques were proposed to make it more intelligent in finding error-triggering inputs, such as program instrumentation [54]. Based on the available knowledge about the SUT (e.g., source code), fuzzing can be classified into three categories: Black-box, Grey-box, and White-box [17]. While the fuzzing model running without having access to the SUT's source code was grouped into the black-box fuzzing, the rest that have varying levels of information are classified as either grey-box or white-box fuzzing. If the fuzzing methods only instrument the code for obtaining coverage information at runtime, they are considered the grey box. Otherwise, they are white-box.

Algorithm 1 Mutation-based fuzzer process, extracted from [4]

Require: s, I ▷ Set the software under test and samples of the input
1: $O \leftarrow \emptyset$ ▷ Initialize Output set
2: $B \leftarrow \emptyset$ ▷ Initialize Bug set
3: $C \leftarrow I$ ▷ Corpus set is filled with input sample
4: **while** $true$ **do**
5: $i \leftarrow Mutate(C)$ ▷ Get an input from mutation process
6: $feedback \leftarrow Inject(i, s)$ ▷ Get the feedback from the execution
7: **if** $feedback$ contains error **then**
8: $C \leftarrow C \cup i$ ▷ Save the input into Corpus
9: $O \leftarrow O \cup i$ ▷ Save the input into Output
10: $B \leftarrow B \cup o$ ▷ Save the output into Bug
11: **else if** $feedback$ is interesting **then** ▷ e.g., feedback score is high
12: $C \leftarrow C \cup i$ ▷ Save the input into Corpus
13: **end if**
14: **end while**
15: **return** B, O

The following sections will explain the common fuzzing approaches used in web API testing.

2.2.1 Mutation-based input generation

Mutation-based fuzzing is the most common fuzzing technique that takes valid input data and then creates new inputs by making small, random changes (mutations) to the input [63]. Several common mutations can be applied to the input data, such as flipping some bits, inserting or deleting characters, and changing the order of existing bytes. This mutation-based input is straightforward because it does not

require any knowledge of the software under test and can be used in a black-box testing scenario. However, since the quality of the initial inputs significantly influences the fuzzing performance, having a diverse set of well-formed seed inputs is crucial to begin the fuzzing process [37]. That kind of input can help fuzzer to explore more execution paths quickly. In the context of web API testing, fuzzing is adjusted to adopt more mutation strategies. The mutation can change the HTTP request order in a sequence, HTTP request structure, or HTTP parameter values. These techniques will be explained in more detail in Section 5.4.

2.2.2 Grammar-based input generation

Mutation-based fuzzing demands at least one initial valid input from the user and mutates it to produce plenty of input. However, recent development shows that a fuzzer can employ another way to work as the input generator. It is grammar-based fuzzing, an approach to generate valid input data by employing a grammar that specifies the structure and constraints of the input data [63]. The work of Pezze and Young [77] called this approach specification-based testing because it generates test cases based on the test specification (i.e., grammar). In certain applications, grammar creation is greatly assisted by the supporting documents that are an integral part of certain domains. For example, in the web API context, a fuzzer can employ a grammar model deduced from the OpenAPI Specification or HTML documents (explained in Section 5.1) in order to produce valid API requests. The grammar created in this context consists of request type, server endpoint, and required parameter names, which are set up as static, and required parameter values that are *fuzzable* or replaceable (see Appendix B). So, when calling this grammar to build one valid HTTP request, the fuzzer copies the static data and changes the fuzzable data to concrete values that can be obtained from various sources, such as a dictionary [85]. In addition, the fuzzable data can also be created by applying the mutation-based strategies—referred to as **grammar-aware mutation** [88]—to make the generated inputs more varied and valid. In a nutshell, implementing grammar-based input generation helps the fuzzer to produce HTTP requests that strictly match the rules or specifications.

3 Survey methodology

This section presents the methodology used in collecting and filtering the paper in the literature search.

3.1 Searching papers

We review papers issued in seven research article repositories: ACM Digital Library, IEEE Explore, Science Direct, Wiley, Web of Science, MIT Libraries, and Springer because those are well-known publication venues storing top articles on computer science topics. In addition, most similar survey papers we mentioned in Section 1.2 used those for the article sources. Since our scope is existing fuzzers specially designed for web application security testing through web API, we used keywords, namely `fuzzing`, `web`, `REST`, `API`, `security`, `vulnerability`, and `testing`, and formulated them into different query syntax (see Table 2) to search relevant papers specifically on the repositories. We limited our search to the last ten years (2013–2023) to obtain the most recent studies. We applied additional filters in the repositories to show only research articles. We also paid attention to double-indexed articles (e.g., indexed in ACM and IEEE) to count them as one.

3.2 Manual filtering of the collected papers

Papers whose main ideas do not propose or improve a fuzzer must be dropped. To remove such papers from the paper collection, we scanned the paper abstract, experiment, and result sections. We paid attention to those sections because papers focusing on developing web API fuzzers generally tested their frameworks for any web application and showed their experimental results. If there is no experimental result, we consider the paper not to propose a new improvement method and then drop it from the collection. Specifically, papers were excluded because of one of the following reasons:

1. They were survey or review papers.
2. They only conducted empirical studies to compare other studies without proposing a new framework.
3. Instead of a web application, they intended to test the browser, applications, or other engines.
4. They intended to test specific application APIs (e.g., machine learning APIs or interpreter APIs).
5. They do not do server-side web application testing but rather client-side testing or UI testing.
6. They mentioned web API testing and fuzzing, but their main topic is not web API testing techniques (e.g., they conduct short-campaign testing for analysis purposes).

3.3 Collection expansion

After performing the methodology explained in prior sections, we ended up with 34 articles (see Table 2). To reduce the chance of missing relevant works, we further scanned the related work sections of these papers. In addition to broadening our search, we checked other studies that had cited the collected papers and reviewed them to determine whether they were related to web API fuzzing. If affirmative, we include them in our paper collection.

Table 2 Number of papers collected from some repositories after manual reduction (sections 3.2).

No	Repository	Query	Additional filter	Selected papers
1	ACM Digital Library	AllField:(fuzzing) AND AllField:(web) AND AllField:(REST) AND AllField:(API) AND AllField:(security) AND AllField:(vulnerability) AND AllField:(testing)	Only "Research Article"	13
2	IEEE Explore	("Full Text & Metadata":fuzzing) AND ("Full Text & Metadata":web) AND ("Full Text & Metadata":rest) AND ("Full Text & Metadata":api) AND ("Full Text & Metadata":security) AND ("Full Text & Metadata":vulnerability) AND ("Full Text & Metadata":testing)	Only "Conferences" and "Journals" & Publication time: 2013-2023	15
3	Science Direct	Terms: fuzzing and web and rest and api and security and vulnerability and testing	Only "Research Articles" & Publication time: 2013-2023	2
4	Wiley	Anywhere: fuzzing and web and rest and api and security and vulnerability and testing	Only "journals" & Publication time: 2013-2023	1
5	Web of Science	((((((ALL=(fuzzing)) AND ALL=(web)) AND ALL=(rest)) AND ALL=(api)) AND ALL=(security)) AND ALL=(vulnerability)) AND ALL=(testing))	Only "journals" & Publication time: 2013-2023	1
6	MIT Libraries	Fuzzing and web and rest and api and security and vulnerability and testing	Only "journal articles" & Publication time: 2013-2023	0
7	Springer	Fuzzing and web and rest and api and security and vulnerability and testing	Only "conference paper" or "article" & Publication time: 2013-2023	2
Total				34

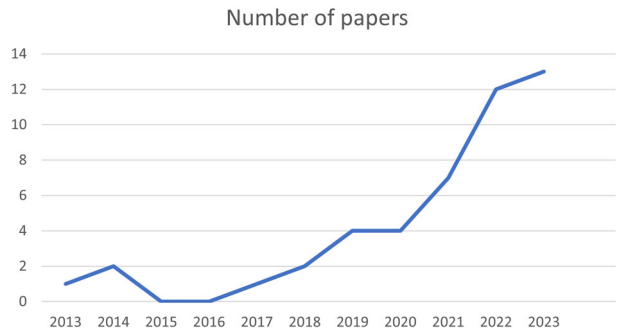

Fig. 2 Number of papers in our collection published over the last ten years

3.4 Summary on publications

Finally, we obtained 47 articles, including 34 initially found by search and 13 new ones by performing the collection expansion step. The majority of them (64%) are conference papers published in proceedings, while the rest are journal articles (34%) and book chapters (2%). Designing a web API fuzzing framework is a trending topic because more and more papers are published over time (see Figure 2). Evidently, there were only 3 papers in total in the initial four-year period, but after that, the number of publications increased significantly.

4 Web API fuzzing overview

Web API fuzzing consists of two phrases: web API and fuzzing, which the phrases have been explained in Sections 2.1 and 2.2, respectively. Based on the descriptions in those sections, we conclude that web API fuzzing is an automatic test through typical web interfaces with a large amount of HTTP requests for finding web vulnerabilities. Overall, web API fuzzing has distinctive characteristics of its program under tests (PUTs) and uses either crash-driven or vulnerability-driven methodology.

4.1 Program under test

Web API fuzzer aims at any web application (either interpreted or compiled) which requires input in the form of an HTTP request. Even though web application is related to the network protocol, web API fuzzer has several clear differences from network protocol fuzzer because the characteristics of the PUT differ.

4.1.1 Web API fuzzing vs network protocol fuzzing

Network protocol fuzzing targets applications implementing certain network protocols (e.g., FTP) whereas web API fuzzing aims at applications running on top of the network

protocols. Two popular examples of the network protocol fuzzers are AFLNet [78] and ChatAFL [71]. Since the protocol applications are stateful, they enforce strict sequencing that only processes correct inputs in the right order [71]. On the other hand, because the web applications operate over the protocol application, they require more: valid HTTP requests to satisfy the web protocol, with certain parameter values and body payload to satisfy web application logic [82]. Therefore, the web API fuzzer has a more extensive scope than network protocol fuzzing because it explores more vulnerable codes while keeping the generated requests valid and complying with the logic.

4.1.2 Web API fuzzing vs binary fuzzing

Due to the diverse execution environment, binary fuzzers—actively developed by the security community—are typically optimised for basic application interfaces, such as command-line [103], which request one set of inputs. On the other hand, web applications present a more complex attack surface, characterized by many API endpoints with different input structures for each endpoint. It makes the web API fuzzers propose sophisticated strategies for prioritizing and sequencing endpoint execution, especially when considering API dependency (explained in Section 5.1.1). In addition, web API fuzzers have to deal with more complex interfaces (i.e., dynamic web pages) or OpenAPI specifications to effectively retrieve available endpoints and validate input constraints. In a nutshell, web API fuzzers incorporate more specialised components than those in the binary fuzzers.

4.2 Methodology

Based on the approach to finding vulnerabilities, existing works on web API fuzzing are classified into crash-driven and vulnerability-driven fuzzing. While the former is intended to test all WUT codes to find web crashes, the latter is supposed to test certain code regions with pre-defined rules to examine specific vulnerabilities the user is looking for.

4.2.1 Crash-driven fuzzing

The crash-driven fuzzer focuses on producing plenty of test cases leading to the WUT crashes, and then, based on the logs of raised errors, the tester analyses the vulnerabilities, errors, or bugs behind the crash. Its main objective is to reach and test all statements and possible states in the web application. For those reasons, this fuzzing has a high chance of finding the 0-day vulnerabilities. Some examples of existing works that design crash-driven web API fuzzers are Restler [12] and MINER [62].

4.2.2 Vulnerability-driven fuzzing

Unlike the first group, which focused on testing the entire code, this group may only look for certain code regions containing desired vulnerabilities. Because its work is designed to focus on specific vulnerabilities, this group is called vulnerability-driven fuzzing. The web API fuzzing frameworks that belong to the vulnerability-driven fuzzing are already supplied with custom mutators and vulnerability checkers to achieve their targets. The fuzzing framework uses particular mutators to produce input satisfying the requirement for triggering the vulnerabilities in the code. Then, it checks for the symptoms of vulnerability using the custom checkers. Some examples of existing web API fuzzers working in this category are Cefuzz [102] and Zokfuzz [95].

5 Web API fuzzer workflow

To identify security vulnerabilities and bugs in web applications, web API fuzzers produce valid HTTP requests first, then systematically craft the requests with varying parameters, headers, payloads, and structures, and monitor the web responses throughout the process. Since most of the fuzzers we reviewed deal with web services which provide OpenAPI specifications, the fuzzers are supplied with modules to analyze the structure of web API to obtain valid request formats. Therefore, in general, web API fuzzers involve four crucial processes: producing HTTP request templates, rendering them to produce concrete HTTP requests, executing them and getting feedback from WUT (Web Under Test), and mutating the requests to trigger vulnerabilities (illustrated in Figure 3). We then classify the problems and solutions developed by existing studies into these processes and finally result in the problem-solution taxonomy (see Figure 4).

5.1 Request template generation

RQ1: To explore attack surface, what techniques are used to generate HTTP request templates?

To effectively explore the attack surface of WUTs, web API fuzzers must produce valid HTTP requests with maximum code execution coverage. Achieving this requires both crash-driven and vulnerability-driven web API fuzzers to generate valid request templates first.

