Empirical Comparison of Black-box Test Case Generation Tools for RESTful APIs

Davide Corradini*, Amedeo Zampieri†, Michele Pasqua‡ and Mariano Ceccato§
Department of Computer Science
University of Verona – Verona, Italy
Email: *davide.corradini@univr.it, †amedeo.zampieri@studenti.univr.it, ‡michele.pasqua@univr.it, §mariano.ceccato@univr.it

***Abstract*—In literature, we can find research tools to automatically generate test cases for RESTful APIs, addressing the specificity of this particular programming domain. However, no direct comparison of these tools is available to guide developers in deciding which tool best fits their REST API project.**

In this paper, we present the results of an empirical comparison of automated black-box test case generation approaches for REST APIs. We surveyed the available black-box testing tools that have been proposed in recent literature, finding four usable prototypes: RestTestGen, RESTler, bBOXRT and RESTest. We used these tools to generate test cases for 14 real-world REST services. Then, testing results have been analyzed and compared in terms of robustness (i.e., success rate) and test coverage.

Among the considered tools, RESTler appears to be the most solid, able to successfully test all case studies (the other tools experienced crashes). Conversely, test cases generated by RestTestGen scored the highest coverage, suggesting that its testing strategy is the most effective in testing REST APIs.

***Index Terms*—REST API, Test coverage, Black-box testing, Automated software testing, Experimental comparison**

I. Introduction

RESTful APIs (or REST APIs for short) are the reference architectural style to design and develop Web APIs, using the REpresentational State Transfer paradigm. They are largely adopted to integrate and interoperate information systems, especially when connected to the cloud [1].

Despite testing being a cornerstone practice in software development to reveal implementation defects, manually writing all test cases for a REST API can be tedious, time-consuming and expensive. Hence, automated test case generation emerged as a way to ease and support developers in testing REST APIs.

Automated testing strategies have been proposed and implemented, based on different test case generation algorithms and conceived with different designs to serve different testing purposes. Test interactions can be composed, for instance, based on data dependencies among operations, or applying heuristics to elaborate promising request sequences. Moreover, several input data generation techniques have been proposed: either based on documented examples, reusing previously observed values, choosing values from dictionaries, applying mutations, or exploiting constraints among input parameters.

Despite several of them being available, to the best of our knowledge, no guideline is available to help developers in making an informed decision on which tool is more suitable to automatically test a REST API. Indeed, the different testing algorithms implemented by the tools have been designed with different fault model in mind, to expose possibly different faults (e.g., data integrity Vs security issues). So, different tools might come with different and incompatible oracles to reveal defects. Thus, a direct comparison on bug detection might be unfair.

Considering that automated REST API testing tools typically adopt a black-box approach and base the test case generation process on API interface specifications, an alternative and more fair comparison perspective could be with respect to the interface itself, i.e., what extent of the API interface is exercised when running the automatically generated test cases.

This paper defines and adopts a formal framework to empirically compare tools meant to automatically generate test cases for REST APIs. The framework (fully available in our replication package [2]) contains 14 open source REST API case studies, available as Docker images to facilitate deployment and replication.

Four state-of-the-art testing tools have been identified by surveying the relevant literature: RestTestGen, RESTler, bBOXRT and RESTest. These tools have been deployed to test the REST APIs case studies under the same conditions, i.e., each tool was run with same time budget and the database of each case study was reset before starting each testing session. Test cases are compared using eight coverage metrics adopting a black-box viewpoint (proposed by Martin-Lopez et al. [3]).

Experimental results underline that available tools come at different levels of reliability, causing less mature implementations (e.g., research prototypes) to crash on some case studies. Indeed, RESTler shown to be the most solid tool, since it is the only one applicable to all the case studies. Conversely, RestTestGen and bBOXRT could work, respectively, on 14 and 8 case studies. RESTest, as a research prototype, could be successfully applied only to 2 case studies.

Moreover, results point out that different testing strategies allow maximizing different coverage metrics. In fact, while bBOXRT achieved the highest coverage of parameter values, RestTestGen achieved the highest coverage on almost all the other metrics, including coverage of paths, parameters, request/response content-types and status codes.

The rest of the paper is organized as follows. Section II covers the background on REST APIs, while Section III describes the four state-of-the-art tools for testing REST APIs we employed. In Section IV, we describe the theoretical measurement framework that we adopted to assess the cov-

erage of the REST APIs testing tools. In Section V, we describe the experimental setting used to compare the testing tools taken into account. In Section VI, we have the results of the comparison. Finally, after discussing related work in Section VII, Section VIII closes the paper.

II. Background

A. RESTful APIs

A RESTful API (or REST API for short) is an API that respects the REST (REpresentational State Transfer) architectural style [4]. REST APIs provide a uniform interface to create, read, update and delete (CRUD) a resource. A resource is generally identified by an HTTP URI, and CRUD operations are usually mapped to the HTTP methods POST, GET, PUT and DELETE.