HTTP request template generation is the process of analysing given information to produce diverse request templates, which is essential in assembling long request sequences to reach deeper WUT statements and explore more security-related bugs. This process can also be called **grammar generation** because it constructs the grammar sets that define the structure and format of the input data. Since

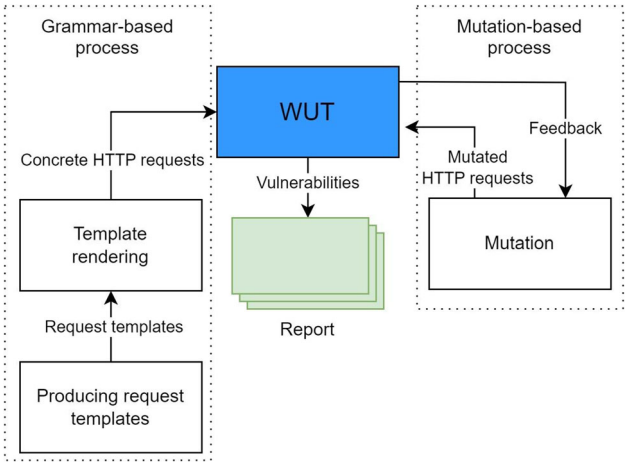

Fig. 3 General overview of web API fuzzer. The first two steps in the left (explained in Section 5.1 and 5.2) are classified as grammar-based processes, while the next steps in the right (explained in Section 5.3 and 5.4) are mutation-based process

creating templates is challenging considering the limited information that can be used as a reference, several studies have identified certain problems related to this issue and proposed some solutions as follows.

5.1.1 Problem: request dependency

Most existing works raise a request dependency problem because it is unclear how to create correctly sequenced HTTP requests. Basically, the frameworks use the Open API specification as the guidance to produce valid HTTP requests. However, the document does not provide clear information regarding the order in which the HTTP requests are made –for example, which requests must be called first and which can be called later. When the request sequence is chaotic, the web server likely cannot process most requests because certain web states or conditions have not been met to execute the commands on those requests, which ultimately drops the fuzzer probability of uncovering web vulnerabilities. For

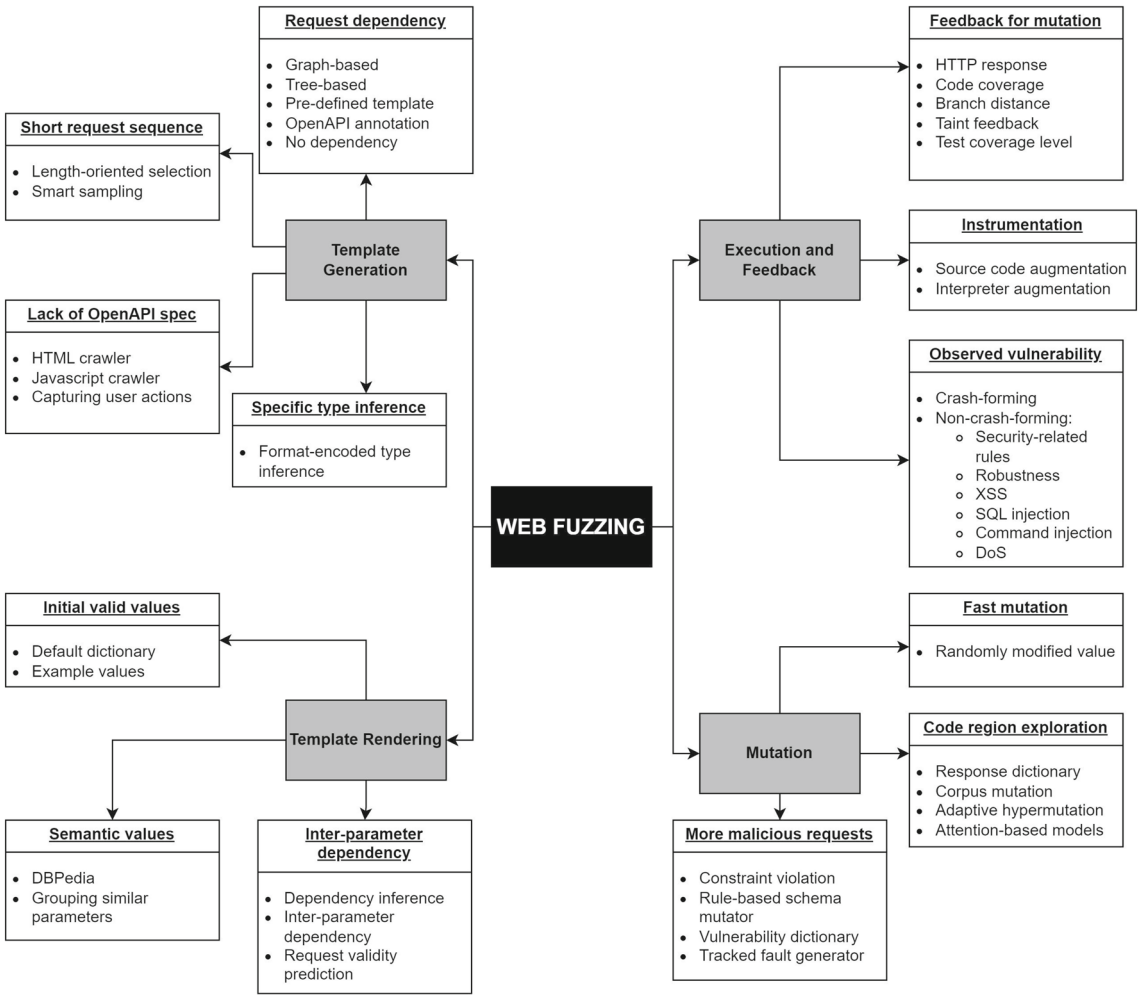

Fig. 4 Taxonomy of the problems and solutions from prior web API fuzzing studies described in Section 5. The grey boxes are the web API fuzzer processes explained from Sections 5.1 until 5.4. Each process has different problems along with their respective solutions (drawn in white boxes)

Fig. 5 An example of the dependency graph model. The *get category* API will produce a cat ID that will be needed for calling the *post book* API. The latter will generate a book ID used by the *delete book* API

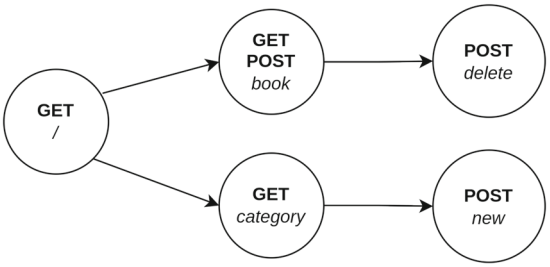

Fig. 6 An example of the tree model proposed by Lin et al. [59]. The tree describes valid endpoints that can be accessed: the web API root (/), /book, /book/delete, /category, and /category/new with some HTTP methods

instance, the web server cannot execute a delete request if the resource to be deleted does not exist. Therefore, some studies developed diverse ways to solve this problem.

Graph-based dependency Some researchers have developed a dependency graph to construct the relationship among HTTP requests. This graph provides clear guidance to find which requests must be committed before calling a specific request. The researchers proposed various techniques to build this graph, especially in deducing the request dependency. Atlidakis et al. [12] infer the request dependencies by using the request types declared in the specification. RESTler, the framework they built, analyses the specification to determine which resources in one response become requirements in another request (see Figure 5). For example, calling a *new item* request will produce an *item id* in the response, which this *id* field has to be inserted when calling an *update item* request. Then, it can be concluded that the *update item* request requires calling the *new item* request first because of the *id* field. Corradini et al. [20], other researchers who adopted the RESTler method, explained that identifying those dependency fields might be tricky because the field names in different operations are sometimes written differently, yet they are the same semantically. Therefore, those researchers use several strategies to match the field names, namely case sensitivity, ID completion, and stemming. Moreover, they utilise the CRUD semantics to determine the request order. For instance, the POST requests have to be called first before the GET or DELETE requests. Another researcher, Yamamoto [90], designed a bipartite graph to ease the request dependency inference. The graph describes the relation between APIs and named values that appear in both request and response parameters.

Tree-based dependency Lin et al. [59] found the classic dependency graph is inefficient because of the dependency explosion among APIs. This explosion led to too many possible paths that will overwhelm the fuzzing when traversing a complex graph model to explore APIs. As a result, they developed a tree-based dependency that is much simpler because the fuzzing frameworks only need to traverse a tree (which is linear complexity) instead of traversing a graph (which is quadratic complexity). This dependency is inspired by the fact that web users generally execute the parent node before calling its child nodes. A valid URL comprises several components separated by the slash (/) character. Nodes represent these components, and an edge between two nodes appears if a valid URL contains those components. Illustrated in Figure 6, the tree model depicts the relationship between a root URL and its child URLs in which a node can contain one or more HTTP methods. Traversing this model via a depth-first or bread-first order can be more straightforward. In addition, Wu et al. [89] use such hierarchical relations of resources that can be depicted in a tree structure to generate request sequences. Specifically, they proposed a constraint handler based on the hierarchical relations of resources and CRUD semantics to construct the template dynamically. The handler checks if each request in a template does not access a resource before its creation or after its deletion. For instance, the handler rejects the construction of a sequence consisting of GET /book/id and POST /book because it accesses the bookID before being created by the POST method.

Pre-defined template The pre-defined template is a list or table containing the HTTP request order of an API. Tsai et al. [84] prefer to use manual input combined with depth-first search to solve the API dependency and create the template. On the other hand, Zhang et al. [100] proposed two templates: an independent template as an action that does not affect follow-up actions on any resource and a non-independent template as an action whose effects cannot be predetermined (see Table 3). Then, they also designed a smart sampling method (explained in Section 5.1.2) to construct a request sequence based on both templates.

OpenAPI annotation Instead of generating graph or tree-based dependency, a fuzzing framework can put the information, i.e., annotate the OpenAPI specification, to add more comprehensive information. Deng et al. [24] use this idea to propose a set of customised annotations for OpenAPI Specification to help fuzzing generate desired request sequences. They introduced several types of annotations. For example, `dep-operation` expresses the dependent operations that must be performed before a certain operation. Since the annotations are human-readable and automatically processable, they can be performed by both human experts and fuzzing modules. Fuzzing modules infer the dependency based on the

Table 3 Some examples of the template model proposed by Zhang et al. [100]

	Description	Independent?	Template
1	To retrieve a resource	YES	GET
2	To create a resource	NO	POST
3	To create an existing resource	NO	POST-POST

response fields of a certain operation that are used in other requests.

No dependency Some web API fuzzing frameworks do not describe any specific model to infer valid consecutive API requests because the frameworks only construct one HTTP request for each test case. Some examples of existing works implementing this approach are the works of Van Rooij et al. [80], and Trickel et al. [82]. They did it because they mainly focused on enabling binary fuzzing tools or methods in web applications where the original tools in the binary applications do not require input sequences.

5.1.2 Problem: short request sequence

Another problem mentioned by existing studies is the short request sequences. Since existing web API fuzzers tend to produce such sequences, Lyu et al. [62] stated that a long request sequence is essential to reach deeper states and vulnerable codes in web applications because it can cover many possible request combinations. One of these combinations may be valid for exploring vulnerable codes in hard-to-reach states. The authors also conducted a preliminary experiment showing more than 75% of the issues in the GitLab services -the WUT- can only be reached using a long request sequence. Specifically, the authors took at least 3 HTTP requests in a sequence to reproduce those issues. Therefore, producing long request sequences can enable better testing coverage. Several methods can be used to produce longer request sequences, summarised below.

Length-oriented selection Lyu et al. [62] developed the length-oriented sequence construction module to construct candidate sequence templates. Initially, they employ the graph-based dependency as in the RESTler [12] to produce initial sequence templates. Then, because the module uses a custom probability function, longer templates will have a higher chance of being chosen for the next stage: the extension process. This process puts a new request at the end of each chosen template. The newly generated templates will be retained if they bring valid responses. Therefore, each fuzzing iteration will produce longer and longer request sequences.

Smart sampling Arcuri [7] proposed a smart sampling technique and pre-defined templates (explained in the prior section, see Table 3) to construct an individual test case containing some consecutive HTTP requests. The sampling involves four methods that can be chosen, namely by sampling: a resource with an independent template, a resource with a non-independent template, two resources, and more than two resources in which the last two only allow the last resource with non-parameter GET. Using this sampling technique will help fuzzing produce longer and longer sequences because other requests will be added before the existing request. The added requests placed in front are expected to put the WUT into the correct state for executing the existing request.

5.1.3 Problem: lack of OpenAPI specification

Most web API fuzzing frameworks use OpenAPI specification documents to retrieve API formats in the web application. Many web applications do not have OpenAPI specifications, but the web fuzzers can rope with this by crawling and parsing the web client pages or employing a human.

HTML crawler Duchene et al. [28] proposed a state-aware crawler to parse the HTML documents and learn the control flow of the web application. The crawling process results in a control flow model (CFM) representing web pages (node) and requests (transition), and then the results are used by the KameleonFuzz [27], the authors' fuzzing framework. Van Rooij et al. [80] developed a similar HTML crawler to scan links from *anchor* and *form* elements in each HTML document reached using the *html5lib* library. This similar crawler workflow was also used by Witcher, a web fuzzer developed by Trickel et al. [82]. Given an entry point URL by the user, the Witcher's crawler works by scanning the HTML document responded to by the web server to find links and relevant fields that can establish HTTP requests, like form, input, select, and textarea elements.