For example, consider a REST API *PetStore* managing a collection of pets. A possible HTTP URI pointing to the resource could be `/pets`. In this case, the HTTP operation `GET pets` is used to retrieve the list of pets and `POST /pets` could be used to add a new pet to the collection.

The API may accept input parameters to specify additional information for executing operations, such as the identifier of the object to retrieve (e.g., `/pets/{petId}`) or a structured object to be added to the collection using the POST method.

B. The OpenAPI Specification

OpenAPI[1] defines a standard to document REST APIs. According to OpenAPI, an API service is described using a structured file (either YAML or JSON) that specifies how to reach the API using a URI, which authentication schema is adopted and the details of all the operations available in the API: the input parameters (and their schema) to be used in requests and the schema of responses.

After an initial header that specifies versions, licenses and the base URL of the API, an OpenAPI specification contains an array of *paths*, namely the list of URL paths available in the API. In the *PetStore* example we have two paths: `/pets` and `/pets/{petId}`. Each path supports one or more HTTP methods. Operations in the API are pairs of paths and methods, and usually are identified by an *operation ID*. For instance, the method GET in `/pets` (`getPets`) is used to retrieve the list of all the pets, while the method GET in `/pets/{petId}` refers to the operation `getPetById`, meant to retrieve the `Pet` object that matches a specific `petId`. Path parameters are specified directly in the path URL using curly braces, such as the `petId` input parameter in the previous example.

Request input and output are associated with a *schema* that specifies their type and, optionally, a set of constraints on values (e.g., a *min* or *max* value for numeric parameters). Types can be atomic (e.g., integers and strings) or structured (i.e., compound objects). For instance, the parameter `petId` of `/pets/{petId}` could be of type `string`, while the response to the corresponding GET operation is expected to be in JSON, according to the `Pet` schema (also defined in the specification). A specification not only describes the response format in nominal cases (e.g., a response status code `200`), but it also describes the format expected when errors occur.

[1]https://www.openapis.org/

TABLE I
Object tools strategies.

Tool	Operations sequence(s)	Input values
RestTestGen [5]	Data dependencies	Random Documented examples Observed values Dictionary Mutations (3)
RESTler [6]	Data dependencies Full enumeration of sequences	Observed values Dictionary
bBOXRT [7]	(not reported)	Random Mutations (57)
RESTest [8]	(not reported)	Random* Documented examples* Dictionary* Mutations* (4)

∗ exploiting inter-parameter dependencies

III. Object Tools

In the last years, the research community proposed several approaches to automatic generation of test cases for REST APIs. We surveyed the literature, selecting the available state-of-the-art tools. In particular, we carefully checked the works published in the top testing and software engineering conferences[2] (together with their satellite workshops) and journals[3] appeared in the last 4 years. Furthermore, we also used publications search engines (like *IEEE Xplore*[4]) with the following keywords: "REST", "RESTful API" and "black-box testing". In our search, we looked for approaches complying with the following requirements: (i) approaches operating with a black-box perspective; (ii) approaches implemented into a software tool; (iii) approaches with an open-source implementation or with publicly available binaries.

Eventually, we obtained our final set consisting of four tools: RestTestGen, RESTler, bBOXRT, and RESTest. All the selected tools have a common characteristic: they use the information documented inside an OpenAPI specification to build a testing strategy according to their own algorithm. Approaches differ in terms of operation sequence(s) assembly and input values generation. Table I summarizes the strategies of the four object tools.

In the following, a brief overview of the peculiarities of the selected tools.

A. RestTestGen

RestTestGen is an automated black-box test case generation tool for REST APIs proposed by Viglianisi et. al [5]. It is Java-based and the executable JARs are available on GitHub[5].

[2]ICSE, ESEC/FSE, CCS, ISSTA, ICST, AST, EDOC, ICICT and ICCSDET.
[3]TOSEM, TSE, EMSE and TOSC.
[4]https://ieeexplore.ieee.org
[5]https://github.com/resttestgenicst2020/submission_icst2020

The strategy of RestTestGen is based on the *Operation Dependency Graph* (ODG), a graph which encodes data dependencies among the operations available in the API. For instance, the operation `GET /pets/{petId}` (from the example in Section II) depends on the operation `GET /pets`, that returns a list of valid pet IDs, because the output of the latter can be used as the input for the former. Dependencies in the ODG are inferred from the OpenAPI specification by matching parameters names and schemas. The ODG is used to optimize the operations testing order, prioritizing operations with satisfied data dependencies. It is updated at runtime according to the results of executed tests.

RestTestGen is composed by two modules, the *Nominal Tester* and the *Error Tester*, that are in charge of generating nominal and error test cases, respectively.

Nominal test cases are meant to test nominal interactions with the API: they are generated to comply with the interface documented in the OpenAPI specification. Previously observed values, if available, are re-used as input values. Alternatively, RestTestGen uses a dictionary, examples or random values. An oracle, based on the status code of responses, evaluates the outcome of each test case execution: `2XX` and `4XX` status codes are classified as successful executions (indeed, those status codes represent correct executions or graceful errors in the HTTP protocol); `5XX` status codes are instead classified as failures (server errors in the HTTP protocol).

Error test cases are generated by mutating successful nominal test cases. RestTestGen applies three different mutations: *missing required*, by removing parameters that are documented as mandatory; *wrong input type*, by modifying types of input parameters; and *constraint violation*, by setting unsupported parameter values according to the documented constraints. An oracle, based on the status code of responses, evaluates test cases executions: `4XX` status codes are classified as successful, meaning that the server correctly identified the malformed input; `2XX` status codes are classified as failures, because the server accepted a malformed input as valid; and `5XX` status codes are once again classified as failures, meaning that the malformed input caused a server-side error.

Both nominal and error test cases are also evaluated by a further oracle, which validates the response schema documented in the OpenAPI specification against the one obtained in the test cases responses. A test case passes if the response matches its schema definition.

B. RESTler

RESTler is a stateful REST API fuzzer presented by Atlidakis et al. [6] at Microsoft Research, written in Python and available on GitHub[6].

RESTler generates stateful sequences of requests by inferring producer-consumer relations between request types described in the specification. It also dynamically analyzes responses to intelligently build request sequences and avoiding sequences leading to errors.

RESTler relies on different test generation algorithms, and each one implements a different test space search logic. The *BFS* algorithm appends every possible compatible request to every existing sequence, namely performing an exhaustive search. In the *BFS-Fast* algorithm, every request is appended to at most one existing sequence. This results in a smaller set than the full BFS approach, but it does not guarantee that every possible request sequence is generated. *BFS-Cheap* [9] implements the dual trade-off of BFS-Fast: all the sequences are generated but only at most two sets of parameters values (one valid, one invalid) are allowed for each request. Finally, the *Random-walk* algorithm randomly selects a valid request sequence to which append a random request.

To fuzz input values, RESTler relies on a user-configurable dictionary. The user can manually extend this dictionary with custom values that better fit the service, or that are known to be more effective in the testing phase. When RESTler detects data dependencies among operations, also previously observed values are used as parameters. For error detection, RESTler uses HTTP status codes: if a status belonging to the `5XX` class is detected, then the test sequence could have discovered a bug, so it is logged for further analysis.

C. bBOXRT

bBOXRT is a black-box robustness testing tool for RESTful APIs proposed by Laranjeiro et al. [7], written in Java and available on the authors' website[7]. The aim of bBOXRT is to assess the robustness of REST APIs observing the behavior of services under test when providing invalid requests.

The peculiarity of bBOXRT is the large number of supported mutations. The provided fault model consists of 57 different mutations applicable to input parameters of various types (numbers, strings, booleans, dates, times, arrays, etc.).

The bBOXRT execution starts with the analysis of the OpenAPI specification to collect information about the service under test. Subsequently, the *Workload generator* component starts generating and executing valid requests with the aim to understand the behavior of the service under test in absence of faulty workloads. Parameter values are randomly generated to comply with the specification. Requests triggering a `2XX` response are stored for future use, while interactions that triggered `4XX` and `5XX` status codes are either retried with different values or discarded. Authors do not explain the strategy they adopted to order operations.

The next component, the *Faultload generator*, creates faulty requests by mutating successful requests. It applies mutation rules to parameters, one at a time. Faulty interactions are stored for further analysis by test engineers. According to the authors, bBOXRT does not fully automate the analysis of the test cases outcome, so manual intervention is still required.

D. RESTest

RESTest is an automated black-box testing tool for RESTful APIs proposed by Martin-Lopez et al. [8]. It is written in Java

[6]https://github.com/microsoft/restler-fuzzer (cloned on Dec. 27th, 2020).

[7]https://eden.dei.uc.pt/~cnl/papers/2020-access.zip

and the source code is available on a GitHub repository[8].

The peculiarity of this tool is the *inter-parameter dependencies* support. Some REST APIs, in fact, impose constraints that restrict not only input values, but also the way in which input values can be combined to fill valid requests. For example, the YouTube API search operation requires the `publishedAfter` parameter to be greater or equal to `publishedBefore`. Currently, the OpenAPI grammar does not support a formal documentation of such dependencies, so Martin-Lopez et al. [10], [11] proposed a domain-specific language, called IDL (inter-parameter dependency language), to this aim. RESTest uses the documented dependencies to code a constraint satisfaction problem and deploys a reasoner to generate test cases accordingly. Since, at the moment, OpenAPI specifications do not support inter-parameter dependencies, no REST API comes with a dependencies-enriched specification. For this reason, the IDL module of RESTest was not active in our experiment.