Javascript crawler Javascript documents can be an alternative to crawl web URLs compared to HTML-formed responses since modern web applications often put their URLs to the web server in JavaScript documents. As an alternative to parsing the HTML documents, Gauthier et al. [33] developed BackREST using a client-side javascript document to obtain a REST API inference model. They demonstrated one example: the web API entry points in a javascript document used in the Node.js Express application

are similar to those in the Open-API Specification. Since these documents reveal similar information, BackREST introduced a state-aware crawler to analyse the javascript code that calls web APIs. The crawling process yields valid HTTP requests, and the requests executed by the browser are intercepted by the Man-In-The-Middle (MITM) proxy to build an API inference model. This similar approach was also adopted by Yandrapally et al. [91], who designed an API test carving to produce an API-level test suite and a test execution report and OpenAPI specification. Their work is intended for all web applications, irrespective of the web frameworks they use.

Capturing user actions Instead of employing crawlers to obtain valid APIs, Fung et al. [31] utilise a human to interact with the WUT in the browser as usual. They built a browser add-on to capture the user actions, and the add-on can track the parameter dependency from the submission requests and the server responses. Finally, the capturing phase will produce valid request templates that can be mutated later.

5.1.4 Problem: specific type inference

The previous works on web API fuzzing have not addressed the root cause of request candidate space explosion, which is the limited data types available to represent API parameters. For example, a study by Lei et al. [53] showed that most web APIs use string-typed parameters, and the string type has a huge space to be explored. Generating effective string values that can pass parameter validation, reach business logic, and trigger security-related bugs is challenging because there is little information to narrow down the input space of the parameter. Therefore, to address this issue, recent work focused on inferring the information about the data types, as discussed below.

Format-encoded Type (FET) inference Lei et al. [53] proposed a FET inference technique to provide a fine-grained description method for parameters that utilises data type and value format. FET can be defined as new data types that are more specific than the conventional types the software developers are using in programming (e.g., string or integer). To construct the FET lattice, the authors referred to popular RESTful services (a total of 1268 APIs). They resulted in 21 ubiquitous FETs organised in 5 levels. All API parameter values are classified into those FETs to enable the application of different mutation strategies. For example, the FET of datetime (value example: ”2019-2-29”) has a different treatment from the FET of hash (value example: ”19CGHEE2”) even though both are string-typed. Fuzzing will apply a strategy to make year overflows in the datetime parameter and to use a non-hexadecimal-number string in the hash parameter.

Answer to RQ1: To generate HTTP request templates for attack surface exploration, the most prominent solutions are: 1) creating request dependency (either graph-based or tree-based) inferred from OpenAPI specification, and 2) using pre-defined templates. The others are OpenAPI annotation, length-oriented selection, smart sampling, HTML/JS crawlers, capturing user actions, and format-encoded type inference.

5.2 Template rendering

RQ2: How are the request templates rendered?

After generating request template sequences from the prior step, existing web API fuzzers commonly continue to concretise the sequences to make the valid sequences reach deep vulnerable codes in WUTs. This process is known as **template rendering**, which takes generic input templates and populates them with concrete values to create real requests that can be sent to the WUT (see Figure 7). These two initial processes (request template generation and template rendering) can also be called **grammar-based input generation** because the input templates define the structure and format of the input data, and fuzzer will fill in the placeholders in the templates. The processes allow the fuzzing framework to generate valid inputs that conform to the expected format of the WUT. Several problems arise in this stage, as follows.

5.2.1 Problem: initial valid values

Fuzzer will miss opportunities to test deeper web functionalities and reach vulnerable statements if it produces initial HTTP requests that do not comply with the WUT’s requirements. The WUT will reject those malformed requests, and then the mutation process struggles to modify them to be valid. In the end, no matter how long the fuzzing process operates, it is very likely that fuzzing will fail to explore interesting features in the WUT. Therefore, initial valid values serve as a good starting point for the fuzzing campaign. However, some studies mentioned the complexity of finding valid values because the OpenAPI specification rarely provides valid examples of each API field. This situation made the researchers design a default dictionary in the template rendering process.

Default dictionary and example values (DD) Basically, all existing web API fuzzing frameworks prepare default values for each data type to concretise the request templates. Atlidakis et al. [12] use a user-configurable value dictionary to fill the request templates based on the field types. For example, they set values 0, 1, and -10 for fields requesting integer values and "sampleString" for fields requesting string values. Other examples are Godefroid et al. [38] and Lyu et al. [62],

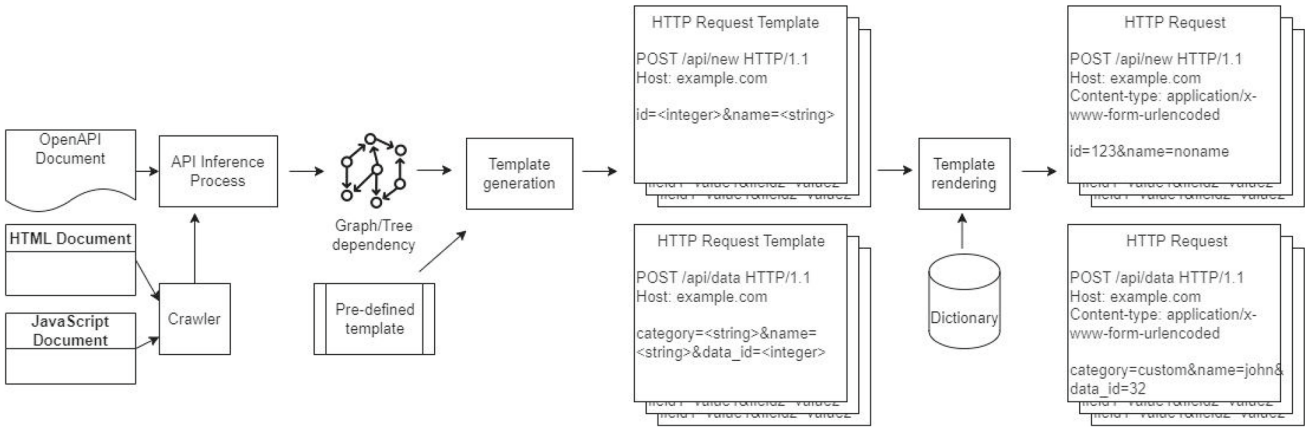

Fig. 7 Illustration of generating concrete request sequences described in sections 5.1 and 5.2

which defined various valid value options for each data type. Besides using this technique, other researchers also found that some OpenAPI specifications may provide the *example* fields that hold concrete example values documented by the web developers. Therefore, apart from using their default values, some researchers, such as Corradini et al. [20] and Deng et al. [24], utilise the *example* field to fill the initial values of HTTP request templates.

5.2.2 Problem: semantic value

Some studies highlight the issue of semantic value, where fuzzers should generate inputs that are not only syntactically correct but also semantically meaningful in the beginning. This challenge arises because human testers usually provide inputs that are both valid and contextually appropriate for each API parameter [2], thereby increasing the likelihood of reaching unsafe codes within the application. In real-world API applications, providing only syntactically valid but semantically invalid inputs will bring unsatisfactory results. For example, the search operation in the YouTube API application demands the user to fill in semantically valid values for every search field. If not fulfilled, the application will not show a result because it does not match with any data. This means the user cannot thoroughly test the search feature without the semantically valid values. In addition, producing semantically valid inputs is challenging because no semantic information can be derived from the OpenAPI specification. Therefore, some studies proposed a strategy to address this issue as follows.

DBPedia Previous works used external knowledge dictionaries to help fuzzing. Alonso et al. [1] [2] leverage semantic knowledge discovery to generate realistic test inputs. They used DBPedia [14], a knowledge base, because they had identified its potential, but no one had implemented it in the web testing domain. However, employing this dictionary is not straightforward because of its diverse domains. For example, when fuzzing demands the concrete value for a *title* field, the DBPedia will return hundreds or even thousands of diverse title-type inputs (e.g., movies, games, or books) [2]. The authors managed to narrow down the results by inserting more specific criteria derived from HTTP parameter combinations. For instance, adding the *publisher-name* field in the search queries with makes the results more specific to book title values.

Grouping similar parameters Liu et al. [60] group the string parameters with the same semantic meaning to generate semantically correct results. For example, *loginid*, *username*, and *account* have a similar semantic meaning that demands the user to provide a name. Therefore, the grouping module will utilise certain modules to produce correct names. Another example is the group of the description, message, and comment field that requires the user to give a description text.

5.2.3 Problem: inter-parameter dependency

Inter-parameter dependency means that one or several parameter values depend on other parameters. For example, the Youtube API documentation states that the *type* parameter must be filled with *video* before setting up the *videoDefinition* value [65]. Therefore, if the *type* parameter is not filled in, the web server will not process the value of the *videoDefinition* parameter, making critical paths within the application remain unexplored. Another example is that a parameter representing the marital *status* of an individual must have the value of *married* before filling in the *spouse-name* field. Martin-Lopez et al. [65] raised this issue because OpenAPI specification does not inform this dependency explicitly. Their study revealed that around 85% of APIs from 40 real-world applications (containing more than 2.5K APIs) have such parameter dependency rule. Without satisfying the rule, a fuzzing campaign cannot go further to test the corre-

sponding functions. To overcome this problem, some studies proposed solutions as follows.

Dependency inference Wu et al. [89] employ a Natural Language Processing (NLP) method to infer the parameter dependency constraints that are usually written informally in the *description* field of the Open-API Specification documents. They leverage spaCy[5], an open-source library, to perform this task. Before a fuzzing campaign begins, they prepare some patterns to look for, such as *if PARAM_A is VALUE_B, PARAM_C is required*. Then, using the pattern-based matching engine, the library will check if any text in the Open-API Specification documents matches those patterns. Adopting a similar idea, Kim et al. [47] designed an NLP-based rule extraction to infer the rule and inter-parameter dependencies from the description fields. The extraction process involves vocabulary term identification, value and parameter name detection, and rule generation. Finally, the process produces OpenAPI-compliant rules that are machine-readable.

Inter-parameter Dependency Language (IDL) Martin-Lopez et al. [70] presented a domain-specific language called IDL (Inter-parameter Dependency Language) to express seven types of inter-parameter dependencies: *Requires, Or, OnlyOne, AllOrNone, ZeroOrOne, Arithmetic/Relational*, and *Complex*. In this language, those dependencies are expressed using invariants, conditional definitions, logical operators, and relational operators. However, this study still uses manual work to create the corresponding IDL from the API documentation. Apart from IDL, the study also proposed an automated analysis tool to check whether a request meets all the dependency constraints. Ultimately, the tool and IDL can automatically generate concrete requests satisfying the inter-parameter dependencies [67].

Request validity prediction Mirabella et al. [73] employ a deep learning model to predict if a certain HTTP request is valid that satisfies all the input constraints, including the inter-parameter dependencies. The training data is a dataset of API calls with valid or invalid labels. The learning model consists of the input layer that fits the data frame numbers, five inner layers (32, 16, 8, 4, and 2 neurons, respectively), and the output layer (one neuron). If the output layer produces a value greater than 0.5, the input is valid; otherwise, it is faulty.

Answer to RQ2: To render the request templates, the most prominent solutions are using default dictionary and example values. The others are initial valid values, semantic values, inter-parameter dependency, DBPedia, grouping similar parameters, dependency inference method, inter-parameter dependency language, and request validity prediction.

[5] https://spacy.io/

5.3 Execution and getting feedback

Fuzzer sends all HTTP requests produced by the prior steps to the WUT and then receives the reply. Based on the reply, the fuzzer will decide which requests or sequences must be explored and what mutation strategies must be applied to reach problematic codes and trigger any vulnerabilities in such codes. In this stage, the fuzzing researchers mention several problems related to reply messages, as follows.

5.3.1 Problem: feedback for mutation

RQ3: What types of security-relevant feedback are collected from the WUT?

Feedback from the WUT is crucial for the mutation process because it helps comprehend the impact of the generated requests. If supplied with helpful feedback, fuzzer performance will be better because it can neglect the ineffective requests to exploit the effective ones to trigger security vulnerabilities. In addition, particular feedback helps fuzzer to identify areas of the code that have not been tested and generate inputs specifically to explore those regions. This can increase the likelihood of finding hidden vulnerabilities. Prior web API fuzzing studies observed the following feedback information.

HTTP response HTTP response is the standard information a web application provides after executing an HTTP request. Since it is common feedback, all web API fuzzers utilise this information to deduce whether specific vulnerabilities have been successfully triggered. In particular, all crash-driven fuzzers focus on the 500-status code, which indicates a server crash or failure, as a main signal of application instability following malicious inputs. In addition, a 200-status code, which represents a successful response, can guide the fuzzer deeper into the execution path of WUTs for further exploration of potential vulnerabilities in more complex or hidden functionality. For example, a fuzzing framework built by Lyu et al. [62], MINER, re-uses request sequences that bring valid HTTP response codes to produce longer sequences. Similarly, Wu et al. [89] developed a fuzzer called RestCT that obtains successful requests to utilise several constrained covering arrays for generating next-round requests. Then, a fuzzing framework called ResTestGen built by Corradini et al. [19] collects requests with successful status codes to be used to create new test cases.

Code coverage Code coverage is how much code in a WUT is executed when it receives inputs. This feedback is crucial for fuzzers to monitor which portions of the code may remain unexecuted due to the presence of complex branching conditions. Maximizing code coverage is essential because it increases the likelihood of exposing vulnerabilities by reaching and executing all faulty codes. Given code coverage is

not standard feedback in web applications, it can only be enabled by implementing instrumentation, which will be provided in Section 5.3.2. Prior works employed code coverage to enhance web API fuzzing frameworks. Van Rooij et al. [80] use coverage score and other metrics to rank requests. The AFL algorithm [52] highly inspired their work to keep a new high-score request for future mutation. The algorithm also imbues Trickel et al. [82] to develop a web API fuzzer called Witcher. This work augmented the web interpreter to produce code coverage so that the Witcher could store and mutate the request that brought a new execution path.

Branch distance Some vulnerable codes may exist behind a hard-to-solve branch condition, making a fuzzer hard to generate an HTTP request to satisfy the condition. To get more precise information on which conditions need to be solved to go deeper, Arcuri [5] calculates a branch distance, which is how close a particular input is to solving the constraints [49]. For example, the conditional branch of *if (a==20)* is reached by an HTTP request with parameter *a=15*. Therefore, the branch distance of the request is 5. To be able to calculate the distance, custom instrumentation is needed. In some cases, satisfying the conditional branches may be complicated because the involved variables are not user-controlled data that are not required in the HTTP request. Instead, those are influenced by other functions. Therefore, the fuzzing framework proposed by Arcuri [5] tends to pick requests with low branch distances for the mutation stage because those are relatively close to solving the branch.

Taint feedback Taint analysis and tracking are important techniques in fuzzing because they provide information about how data flows through a program, highlighting the influence of external input on the program's state [83]. Taint feedback, the result from the taint analysis, helps to understand the input data propagation through the program. Marking certain input data as tainted can trace the input flow and identify potential security-sensitive operations that involve user-controlled data. Using this approach, the fuzzing framework can focus on generating inputs specifically targeting the paths and operations influenced by HTTP requests to increase the chances of discovering vulnerabilities. Gauthier et al. [33] proposed a taint-driven fuzzing containing taint analysis that is executed in the initial stage before starting the fuzzing campaign. During the campaign, the analysis will result in taint feedback on which requests reach security-sensitive program locations that are already defined. Arcuri et al. [9] use the taint analysis to track variables at runtime for seeding strategies.

Test Coverage Level (TCL) Developed by Martin-Lopes et al. [69], TCL assesses the coverage of the request collections. It roughly measures the extent to which the WUT's code has been exercised by the generated test inputs using the black-box approach. It uses input and output criteria (e.g., paths, operations, and content type) that can be observed without having the source code. TCL0 represents the weakest coverage level, and the strongest one is TCL7. Each TCL level has different criteria that the requests must satisfy. To reach a certain TCL, the request collection must meet all requirements belonging to the previous levels. Tsai et al. [84] developed a fuzzing framework called *HsuanFuzz* employing this TCL concept. If request collection can increase the TCL of a certain API path, it will be stored in the fuzzing corpus.

Answer to RQ3: The security-relevant feedback that is most often extracted from the WUT is the HTTP response code. The others are code coverage, branch distance, taint feedback, and TCL.

5.3.2 Problem: instrumentation

RQ4: How are WUTs instrumented to capture security-critical behaviours and responses?

Code instrumentation means an instrumentation tool will put some probes in the WUT's source code to collect software internal states (e.g., how much code the WUT reaches during the execution) [44]. The instrumentation can be done either in a static or dynamic way, in which the former means it is performed at compilation time and has less run-time overhead; meanwhile, the latter is done at run-time and has better performance [63]. Since the instrumentation enables detailed tracking of code execution, it empowers fuzzers to systematically explore HTTP requests that reach more vulnerable codes and deeper statements. Even though the instrumentation is helpful for fuzzers, it is only employed by a few prior fuzzing frameworks because its implementation is challenging.

Source code augmentation Putting probes in the source code or byte code level is the common way to instrument applications, in which around 32% of prior studies adopt this technique. EvoMaster [5] developed new instrumentation tools to get code coverage from various WUT platforms. First, for Java-based WUT, Arcuri et al. [6] instantiate a Java agent to intercept all class loadings and add probes in the bytecode. On the other hand, for Javascript-based WUT, Zhang et al. [99] develop a plugin for Babel (a JavaScript transpiler) to create an instrumented version of the WUT which contains probes in the source code. Rather than employing ready-to-use third-party libraries (e.g., JaCoCo), designing these custom tools enables EvoMaster to add extra features (e.g., calculating the branch distance) for the mutation feedback. Figure 8 illustrates the instrumented version of the code.

Van Rooij et al. [80] created an instrumentation method to get live information from PHP-based WUT. It instruments the application in the AST (Abstract Syntax Tree) to catch basic block or branch coverage. AST is a tree structure representing source code syntax without showing the details that can be used to identify statements or declarations in the program

```
let x = 0;
```

(a) Original code

```
const __EM__ = require("evomaster-client-js")
    .InjectedFunctions;
__EM__.registerTargets(["File_test.ts","
    Line_test.ts_00001","Statement_test.
    ts_00001_0"]);
__EM__.enteringStatement("test.ts",1,0);
let x = 0;
__EM__.completedStatement("test.ts",1,0);
```

(b) Instrumented version

Fig. 8 An example of the instrumentation process taken from the paper of Zhang et al. [99], in which 8a is the original code written in typescript and 8b is the instrumented version

[96]. Using this approach, this instrumentation method parses each file using the PHP-Parser library and identifies basic blocks during the AST traversal process. Then, it puts probes at the beginning of a basic block, such as the first statement in a function definition or a control function.

Interpreter augmentation Instead of augmenting the source code as explained above, Trickel et al. [82] prefer to augment the interpreter application. In the context of web applications, an interpreter is a component running in the web server (recall Figure 1) that reads the source codes of the server-side web application and translates them into byte-code instructions without the need for a separate compilation step. Their work augmented the interpreter by modifying and recompiling its source codes. The augmented interpreter will then call the proposed library function, Witcher's Coverage Accountant, to send the line number, opcode, and parameter of the current byte-code instruction during the WUT execution. The Accountant function aims to measure the test coverage from the sent data.

Answer to RQ4: Some existing web API fuzzer studies (32%) use source code or interpreter augmentation (2%) for the instrumentation. The others (66%) do not instrument the WUT because they use the black-box testing approach.

5.3.3 Problem: observed vulnerability

RQ5: What vulnerabilities are observed?

WUT generally exposes its vulnerabilities when receiving a request raising inevitable errors. Determining vulnerabilities to look for is a crucial issue because it highly influences the fuzzer design. Basically, the vulnerabilities observed by the existing web API fuzzers can be grouped into two big categories: crash-forming and non-crash-forming vulnerabilities.

Crash-forming vulnerabilities Crash-driven fuzzers generate inputs to make software crash and look for any vulnerability behind the crash. The vulnerabilities found in this way can be called crash-forming vulnerabilities. Most of the existing web API fuzzers (74% of existing papers) report the WUT crash (i.e., internal server error) by using HTTP status codes. For example, EvoMaster [5], RESTler [12], RestTestGen [85], RESTest [68], foREST [59], and RestCT [89], utilise this technique to know the success of the injected inputs by noting how many HTTP requests triggered 500-response codes. Then, the fuzzer users must check internal error information (e.g., full stack traces of thrown exceptions) to know what actual error is raised. However, to detect more vulnerabilities, the users cannot rely solely on this information because most web vulnerabilities do not manifest themselves as error status codes. Another issue with relying solely on error codes is a 5xx-response code does not always mean a software fault if the web API is connected to external services [64]. For example, a fuzzer sends a request to the first web service that relies on the second service. If the second is down, the first will return a 5xx-response code even though the first service runs normally without error.

Non-crash-forming vulnerabilities

In addition to the crash, to ensure malicious parties cannot exploit the WUT, some web API fuzzers catch specific vulnerabilities that do not form a crash. To detect these vulnerabilities, web fuzzers must be equipped with typical bug catchers, as follows. To improve the RESTler capabilities [12] for capturing specific bugs (resource violation), Atlidakis et al. [11] designed four security rules, namely the use-after-free rule, resource-leak rule, resource-hierarchy rule, and user-namespace rule. These rules are checked after getting the web responses. Take the second rule as an example. If a child resource of a parent resource can be accessed from another parent resource, it means the response breaks the resource-leak rule and is classified as a bug. Corradini et al. [19] improved the RestTestGen's capability [85] in catching mass assignment vulnerabilities that can happen when external users can manipulate the value of a resource meant to be read-only by exploiting a misconfiguration of the automatic parameter binding. The authors proposed observing the WUT by checking whether the read-only attributes can be overwritten and whether they differ from their default values.

Another web fuzzer, BackREST, was designed to catch web vulnerabilities like XSS, SQL injection, Command injection, and Denial of Service (DoS). It also enhanced its work with taint feedback to detect more SQL and Command injection vulnerabilities. Van Rooij et al. [80] designed a fuzzer to catch stored and reflective XSS vulnerabilities. Initially, their fuzzer injects XSS payloads in the HTTP request parameters, which will then call the alert function. Parsing Javascript code in the HTML responses using *esprima*[6] library, WebFuzz –their fuzzer's name–

[6] https://github.com/Kronuz/esprima-python

will check the corresponding alert function call. If affirmative of containing the alert, it can be concluded that the web application is vulnerable to XSS. In addition to WebFuzz, other web fuzzers used a similar approach to detect XSS bugs, such as State-Aware Vulnerability Scanner [26], KameleonFuzz [28], Cefuzz [102], and ZokFuzz [95].

Answer to RQ5: Most web API fuzzer studies (74%) observe crash-forming vulnerabilities using the 500-response code. The others look for specific web vulnerabilities, namely the violation of certain security-related rules, XSS, SQL injection, and command injection.

5.4 Mutation

RQ6: How are the existing input spaces expanded to uncover deeper or previously unseen security vulnerabilities?

The mutation process is the core of the fuzzing method because it gradually expands the existing input to uncover deeper vulnerabilities and find more relevant input space. Running a fuzzer for a long time will allow the mutator –the fuzzing's component doing the mutation process– to generate more inputs to explore execution paths that contain uncovered vulnerable code. However, suppose the mutator is poorly designed and does not suit the characteristics of the targeted application. In that case, no matter how long the fuzzing continues, its effectiveness will remain low (e.g., no bugs will be found or code coverage will be flat). The following are several problems related to mutator design and solutions provided by previous works.

5.4.1 Problem: fast mutation

Before advanced techniques appeared, a fuzzer made simple modifications to existing input values to have a fast mutation process. The modifications, such as random bit flipping or byte deletion, can gradually discover unexpected behaviour in PUTs. However, they are not enough in the web application context because they are commonly intended for finding memory-related vulnerabilities in binary applications. Therefore, existing web API fuzzing proposals devise another way to do fast mutation to make the mutation process more suitable for web domains, as follows.

Randomly modified values (RMV) Creating random data may be the most straightforward technique for mutation in all fuzzing types, including web API fuzzing. However, web API fuzzer rarely applies mutation at the bit level. The work of Lei et al. [53] showed that most RESTful web applications use around 67% string-typed and 32% number-typed parameters. Therefore, all web API fuzzers create new mutated HTTP requests using string modification (e.g., duplicate random string or swap string position) or number operation techniques (e.g., multiplication) to fill the parameter values. These techniques can trigger web crashes since some WUTs are not capable of handling unexpected strings (recall Section 5.3.3).

5.4.2 Problem: code region exploration

Since the random value modification can result in either valid or invalid values, the fuzzing framework should be careful when using that technique. Aggressively using a random generator may damage the data structure, leading to the rejection by WUT [80]; however, data changes that are too small may not be enough to trigger new execution flows. Since in the initial phases web API fuzzers need to explore more code regions before exploiting them with mutation attacks, previous studies designed several mutation strategies only to generate requests that bring more code coverage, as follows.

Response dictionary (RD)

Besides using random data, Viglianisi et al. [85] use a response dictionary containing a map between field names and their valid values to complement the default dictionary. The valid values come from the values that appear in the valid HTTP responses. Reusing the already tested values is effective, especially for certain types whose values are generated by the WUT, such as id. However, because of the developer's carelessness, the responses that are supposed to use the same parameter names may bring names that are slightly different to the existing ones yet are the same semantically. For instance, a response from the WUT contains *nameid*, but another response from a different function carries *id_name* instead of *nameid*. Hence, the authors also prepared strategies to match similar field names. They match those names with other variations that may happen (e.g., *nameid* should be matched to *id_name*). Lin et al. [59] also use the response dictionary idea to build a hierarchically tree-shaped resource pool to hold each resource's possible values. This pool data is extracted from response messages.

Corpus mutation (CM) In the context of web API fuzzing, a corpus is the place to put interesting HTTP requests (e.g., a request that brings a crash). Some researchers maintain this corpus to reuse such requests in the mutation process. For example, Trickel et al. [82] blend the combination of parameter names and values between the requests stored in the corpus to generate a mutated request.

Adaptive hypermutation (AH) Doing random mutation or using the response dictionary can be tricky for web API fuzzing because a WUT may treat each request parameter differently. For example, in a WUT, an *id* field can be checked multiple times because it may relate to another data table, so it must be correct. Still, a *name* may be automatically used without performing a deep examination, so using very varied names will always result in the same successful response

Fig. 9 Illustration of the adaptive hypermutation proposed by Zhang et al. [97]. One individual consisting of some actions (e.g., HTTP requests) has many genes, and each gen has a different probability of being selected for mutation. The probability depends on the impact or feedback from the WUT. The impact will also influence how narrow the mutation range is

from the WUT. Zhang et al. [97] explained this situation in detail in which some parameters do not impact WUT execution flow. Spending too much time mutating them will be useless because it will not improve the effectiveness of the fuzzer.

Therefore, inspired by gene mutations, they proposed an adaptive hypermutation to select and mutate genes adaptively based on their feedback and mutation history. This strategy is called hypermutation because it applies high mutation rates, leading to the mutation of multiple genes on average. One gen corresponds to one HTTP request parameter with either an atomic type (e.g., integer or boolean) or artificial types consisting of other genes (e.g., Date object consisting of day, month, and year). The feedback can be how many fuzzing objectives (e.g., code coverage) are achieved or the branch distance score (recall Section 5.3.1). Based on the mutation history, this strategy will only select genes that can help achieve the fuzzing objectives. The selected genes will be mutated differently depending on the type. For example, the mutated integer value will be calculated using formula $a \pm 2^i$ which a is the old value and i is the random value within an adaptive range that depends on the impact score. For more detail, see an illustration in Figure 9. The figure explains that only chosen genes, instead of all, will be mutated in the mutation process.

Attention-based models (AM) Lyu et al. [62] employs a trending machine learning method, an attention-based model, to help produce plenty of test cases that have more impacts on the WUT. They utilise such a model to learn the implicit relationship between parameter name-value pairs. They designed the model consisting of a Gated Recurrent Unit (GRU) neural network [18], an attention layer [61], and a linear layer to result in valid param-value pairs for each request name. During the fuzzer runs, the collection module collects requests with valid responses to be set up as the training data for the model, in which a single row of the training data is in the form of $< request_name >< param_value_1 > .. < param_value_n >$. The fuzzer retrains the model every two hours from scratch.

5.4.3 Problem: more malicious requests

Generating error cases can be tricky because the fuzzer cannot simply use random data. Since the malformed input is expected to test exceptional application scenarios, utilising the random data generator is insufficient to assess all possible error schemes. Fuzzer should introduce the faulty inputs slowly to reach deeper bug codes hidden in deeply nested conditionals. Most web API fuzzing frameworks adopt this idea and propose several strategies as follows.

Constraint violation (CV) Constraint violation techniques are the most commonly used techniques to create incorrect input and potentially cause WUT to crash. Fuzzing generally knows the constraints in filling valid values to the HTTP request, so at this time, it just ignores the constraint rules. This technique involves: 1) changing the data type of request parameters (e.g., from String to an array) [80], 2) missing the required request parameters [85], 3) violating the specific field's constraint (e.g., giving a long string for a field setting up the *maxLength* constraint). However, implementing this strategy too aggressively in a single HTTP request can lead to "400 bad requests" because the server considers it a malformed HTTP request.

Rule-based schema mutator (RSM) Godefroid et al. [38] formulated some schema fuzzing rules for the schema mutator. The schema in this context is the HTTP request body consisting of parameter name-value pairs. This schema can be seen as a tree structure because certain parameters may have child parameters. The schema fuzzing rules involve the node rule, which defines the node modification techniques, and the tree rule, which determines which part of the schema will be fuzzed. The node rules cover dropping one child node, removing all nodes and keeping only one node, duplicating a new child node, and changing the node type. Then, the tree rules encompass picking a single node, choosing a path consisting of several nodes, or selecting all nodes, to apply to the node rules.

Vulnerability dictionary (VD) To trigger particular vulnerabilities, Gauthier et al. [33] fill specific HTTP parameter values with the values from a pre-defined dictionary of vulnerable payloads. This dictionary contains a set of vulnerable payloads mapping to certain vulnerability types. For example, when fuzzing requests a SQL injection payload for the name field, the dictionary will deliver strings like `' OR '1'='1' --`. Van Rooij et al. [80] employ a real-life XSS payload dictionary to bring their fuzzing framework effective malicious data. Zhao et al. [102] determined special seeds containing commands or functions (e.g., command 'system' in Linux) for triggering a PHP Remote Command/Code Execution (RCE) vulnerability. Based on the mutation rule, their fuzzing framework will take a particular seed to be combined with the formula forming the test case. Zhang et al. [95] use

the same concept, but the initial seeds contain SQL injection payload.

Tracked fault generator (TFG) Laranjeiro et al. [51] employ a fault generator to insert fault codes into valid API requests guided by a Fault Mapper. The mapper tracks the inserted faults and their injection location in API requests to avoid exploring already visited locations. The fault dictionary produces concrete faults containing 57 fault rules, such as replacing valid values with null, removing random elements in an array, and others.

Answer to RQ6: To expand the existing input to uncover deeper security vulnerabilities, most web API fuzzers use randomly modified values and dictionaries (either response or vulnerability dictionary). The others use corpus mutation, adaptive hypermutation, attention-based model, constraint violation, rule-base schema mutator, and tracked fault generator.

5.5 Choosing WUT for experimental evaluation

RQ7: What security benchmarks are used for empirical evaluations?

All fuzzing researchers conducted experimental studies using various benchmarks to evaluate whether the proposed works were better than the previous ones. Generally, they test their fuzzers on third-party security benchmarks, public (online) web applications, or local benchmarks they prepare.

5.5.1 Third-party security benchmarks

To evaluate their fuzzers in catching security-related bugs, around 21% of the existing studies employ third-party security benchmarks as their WUTs. The third-party benchmarks, also called test-bed applications, are designed by other people to contain many bugs for cyber security learning [23]. Some examples of test-bed applications are *WebGoat*[7] and *Gruyere* [8], which ones among the targets of KameleonFuzz [27]. Those vulnerable web applications, built as web testing education, were proven to contain XSS bugs. Compared to other scanners, KameleonFuzz with LigRE [28] could detect more true XSS bugs in those applications. Another work doing similar things is BackREST [33]. It took test-bed applications built in Node.js platforms, such as *Nodegoat*[9]. It also compared its performance to vulnerable scanners like Arachni[10] and OWASP ZAP[11] to detect SQL injection, command injection, XSS, and DoS vulnerabilities. The result was BackREST could catch the vulnerabilities more than the scanners. The last example is Witcher [82], which used known vulnerable applications as its fuzzer targets, consisting of 8 PHP-based web, 5 C-based firmware images, 1 Java-based web, 1 Python-based web, and 1 Node.js-based web application, with a total of 36 known vulnerabilities. The experiments showed that Witcher was better than Burp[12] and other tools in either bug finding or code coverage. Similarly, Cefuzz [102] and ZokFuzz [95] used some PHP-based vulnerable web applications as their target, such as DVWA[13].

5.5.2 Self-developed benchmarks

In addition to leveraging third-party benchmarks populated with known vulnerable web applications, researchers in web API fuzzing can build their custom benchmarks by deploying popular open-source web applications. While this approach requires extra work to identify or introduce security vulnerabilities to the collected applications, it offers the advantage of performing fuzzing on more modern web environments. Approximately 45% of prior research chose to build their own benchmarks to meet their specific evaluation needs. For example, EvoMaster Benchmark (EMB) [10] holding various web applications, such as Java, Kotlin, JavaScript, and C# was developed to test EvoMaster [6]. On the other hand, the work of Van Rooij et al. [80] proposed an actual bug injection methodology into PHP-based web applications to create a proper benchmark. Getting inspiration from similar models like LAVA [25], they gave a standard way to evaluate identical fuzzers in finding web vulnerabilities. Their benchmark contained CE-Phoenix[14], Joomla[15], and others that had been injected with XSS vulnerabilities

5.5.3 Public WUT

Some prior studies (34%) chose public web applications as the fuzzer target. Even though those applications can be relatively easy since researchers do not need to install and deploy them, they have some limitations: no source code and no instrumentation. No source code means the fuzzer cannot analyse the code workflow to generate better test cases, and no instrumentation means the fuzzer does not know what is happening during the execution of the WUT. Therefore, the WUT must be treated as a black box. Fuzzing researchers can consider public web applications listed in the *APIS.Guru*[16]

[7] https://github.com/WebGoat/WebGoat

[8] https://google-gruyere.appspot.com/

[9] https://github.com/OWASP/NodeGoat

[10] https://www.arachni-scanner.com/

[11] https://www.zaproxy.org/

[12] https://portswigger.net/burp/vulnerability-scanner

[13] https://github.com/digininja/DVWA

[14] https://phoenixcart.org/

[15] https://www.joomla.org/

[16] https://apis.guru/browse-apis/

website. Viglianisi et al. [85] used the website as the reference to test 87 web APIs, perform 2612 testing operations, and find 151 internal error faults. Furthermore, Laranjeiro et al. [51], who designed bBOXRT implemented in Java, tested 52 public REST services comprising 1,351 operations. The fuzzer detected at least one robustness problem (explained in Section 5.3.3) in half of the services tested.

On the other hand, some fuzzing papers considered popular and public RESTful Web API as their target. For example, Atlidakis et al. [12] tested their fuzzer on some popular online RESTful APIs: three *Azure*[17] and one *Microsoft Office365*[18] online services. Their fuzzer found some bugs in those services that were 500-response codes. In addition, they also conducted an experiment on GitLab API[19] deployed on a local server and found bugs in commit and branch-related operation APIs. The other web API fuzzer, RESTest [67], also performed similar experiments, evaluating its performance on some popular web applications: GitHub[20], Foursquare[21], and others. It got bugs related to server and client error response codes. Another example is Peng et al. [76] tested their fuzzer to ByteDance[22].

Answer to RQ7: For empirical evaluations, most studies (45%) develop new benchmarks based on popular open-source web applications. Other works use publicly available WUT (34%) or employ third-party security benchmarks (21%).

5.6 Summary

Based on the explanations in prior sections, we summarise the information on the existing web API fuzzers in Table 4. Instead of paper-based, we resume the insight based on the fuzzer the researchers introduced in their works. We got 19 distinct web API fuzzers even though we reviewed 47 papers because one framework can be explained or improved by multiple studies. For example, EvoMaster is used in nine studies [6] [5] [8] [7] [100] [9] [97] [39] [99]. We sorted the frameworks based on the year they were published first. Most of them are black-box fuzzing, which only relies on HTTP responses. The others also utilise the HTTP response but are supplemented by other feedback, as listed in the table.

[17] https://azure.microsoft.com/en-us/

[18] https://www.office.com/

[19] https://docs.gitlab.com/ee/api/

[20] https://github.com/

[21] https://foursquare.com/

[22] https://www.bytedance.com/en/

6 Open challenges

RQ8: What open challenges are identified?

Section 5 has explained a variety of prior approaches and techniques to enable fuzzing strategies for web application security testing. Despite the progress made, some shortcomings and challenges still need to be addressed. This section discusses open challenges in web API fuzzers mentioned in the summarized studies. The challenges are extracted from the relevant sections of each reviewed paper. The challenges mentioned in those papers that are tackled by more recent studies are excluded from the discussion.

6.1 Less effective source code instrumentation

In many contexts, the use of grey or white-box web API fuzzers might provide better results for fuzzing effectiveness, assuming that access to the WUT source code is possible. However, using these fuzzing approaches may introduce an additional overhead due to the required preparation time (e.g., instrumentation) before testing. As explained in Section 5.3.2, various instrumentation methods have been developed over the years for interpreted web applications because these applications are popular WUTs for web API fuzzers. However, whether the instrumentation pays off is a valid question.

Generally, the instrumentation makes the source code bigger and slows down the server execution. It is caused by the probe code placed everywhere [80] in the WUT. In addition, the instrumentation process forces the WUT to be re-compiled and rebuilt, which certainly takes a significant amount of time. According to our preliminary experiments in relatively big Java-based web applications using EvoMaster fuzzer [5], carrying out fuzz testing for a short duration (e.g., 1 hour) is often ineffective. This is because the fuzzing process is only carried out after the re-compile and re-build process has been completed, and in some cases, the compile and build processes may take longer than the fuzzing process itself. The problem becomes more acute when the web developers are expected to deliver the application as quickly as possible, which may render the instrumentation process unacceptable. Therefore, the existing instrumentation techniques can be less effective in real-world settings where WUT consists of many files and the development is fast-paced.

Designing a custom interpreter to catch interesting execution paths, rather than instrumenting the source code, can be a possible workaround to this problem because the majority of web applications are interpreter-based. Trickel et al. [82] have applied this idea to augment the PHP interpreter tool for coverage analysis. However, there is still no experimental study that compares fuzzing with instrumentation and fuzzing without instrumentation (e.g., interpreter augmentation) in terms of effectiveness.

Table 4 Web API fuzzer comparison.

Fuzzer name	Method	WUT	Template generation	Mutation	Feedback	Vulnerability
KameleonFuzz [27]	Vul ■	3rd-party benchmark	HTML crawler [28]	VD	Taint feedback	XSS
EvoMaster [6]	Crs □	JVM-based, NodeJS-based [99], .NET-based [39]	Smart sampling [7]	AH [97]	Branch distance	Crash
RESTler [12]	Crs ■	Public web	Dependency graph	RSM [38]	HTTP response	Crash, Resource violation [11]
RestTestGen [85]	Crs ■	Public web	Dependency graph	RD	HTTP response	Crash, Mass assignment [19]
bBOXRT [51]	Crs ■	Public web	No dependency	VD	HTTP response	C.R.A.S.H.
RESTest [67]	Crs ■	Public web	No dependency	DBPedia + IDL	HTTP response	Crash
WebFuzz [80]	Vul ◐	PHP-based	HTML crawler	VD	Code coverage	XSS
HsuanFuzz [84]	Crs ■	3rd-party benchmark	Pre-defined template	RMV	TCL	Crash
BackREST [33]	Vul ◐	NodeJS-based	JS Crawler	VD	Taint feedback	SQLi, CMDi, XSS, DoS
RestCT [89]	Crs ■	Public web	Tree-based structure	RD	HTTP response	Crash
Cefuzz [102]	Vul □	PHP-based	No dependency	VD	Taint feedback	PHP RCE
Zokfuzz [95]	Vul ■	PHP-based	No dependency	VD	HTTP response	SQLi, XSS
MACROHIVE [35]	Crs ◐	Microservice benchmark	Dependency graph [36]	RD	HTTP response	Crash
WAPT [13]	Vul ■	3rd-party benchmark	HTML crawler	VD	HTTP response	XSS, etc
foREST [59]	Crs ■	Public web	Tree-based structure	RD	HTTP response	Crash
Witcher [82]	Vul ◐	PHP-based, Python-based, Java-based, NodeJS-based, Ruby-based	HTML & JS crawler	CM	Code coverage	SQLi, CMDi
MINER [62]	Crs ■	Public web	Length-oriented selection	AM	HTTP response	Crash
NAUTILUS [24]	Crs ■	3rd-party benchmark	OpenAPI annotation	VD	HTTP response	SQLi, Privilege Escalation, etc
Leif [53]	Crs ■	Public web	FET inference	RMV	HTTP response	Crash

Crs = Crash-driven fuzzing, Vul = Vulnerability-driven fuzzing, = open-source project, ■ = black-box, ◐ = grey-box, □ = white-box, = utilize Open-API Spec, RMV = Randomly modified value, RD = Response dictionary, CM = Corpus mutation, AH = Adaptive hypermutation, AM = Attention-based model, MIO = Many independent objective, CV = Constraint violation, VD = Vulnerability dictionary, RSM = Rule-based schema mutator, TFG = Tracked fault generator

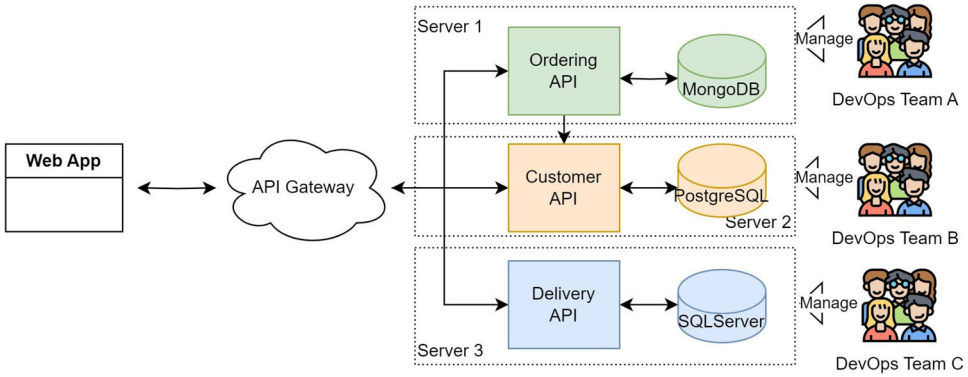

Fig. 10 Illustration of a web application using the micro-service approach. Each service treats other services as black-box systems, which means it only sends inputs and receives outputs through provided communication channels and interfaces without having any internal information. Solely testing the API gateway can be misleading since the tester does not know which service crashes

6.2 Complexity of handling microservice architecture

Most world-leading web applications are using microservice architecture because it brings many benefits, such as ease of maintenance [55]. Since this kind of application is designed to be distributed, and each microservice typically operates independently, testing interactions between these distributed components can be complex [15] (see Figure 10). This complexity presents unique testing challenges because the software tester must isolate each service properly to be able to distinguish the regions of the code where the problems arise. When only testing the main service without considering such complex interactions behind it, it may look as if the main service crashes often even though the real problem originates from other services that do not respond properly. In the microservice architecture, the main services exposing APIs to external users are called **edge microservices**; the others are called **internal microservices** because they expose their APIs only for internal services [45]. In addition, employing existing web API fuzzing techniques that are not designed for microservice applications may require the application testers to install and configure fuzzer for each service. It means that the fuzzer does not treat the WUT as a whole. The web API fuzzing framework should ideally be able to handle this enterprise architecture at scale.

Recently, researchers designed testing approaches to handle microservice architecture. For example, Giamattei et al. [35] proposed a grey-box strategy for automated testing and monitoring of internal microservices interactions. Their recent experiment [36] showed that their proposed framework delivered crucial information about internal coverage and failure, and inferred causality in failure chains. Zhang et al. [101] also developed a white-box fuzzer specially designed for RPC-based APIs that are commonly used in microservice architectures. However, both studies stated that further reearch needed to improve the fuzzers in this setting.

6.3 The difficulty of testing public WUT

Most reviewed fuzzing papers use online web applications (e.g., Youtube API[23]) as the WUTs since they are popular and used by a large number of people. Even though preparing those WUTs for testing is easy: no installation or deployment is required before fuzz testing, some challenges exist that make their testing difficult.

Firstly, since the majority of those popular web applications are closed-source, users can only test them using a black-box web API fuzzer, leading to code coverage ignorance. Without code coverage information, the fuzzers likely test only a small part of the WUT, which may render their results invalid. Secondly, popular web applications managed by big companies usually prohibit massive requests from the public [89]. Besides determining low web resource quota, they also usually set up short-lived access tokens [12] that require the users to do the authentication process more often. These two challenges make use of (online) closed-source WUTs for evaluating fuzzers less ideal.

Lastly, even if the researchers employ open-source WUTs (e.g., WordPress[24]), performing white-box fuzzing over these WUTs, that is, analyzing the source code to get more comprehensive understanding of the code, is difficult because those WUTs are often large projects with complex source codes [82]. This situation requires the fuzzers to understand the recent programming features the WUT uses, such as object-oriented technique and MVC (model-view-controller) architecture. Therefore, more studies need to address this last challenge to improve existing web API fuzzers.

6.4 Lack of corpus diversity

Initial test cases play an essential role in all kinds of fuzzing since they are the initial corpus for the mutation process. Corpus in the web API context is the HTTP request sequence collection. If the corpus provides highly diverse, valid inputs,

[23] https://developers.google.com/youtube/v3

[24] https://wordpress.org/

the work of the mutator can become easier because the mutator does not need to explore many more execution paths; it "just" needs to trigger any error in visited vulnerable codes. As explained in Section 5, existing web API fuzzing frameworks usually use only OpenAPI documents or HTML pages to create the initial corpus. Those documents are useful to help the fuzzer; however, some studies have shown that those documents may be incomplete. For example, the work of Deng et al. [24] states that the information extracted from such documents is insufficient to produce diverse yet correct requests. Relying solely on these limited documents to generate an initial corpus for fuzzing can be risky because it produces low-quality corpus that will significantly affect the overall fuzzer performance.

Even though more comprehensive OpenAPI documents can result in a bigger corpus with more HTTP requests, the corpus should be minimised before use, which means there should be a process to discard similar requests that lead to the same execution paths. The study from Herrera et al. [43] shows that the minimised corpus can lead the fuzzer to explore new execution paths faster because it does not need to waste time executing inputs that produce an already known output. To our knowledge, most existing works on web API fuzzer do not perform the corpus minimisation process yet.

6.5 Lack of web API fuzzing benchmarks

Researchers have developed various web API fuzzers using black-box, grey-box, or white-box approaches, and more are expected to come. These developers of fuzzing frameworks often claim that their fuzzers are really good at finding web vulnerabilities over a certain set of WUTs. However, there is neither a widely accepted security benchmark for web API fuzzers nor established security-relevant metrics in comparing the performances of the fuzzers. For a proper and fair evaluation and comparison of fuzzers, comprehensive information about bug numbers detected by the fuzzers, including the miss and false bugs rate, is needed. Therefore, it is essential to develop a particular benchmark for web API fuzzers. Similar to binary fuzzing benchmarks (e.g., [42]), the Web API fuzzing benchmarks should contain a diverse set of WUTs with injected bugs or vulnerabilities, integrate some of the well-known fuzzers and some scripts to automate all processes: deploying the WUTs, injecting the vulnerabilities, starting the fuzzers, and counting all evaluation metrics.

Even though there is no extensive web API fuzzing benchmark yet as in binary fuzzing, some benchmarks that have been made publicly available, as (explained in Section 5.5.2). For example, Arcuri et al. [10] released EvoMaster Benchmark (EMB), a set of comprehensive WUTs for evaluating EvoMaster fuzzer. However, the WUTs in EMB are original applications released by other developers, without annotated security-related bugs inside them. Without the bug information, benchmark users cannot compare the web API fuzzer performance in catching bugs.

Answer to RQ8: Identified open challenges are less effective source code instrumentation in huge web applications, the complexity of handling microservice architecture, the difficulty of testing the public WUT, lack of corpus diversity, and lack of web API fuzzing benchmark.

7 Potential research directions

Apart from the open challenges described in Section 6, we identified some potential research directions related to web API fuzzing. In doing so, we have taken into account the technologies that are currently being developed: web client programming, mobile web, and generative AI.

7.1 Fuzzing for web client programming

Given the increasingly sophisticated web applications, the load on web servers has been growing. For example, recommendation services employing artificial intelligence methods in marketplace platforms are resource-hungry. The work of Wai et al. [87] investigates web client programming as one of the solutions to distribute the server load. This programming paradigm tries to move some of the computational processes from servers to client devices. This approach is reasonable since smartphones and computer desktops will constantly evolve and bring new hardware features to carry out sophisticated computations. Nowadays, while Java-script language can be considered as the main web client programming language, it is expected that more client programming languages that offer more portable and efficient features will be adopted in the future. For example, WebAssembly, developed by Haas et al. [41] together with the W3C community to bring low-level C programming into the web, has attracted much attention from the fuzzing community. Therefore, employing fuzzing for this application type can become popular in the future.

7.2 Fuzzing mobile web applications

Many web pages are prepared/delivered explicitly for/to mobile devices [81]. Even though mobile web applications are generally similar to regular web applications, there are certain differences. Both usually have the same content sources, but they are rendered differently depending on the client device's capabilities. For example, the mobile web does not show complex user interfaces in order to provide better user experiences on a smaller screen. Next, some mobile hardware limitations may cause the mobile device not to execute complex client-side (e.g., Javascript) code. Considering

the growth of the smartphone market, many more mobile web applications are expected to be launched. Testing this type of WUTs is challenging because fuzzers often demand more sophisticated computational resources that the mobile devices do not have. Developing a fuzzing approach for smartphones or emulating the mobile web on a desktop can be a possible alternative even though there are certain drawbacks of this approach. For example, mobile web emulation can be less accurate since mobile devices are highly diverse, from old to cutting-edge systems. Whatever approach is chosen, anticipating this technology as early as possible is a reasonable consideration.

7.3 Generative AI for web fuzzing

Generative artificial intelligence (AI) is a class of AI techniques and models that can generate new content, such as text, images, music, or other forms of data [30]. These models are designed to learn the patterns and structures present in the training data and then produce outputs with similar characteristics. Generative AI, a branch of machine learning, has become increasingly popular in diverse domains including software testing as it can generate content that was not explicitly present in the training data. Some reviewed works in web API fuzzing already utilised powerful AI techniques partially in some steps. For example, Lyu et al. [62] employed the attention model for filling values in the template rendering stage. It is estimated that more papers will adopt generative AI techniques fully to generate HTTP requests to test web API in the future.

7.4 Support for diverse web security vulnerabilities

In addition to using the existing OWASP vulnerability list [74] to define vulnerabilities to be detected, developing strategies for the detection of new bug types that are different from the well-known ones is a promising research direction. Since the OWASP list is based on past investigations, imagining far ahead about potential faults that may not have happened yet is essential, especially by considering/predicting user habits and bugs that may occur in the future. For example, the work of Atlidakis et al. [11] identified a new vulnerability family that can result in the hijacking of a WUT and developed a method to detect bugs related to this vulnerability. They defined four new security rules related to web resource management: use-after-free, resource-leak, resource-hierarchy, and user-namespace rules. If the web API violates one of those rules, the authors conclude that the WUT has this vulnerability.

Extending the existing bug criteria can also be a good option besides specifying completely new security bugs. This is also important in enabling security practitioners to catch complex bugs easily. For instance, the work of Pan et al. [75] developed a strategy to detect Excessive Data Exposure (EDE) vulnerability. Even though OWASP included this vulnerability, catching this bug is not easy since it does not trigger a crash or an unexpected behaviour. Therefore, determining more precise test oracles will help fuzzers to recognise the vulnerability. Finally, among the vulnerabilities covered by the OWASP list, only two types, SQL/code injection and XSS, have been studied extensively. Effective detection strategies for the other vulnerabilities are still quite open to research.

8 Threats to validity

Paper selection Choosing relevant papers from search databases and filtering irrelevant studies requires manual effort and expertise. As a result, this process might be susceptible to human error. The authors did searching and filtering of the paper collection more than two times to reduce that possibility.

Information extraction Analysing and extracting relevant information from the selected papers was done manually too. Although the authors are confident with the results, they might also be susceptible to human misunderstanding. To reduce this chance, the authors read and analysed the papers more than two times.

9 Conclusion

In this survey paper, we reviewed articles presenting research results on web API fuzzing frameworks. We classified them according to their testing objectives and techniques used to generate valid HTTP requests for attack surface exploration, utilise security feedback from the web under test (WUT), and mutate existing requests to uncover more security vulnerabilities. In addition, some insights about open challenges and anticipated research related to web API fuzzing are provided. Ultimately, we aim this paper to serve as a foundation for further research in web API fuzzer.

Appendix A OpenAPI Specification Example

```
openapi: "3.1.0"
info:
  title: API Example
paths:
  /book:
    get:
      parameters:
        - name: limit
          in: query
```

```
          description: How many items
    to return at one time (max 100)
          required: false
          schema:
            type: integer
            maximum: 100
            format: int32
       responses:
         '200':
          description: A paged array of
     books
          headers:
            x-next:
              description: A link to
    the next page of responses
              schema:
                type: string
          content:
            application/json:
              schema:
                $ref: "#/components/
    schemas/books"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/
    schemas/Error"
```

Appendix B Grammar example generated by RESTler [12]

```
Request(
    static("GET /api/customer/data"),
    static("?idcustomer="),
    fuzzable("integer"),
    static("&name="),
    fuzzable("string"),
    static("HTTP/1.1"),
    static("Accept:application/json"),
    ......
)
```

Acknowledgements The authors thank Tariq Bontekoe for proofreading this article.

Author Contributions The first author conducted the literature survey and wrote most of the paper; the second author contributed to the discussions and reviewed the paper; and the third author contributed to the discussions, writing and reviewing of the paper.

Funding The first author has received scholarship funding from the Center for Financing of Higher Education (BPPT) and the Indonesia Endowment Fund for Education (LPDP) under the Indonesian scholarship schema.

Declarations

Conflict of Interest The authors do not have any financial or non-financial interests to disclose that are relevant to the content.

Ethical Approval Not applicable.

References

1. Alonso, Juan C.: Automated generation of realistic test inputs for web APIs. en. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Athens Greece: ACM, Aug. (2021). pp. 1666–1668. https://dl.acm.org/doi/10.1145/3468264.3473491
2. Alonso, Juan C. et al.: ARTE: automated generation of realistic test inputs for web APIs. In: IEEE Transactions on Software Engineering. Conference Name: IEEE Transactions on Software Engineering, pp. 348–363. (2023). https://doi.org/10.1109/TSE.2022.3150618
3. Amankwah, Richard et al.: An empirical comparison of commercial and open-source web vulnerability scanners. en. In: Software: Practice and Experience. pp. 1842–1857. (2020). https://doi.org/10.1002/spe.2870. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2870
4. Zeller, Andreas et al.: The Fuzzing Book. (2023). https://www.fuzzingbook.org/
5. Arcuri, A.: EvoMaster: Evolutionary Multi-context Automated System Test Generation'. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). IEEE pp. 394–397, (2018) https://doi.org/10.1109/ICST.2018.00046
6. Arcuri, A.: RESTful API Automated Test Case Generation. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE pp. 9–20, (2017) https://doi.org/10.1109/QRS.2017.11(2017)
7. Arcuri, A.: RESTful API automated test case generation with EvoMaster. en. In: ACM Transactions on Software Engineering and Methodology, pp.1–37, (2019). https://doi.org/10.1145/3293455. https://dl.acm.org/doi/10.1145/3293455
8. Arcuri, A.: Test suite generation with the Many Independent Objective (MIO) algorithm. en. In: Information and Software Technology, (2018). https://doi.org/10.1016/j.infsof.2018.05.003. https://reader.elsevier.com/reader/sd/pii/S0950584917304822
9. Arcuri, A., Galeotti, JP.: enhancing search-based testing with testability transformations for existing APIs. en. ACM Transactions on Software Engineering and Methodology, pp. 1–34, (2022). https://doi.org/10.1145/3477271. https://dl.acm.org/doi/10.1145/3477271
10. Arcuri, A., et al.: EMB: A curated corpus of web/enterprise applications and library support for software testing research. In: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 433–442, (2023). https://doi.org/10.1109/ICST57152.2023.00047
11. Atlidakis, V., Godefroid, P., Polishchuk, M.: Checking security properties of cloud service REST APIs. In: 2020 IEEE 13th

International Conference on Software Testing, Validation and Verification (ICST). IEEE, pp. 387–397, (2020). https://doi.org/10.1109/ICST46399.2020.00046
12. Atlidakis, V., Godefroid, P., Polishchuk, M.: RESTler: Stateful REST API Fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp. 748–758, (2019). https://doi.org/10.1109/ICSE.2019.00083
13. Auricchio, N., et al.: An automated approach to Web Offensive Security. en. In: Computer Communications, pp. 248–261, (2022). https://doi.org/10.1016/j.comcom.2022.08.018. https://linkinghub.elsevier.com/retrieve/pii/S0140366422003267
14. Bizer, C., et al. DBpedia - A crystallization point for the Web of Data. In: Journal of Web Semantics, pp.154–165, (2009). https://doi.org/10.1016/j.websem.2009.07.002. https://www.sciencedirect.com/science/article/pii/S1570826809000225
15. Blinowski, G., Ojdowska, A., Przybyłek, A.: Monolithic vs. microservice architecture: a performance and scalability evaluation. In: IEEE Access. Conference Name: IEEE Access, pp. 20357–20374, (2022) https://doi.org/10.1109/ACCESS.2022.3152803
16. Bülthoff, F., Maleshkova, M.: RESTful or RESTless – Current State of Today's Top Web APIs. en. In: Semantic Web: ESWC 2014 Satellite Events. Ed. by Valentina Presutti et al. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, pp.64–74, (2014). https://doi.org/10.1007/978-3-319-11955-7_6. http://link.springer.com/10.1007/978-3-319-11955-7_6
17. Chen, C., et al.: A systematic review of fuzzing techniques. en. In: Computers & Security, pp. 118–137, (2018). https://doi.org/10.1016/j.cose.2018.02.002. https://linkinghub.elsevier.com/retrieve/pii/S0167404818300658
18. Cho, K., et al. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, pp. 1724–1734, (2014). https://doi.org/10.3115/v1/D14-1179. https://aclanthology.org/D14-1179
19. Corradini, D., Pasqua, M., Ceccato, M.: Automated Black-Box Testing of Mass Assignment Vulnerabilities in RESTful APIs. en. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Melbourne, Australia: IEEE, pp. 2553–2564, (2023). https://doi.org/10.1109/ICSE48619.2023.00213. https://ieeexplore.ieee.org/document/10172607/
20. Corradini, D., et al.: Automated black-box testing of nominal and error scenarios in RESTful APIs. en. In: Software Testing, Verification and Reliability, pp. e1808, (2022). https://doi.org/10.1002/stvr.1808. https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1808
21. Corradini, D., et al.: RestTestGen: an extensible framework for automated black-box testing of RESTful APIs. en. In: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). Limassol, Cyprus: IEEE, pp. 504–508, (2022). https://doi.org/10.1109/ICSME55016.2022.00068 . https://ieeexplore.ieee.org/document/9978261/
22. De, B.: API Documentation. In: API Management: An Architect's Guide to Developing and Managing APIs for Your Organization. Berkeley, CA: Apress, pp. 59–80, (2017). https://doi.org/10.1007/978-1-4842-1305-6_4
23. Deepa, G., Santhi TP.: Securing web applications from injection and logic vulnerabilities: Approaches and challenges. en. In: Information and Software Technology, pp. 160–180, (2016). https://doi.org/10.1016/j.infsof.2016.02.005. https://linkinghub.elsevier.com/retrieve/pii/S0950584916300234
24. Deng, G., et al. NAUTILUS: automated RESTful API vulnerability detection. In: Proceedings of the 32nd USENIX Security Symposium. Anaheim, CA, USA: USENIX Association, pp. 5593–5609, (2023). https://www.usenix.org/conference/usenixsecurity23/presentation/deng-gelei
25. Dolan-Gavitt, B., et al.: LAVA: large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 110–121, (2016). https://doi.org/10.1109/SP.2016.15
26. Doupe, A., et al.: Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. en. In: Presented as part of the 21st USENIX Security Symposium (USENIX Security 12). pp. 523–538. (2012)
27. Duchene, F., et al.: KameleonFuzz: evolutionary fuzzing for black-box XSS detection. en. In: Proceedings of the 4th ACM conference on Data and application security and privacy. San Antonio Texas USA: ACM, pp. 37–48, (2014). https://doi.org/10.1145/2557547.2557550. https://dl.acm.org/doi/10.1145/2557547.2557550
28. Duchène, F., et al.: LigRE: Reverse-engineering of control and data flow models for black-box XSS detection. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 252–261, (2013). https://doi.org/10.1109/WCRE.2013.6671300
29. Roy Thomas Fielding: architectural styles and the design of network-based software architectures. University of California, Irvine (2000)
30. Fui-Hoon, Fiona, Nah et al.: Generative AI and ChatGPT: applications, challenges, and AI-human collaboration. en. In: Journal of Information Technology Case and Application Research, pp. 277–304, (2023). https://doi.org/10.1080/15228053.2023.2233814. https://www.tandfonline.com/doi/full/10.1080/15228053.2023.2233814
31. Fung, Adonis P.H.: et al. : Scanning of real-world web applications for parameter tampering vulnerabilities. en. In: Proceedings of the 9th ACM symposium on Information, computer and communications security. Kyoto Japan: ACM, pp. 341–352, (2014). https://doi.org/10.1145/2590296.2590324
32. Fuzzing Competition (C/C++ Programs). (2023). https://sbft23.github.io/tools/fuzzing
33. Gauthier, F., et al.: Experience: Model-Based, Feedback-Driven, Greybox Web Fuzzing with BackREST. en. In: Leibniz International Proceedings in Informatics (LIPIcs). (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.29. https://drops.dagstuhl.de/opus/volltexte/2022/16257
34. Get Started With The OpenAPI Specification. https://swagger.io/solutions/getting-started-with-oas/
35. Giamattei, L.: et al. : Automated grey-box testing of microservice architectures. en. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS). Guangzhou, China: IEEE, pp. 640–650. (2022). https://doi.org/10.1109/QRS57517.2022.00070. https://ieeexplore.ieee.org/document/10062422/
36. Giamattei, L., et al.: Automated functional and robustness testing of microservice architectures. en. In: Journal of Systems and Software, pp. 111857, (2024). https://doi.org/10.1016/j.jss.2023.111857. https://linkinghub.elsevier.com/retrieve/pii/S0164121223002522
37. Godefroid, P.: Fuzzing: hack, art, and science. en. In: Communications of the ACM, pp. 70–76. (2020). https://doi.org/10.1145/3363824. https://dl.acm.org/doi/10.1145/3363824
38. Godefroid, Patrice, Huang, Bo-Yuan, Polishchuk, Marina: Intelligent REST API data fuzzing'. en. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Virtual Event USA: ACM, pp. 725–736, (2020). https://doi.org/10.1145/3368089.3409719. https://dl.acm.org/doi/10.1145/3368089.3409719
39. Golmohammadi, A., Zhang, M., Arcuri, A.: .NET/C# instrumentation for search-based software testing. en. In: Soft-

ware Quality Journal, pp. 1439–1465, (2023). https://doi.org/10.1007/s11219-023-09645-1. https://link.springer.com/10.1007/s11219-023-09645-1
40. Golmohammadi, A., Zhang, M., Arcuri, A.: Testing RESTful APIs: a survey. In: ACM Transactions on Software Engineering and Methodology, pp. 27:1–27:41, (2023). https://doi.org/10.1145/3617175. https://dl.acm.org/doi/10.1145/3617175
41. Haas, A., et al.: Bringing the web up to speed with WebAssembly. en. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. Barcelona Spain: ACM, pp. 185–200, (2017). https://doi.org/10.1145/3062341.3062363. https://dl.acm.org/doi/10.1145/3062341.3062363
42. Hazimeh, A., Herrera, A., Payer, M.: Magma: a ground-truth fuzzing benchmark. en. In: Proceedings of the ACM on Measurement and Analysis of Computing Systems, pp. 1–29, (2020). https://doi.org/10.1145/3428334. https://dl.acm.org/doi/10.1145/3428334
43. Herrera, A., et al.: Seed selection for successful fuzzing. en. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. Virtual Denmark: ACM, pp. 230–243, (2021).https://doi.org/10.1145/3460319.3464795. https://dl.acm.org/doi/10.1145/3460319.3464795
44. Horváth, F., et al.: Code coverage differences of Java bytecode and source code instrumentation tools. en. In: Software Quality Journal, pp. 79–123, (2019). https://doi.org/10.1007/s11219-017-9389-z. http://link.springer.com/10.1007/s11219-017-9389-z
45. Indrasiri, K., Siriwardena, P.: Microservices for the enterprise: designing, developing, and deploying. Apress, (2018). https://books.google.nl/books?id=Qfd5DwAAQBAJ
46. Jin, Brenda, Sahni, Saurabh: and Amir Shevat. Building APIs that developers love. O'Reilly Media Inc, Designing Web APIs (2018)
47. Kim, M., et al.: Enhancing REST API Testing with NLP Techniques. en. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. Seattle WA USA: ACM, pp. 1232–1243, (2023). https://doi.org/10.1145/3597926.3598131. https://dl.acm.org/doi/10.1145/3597926.3598131
48. Klooster, T., et al.: Continuous fuzzing: a study of the effectiveness and scalability of fuzzing in CI/CD pipelines. In: 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT). pp. 25–32, (2023). https://doi.org/10.1109/SBFT59156.2023.00015. https://ieeexplore.ieee.org/document/10190386
49. Korel, B.: Automated software test data generation. en. In: IEEE Transactions on Software Engineering, pp. 870–879, (1990). https://doi.org/10.1109/32.57624. http://ieeexplore.ieee.org/document/57624/
50. Landscape of API Traffic. (2021). https://blog.cloudflare.com/landscape-of-api-traffic
51. Laranjeiro, N., Agnelo, J., Bernardino, J.: A black box tool for robustness testing of REST Services. In: IEEE Access. Conference Name: IEEE Access, pp. 24738–24754, (2021). https://doi.org/10.1109/ACCESS.2021.3056505
52. lcamtuf. Technical whitepaper for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt
53. Lei, Z., et al.: Bootstrapping Automated Testing for RESTful Web Services. en. In: IEEE Transactions on Software Engineering, pp. 1561–1579, (2023). https://doi.org/10.1109/TSE.2022.3182663. https://ieeexplore.ieee.org/document/9796038/
54. Li, J., Zhao, B., Zhang, C.: Fuzzing: a survey. en. In: Cybersecurity, p. 6, (2018). https://doi.org/10.1186/s42400-018-0002-y. https://cybersecurity.springeropen.com/articles/10.1186/s42400-018-0002-y
55. Li, S., et al.: Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. en. In: Information and Software Technology, pp. 106449, (2021). https://doi.org/10.1016/j.infsof.2020.106449. https://linkinghub.elsevier.com/retrieve/pii/S0950584920301993
56. Li, X., Xue, Y.: A survey on server-side approaches to securing web applications. en. In: ACM Computing Surveys, pp. 1–29, (2014). https://doi.org/10.1145/2541315. https://dl.acm.org/doi/10.1145/2541315
57. Liang, H., et al.: Fuzzing: State of the Art. In: IEEE Transactions on Reliability. Conference Name: IEEE Transactions on Reliability, pp. 1199–1218, (2018). https://doi.org/10.1109/TR.2018.2834476
58. libFuzzer. (2016). http://llvm.org/docs/LibFuzzer.html
59. Lin, J., et al.: foREST: A Tree-based Black-box Fuzzing Approach for RESTful APIs. en. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE). Florence, Italy: IEEE, pp. 695–705, (2023). https://doi.org/10.1109/ISSRE59848.2023.00023. https://ieeexplore.ieee.org/document/10301255/
60. Liu, Chien-Hung, Chen, Shu-Ling, Huang, Hong-Kai: Automated Test Input Generation for Testing Representational State Transfer (REST) Application Programming Interface (API) using Parameter Fuzzing. en. In: 2023 IEEE 6th International Conference on Knowledge Innovation and Invention (ICKII). Sapporo, Japan: IEEE, pp. 249–253, (2023). https://doi.org/10.1109/ICKII58656.2023.10332662. https://ieeexplore.ieee.org/document/10332662/
61. Luong, Minh-Thang, Pham, Hieu, Christopher D. Manning: Effective Approaches to Attention-based Neural Machine Translation. (2015). arXiv:1508.04025
62. Lyu, C., Xu, J., Ji, S.: MINER: A Hybrid Data-Driven Approach for REST API Fuzzing. In: Proceedings of the 32nd USENIX Security Symposium. Anaheim, CA, USA: USENIX Association, pp. 4517–4534, (2023). https://www.usenix.org/conference/usenixsecurity23/presentation/lyu
63. Manes, VJM. et al.: The Art, Science, and Engineering of Fuzzing: A Survey. en. In: IEEE Transactions on Software Engineering, pp. 2312–2331, (2021). https://doi.org/10.1109/TSE.2019.2946563. https://ieeexplore.ieee.org/document/8863940/
64. Marculescu, B., Zhang, M., Arcuri, A.: On the Faults Found in REST APIs by Automated Test Generation. en. In: ACM Transactions on Software Engineering and Methodology, pp. 1–43, (2022). https://doi.org/10.1145/3491038. https://dl.acm.org/doi/10.1145/3491038
65. Martin-Lopez, Alberto, Segura, Sergio, Ruiz-Cortés, Antonio: A Catalogue of Inter-parameter Dependencies in RESTful Web APIs". en. In: *Service-Oriented Computing*. Ed. by Sami Yangui et al. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 399–414, (2019). https://doi.org/10.1007/978-3-030-33702-5_31. http://link.springer.com/10.1007/978-3-030-33702-5_31
66. Martin-Lopez, Alberto, Segura, Sergio, Ruiz-Cortés, Antonio.: Online testing of RESTful APIs: promises and challenges. en. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Singapore Singapore: ACM, pp. 408–420, (2022). https://doi.org/10.1145/3540250.3549144. https://dl.acm.org/doi/10.1145/3540250.3549144
67. Martin-Lopez, Alberto, Segura, Sergio, Ruiz-Cortés, Antonio: RESTest: automated black-box testing of RESTful web APIs. en. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. Virtual Denmark: ACM, pp. 682–685. (2021). https://doi.org/10.1145/3460319.3469082. https://dl.acm.org/doi/10.1145/3460319.3469082
68. Martin-Lopez, Alberto, Segura, Sergio, Ruiz-Cortés, Antonio.: RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. en. In: Service-Oriented Computing. Ed. by Eleanna Kafeza et al. Series Title: Lecture Notes in Computer Sci-

ence. Cham: Springer International Publishing, pp. 459–475, (2020). https://doi.org/10.1007/978-3-030-65310-1_33. https://link.springer.com/10.1007/978-3-030-65310-1_33
69. Martin-Lopez, Alberto, Segura, Sergio, Ruiz-Cortés, Antonio: Test coverage criteria for RESTful web APIs. en. In: Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation. Tallinn Estonia: ACM, pp. 15–21, (2019). https://doi.org/10.1145/3340433.3342822. https://dl.acm.org/doi/10.1145/3340433.3342822
70. Martin-Lopez, Alberto et al.: Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. In: IEEE Transactions on Services Computing. Conference Name: IEEE Transactions on Services Computing, pp. 2342–2355, (2022) https://doi.org/10.1109/TSC.2021.3050610
71. Meng, R.,et al.: Large Language Model guided Protocol Fuzzing. en. In: Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS). San Diego, CA, USA. (2024). https://doi.org/10.14722/ndss.2024.24556
72. Miller, Barton P, Fredriksen, Lars, So, Bryan: An empirical study of the reliability of UNIX utilities". In: Communications of the ACM, pp. 32–44, (1990)
73. Mirabella, A.Giuliano et al.; Deep Learning-Based Prediction of Test Input Validity for RESTful APIs. en. In: 2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest). Madrid, Spain: IEEE, pp. 9–16, (2021) https://doi.org/10.1109/DeepTest52559.2021.00008. https://ieeexplore.ieee.org/document/9476896/
74. OWASP API Security Project — OWASP Foundation. https://owasp.org/www-project-api-security/
75. Pan, L.et al.: EDEFuzz: A Web API Fuzzer for Excessive Data Exposures. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24. New York, NY, USA: Association for Computing Machinery, pp. 1–12, (2024). https://doi.org/10.1145/3597503.3608133. https://dl.acm.org/doi/10.1145/3597503.3608133
76. Peng, C., Gao, Y., Yang, P.: Automated Server Testing: an Industrial Experience Report. en. In: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). Limassol, Cyprus: IEEE, pp. 519–522, (2022). https://doi.org/10.1109/ICSME55016.2022.00071. https://ieeexplore.ieee.org/document/9977442/
77. Pezzè, M., Young, M.: Software Testing and analysis: process, principles, and techniques. Wiley India Pvt, Limited (2008)
78. Pham, Van-Thuan, Böhme, Marcel, Roychoudhury, Abhik.: AFLNET: a greybox fuzzer for network protocols. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp. 460–465, (2020). https://doi.org/10.1109/ICST46399.2020.00062
79. Richardson, L., Amundsen, M., Ruby, S.: RESTful Web APIs: services for a changing world. O'Reilly Media, (2013). https://books.google.nl/books?id=ZXDGAAAAQBAJ
80. Orpheas van Rooij et al.: webFuzz: Grey-Box Fuzzing for Web Application. en. In: Computer Security – ESORICS 2021. Ed. byElisa Bertino, Haya Shulman, and MichaelWaidner. Lecture Notes in Computer Science.Cham: Springer International Publishing, pp. 152–172, (2021). https://doi.org/10.1007/978-3-030-88418-5_8
81. Serrano, Nicolás, Hernantes, Josune, Gallardo, Gorka: Mobile Web Apps. In: IEEE Software. Conference Name: IEEE Software, pp. 22–27, (2013). https://doi.org/10.1109/MS.2013.111
82. Trickel, E., et al.: Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities. en. In: 2023 IEEE Symposium on Security and Privacy (SP). 44. SAN FRANCISCO: IEEE Computer Society, pp. 2658–2675, (2023). https://doi.org/10.1109/SP46215.2023.00007
83. Tripp, O., et al.: TAJ: Effective Taint Analysis of Web Applications. en. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. Dublin, Ireland: Association for Computing Machinery, pp. 87–97, (2009). https://doi.org/10.1145/1542476.1542486
84. Tsai, Chung-Hsuan, Tsai, Shi-Chun, Huang, Shih-Kun: REST API Fuzzing by Coverage Level Guided Blackbox Testing. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 291–300, (2021). https://doi.org/10.1109/QRS54544.2021.00040
85. Viglianisi, E., Dallago, M., Ceccato, M.: RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), pp. 142–152, (2020). https://doi.org/10.1109/ICST46399.2020.00024
86. Vulnerability Scanning Tools — OWASP Foundation. en. https://owasp.org/www-community/Vulnerability_Scanning_Tools
87. Wai, Khaing Hsu et al.: Code Modification Problems for Multimedia Use in JavaScript-Based Web Client Programming. en. In: Complex, Intelligent and Software Intensive Systems. Ed. by Leonard Barolli. Series Title: Lecture Notes in Networks and Systems. Cham: Springer International Publishing, pp. 548–556, (2022). https://doi.org/10.1007/978-3-031-08812-4_53. https://link.springer.com/10.1007/978-3-031-08812-4_53
88. Wang, Junjie et al.: Superion: Grammar-Aware Greybox Fuzzing. en. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). Montreal, QC, Canada: IEEE, pp. 724–735, (2019). https://doi.org/10.1109/ICSE.2019.00081 . https://ieeexplore.ieee.org/document/8811923/
89. Wu, H., et al.: Combinatorial testing of RESTful APIs. en. In: Proceedings of the 44th International Conference on Software Engineering. Pittsburgh Pennsylvania: ACM, pp. 426–437, (2022). https://doi.org/10.1145/3510003.3510151. https://dl.acm.org/doi/10.1145/3510003.3510151
90. Yamamoto, K.: Efficient penetration of API sequences to test stateful RESTful services. en. In: 2021 IEEE International Conference on Web Services (ICWS). Chicago, IL, USA: IEEE, pp. 734–740, (2021). https://doi.org/10.1109/ICWS53863.2021.00101. https://ieeexplore.ieee.org/document/9590435/
91. Yandrapally, R., et al.: Carving UI Tests to Generate API Tests and API Specification. en. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Melbourne, Australia: IEEE, pp. 1971–1982, (2023). https://doi.org/10.1109/ICSE48619.2023.00167. https://ieeexplore.ieee.org/document/10172784/
92. Yin, Zijing et al.: "Scanner++: Enhanced Vulnerability Detection of Web Applications with Attack Intent Synchronization". en. In: *ACM Transactions on Software Engineering and Methodology* 32(1)(2023), pp.1–30. ISSN: 1049-331X, 1557-7392. https://doi.org/10.1145/3517036. https://dl.acm.org/doi/10.1145/3517036 (visited on 09/20/2024)
93. Yun, J., et al.: Fuzzing of Embedded Systems: a survey'. en. In: ACM Computing Surveys, pp. 1–33, (2023). https://doi.org/10.1145/3538644. https://dl.acm.org/doi/10.1145/3538644
94. Zalewski, M.: American Fuzzy Lop. (2014). http://lcamtuf.coredump.cx/afl
95. Zhang, H., Dong, W., Jiang, L.: Zokfuzz: Detection of Web Vulnerabilities via Fuzzing. In: 2022 2nd International Conference on Consumer Electronics and Computer Engineering (ICCECE). IEEE, pp. 281–287, (2022). https://doi.org/10.1109/ICCECE54139.2022.9712748
96. Zhang, J., et al.: A novel neural source code representation based on abstract syntax tree. en. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). Montreal, QC, Canada: IEEE, pp. 783–

794, (2019). https://doi.org/10.1109/ICSE.2019.00086. https://ieeexplore.ieee.org/document/8812062/
97. Zhang, M., Arcuri, A.: Adaptive hypermutation for search-based system test generation: a study on REST APIs with EvoMaster. en. In: ACM Transactions on Software Engineering and Methodology, pp. 1–52, (2022). https://doi.org/10.1145/3464940. https://dl.acm.org/doi/10.1145/3464940
98. Zhang, M., Arcuri, A.: Open Problems in Fuzzing RESTful APIs: a comparison of tools. en. In: ACM Transactions on Software Engineering and Methodology, pp. 1–45, (2023). https://doi.org/10.1145/3597205
99. Zhang, M., Belhadi, A., Arcuri, A. JavaScript SBST Heuristics to Enable Effective Fuzzing of NodeJS Web APIs. In: ACM Transactions on Software Engineering and Methodology, pp. 139:1–139:29, (2023). https://doi.org/10.1145/3593801. https://dl.acm.org/doi/10.1145/3593801
100. Zhang, M., Marculescu, B., Arcuri, A.: Resource-based test case generation for RESTful web services. en. In: Proceedings of the Genetic and Evolutionary Computation Conference. Prague Czech Republic: ACM, pp. 1426–1434, (2019). https://doi.org/10.1145/3321707.3321815. https://dl.acm.org/doi/10.1145/3321707.3321815
101. Zhang, M., et al.: White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study. In: ACM Transactions on Software Engineering and Methodology, pp. 122:1–122:38, (2023). https://doi.org/10.1145/3585009
102. Zhao, J et al.: Cefuzz: An Directed Fuzzing Framework for PHP RCE Vulnerability. en. In: Electronics, pp. 758, (2022). https://doi.org/10.3390/electronics11050758. https://www.mdpi.com/2079-9292/11/5/758
103. Zhu, X et al.: Fuzzing: A Survey for Roadmap. en. In: ACM Computing Surveys, pp. 1–36, (2022). https://doi.org/10.1145/3512345