RESTest can generate both nominal and faulty test cases using two strategies: (i) random testing (RT), by generating random input values; and (ii) constraint-based testing (CBT), by exploiting constraints of inter-parameter dependencies. Test cases are generated in both settings from a *test model* derived from the OpenAPI specification. Nominal test cases aim to stress the service with valid inputs to check its behavior against the specification. Faulty test cases derive from nominal test cases by applying mutations (excluding mandatory parameters, using out-of-range values, and violating the JSON schema). Additionally, RESTest can generate faulty test cases by violating inter-parameter dependencies. Authors do not explain the strategy adopted to sort operations during testing.

To classify test outcomes, 5 oracles are deployed: (i) status code must be lower than `500`; (ii) a response must conform the documented schema; (iii) if the request violates one or more parameter specifications, the status code must be different from `2XX`; (iv) if a request violates inter-parameter dependencies, the status code must be different from `2XX`; (v) if the request is valid according to the specification and inter-parameter dependencies are met, the status code must be different from `4XX`.

E. Other discarded tools

Initially, we considered other tools than RestTestGen, RESTler, bBOXRT, and RESTest, but, for various reasons, they have been excluded from the object tools list. QuickREST, proposed by Karlsson et al. [12], is a proof-of-concept tool that has not been released as a generic tool. It is, instead, available only as a customized build to work on the case studies of a replication package, in a package-specific version. Hence, it is not possible to apply QuickREST to test APIs other than those from their replication package. Another discarded tool was proposed by Ed-Douibi et al. [13]. In this case, the proof-of-concept tool was developed as plug-in for *Eclipse*. This tool was discarded due to the presence of errors in the source code that prevented it from installing properly. Finally, EvoMaster [14] has been excluded because, at the time of writing, it does not follow a black-box approach: it requires the availability of the Java source code to perform static and dynamic analysis.

[8]https://github.com/isa-group/RESTest/ (cloned on Dec. 27th, 2020).

IV. Test Coverage Metrics

When it comes to comparing REST API testing tools, there is not a standard fault model adopted by the state-of-the-art approaches, which might come with different and incompatible oracles to reveal defects. In this scenario, a direct comparison on bug detection might be unfair. Hence, to objectively compare automated test case generation tools, we need a methodology to measure the *coverage* of their test cases.

The four black-box tools that we are comparing assume the source code of REST APIs is not available. So, source code coverage cannot be the metric for the comparison. An alternative approach is represented by *interface coverage*, which measures the testing coverage with respect to the specification of the REST API rather than to its actual code. This is also motivated by the fact that the REST API testing tools base the test case generation process on the OpenAPI specification of the service under test.

In this respect, Martin-Lopez et al. [3] proposed a test coverage framework based on the API interface description available within the OpenAPI specification. They introduced ten *coverage metrics* to measure the coverage of a test suite as the ratio of the tested elements on to the total number of elements available in the API.

Although the available measurement framework provides a starting point for an empirical comparison, adaptations are required to turn metrics operative. Indeed, during the empirical adoption of the framework, we realized that some metric definitions were too abstract to be properly applied in practice.

In the following, we present an overview of the metrics proposed by Martin-Lopez et al. [3], along with our adaptations. They are six metrics related to the generated inputs, and four metrics related to the triggered outputs.

A. Input coverage metrics

Six metrics, called *input coverage metrics*, are meant to measure the capabilities of a test suite *requests* to exercise different parts of the REST API under test.

1) Path coverage: it measures the capability of a test suite to exercise the API paths. It is the ratio of the number of tested paths to the total number of paths documented in the OpenAPI specification. A test suite reaches 100% path coverage if its tests send at least one request directed to each path of the API.

2) Operation coverage: it measures the capability of a test suite to execute the available operations. It is the ratio of the number of tested operations to the total number of operations described in the OpenAPI specification. A test suite reaches 100% operation coverage if there exists at least one request directed to each path with all the documented HTTP methods.

3) Parameter coverage: it measures the capability of a test suite to sample all the available parameters on operations. It is the ratio of the number of input parameters used by test cases to the total number of parameters documented in the OpenAPI specification. A test suite reaches 100% parameter coverage if all input parameters of all operations are included in requests at least once.

4) Parameter value coverage: it measures the capability of a test suite to choose meaningful values for input parameters. It is the ratio of the number of the exercised parameter values to the total number of possible values that parameters can assume according to the OpenAPI specification. This metric only applies to domain-limited parameters, such as boolean and enum types. A test suite reaches 100% parameter value coverage if requests contain all the possible values for each parameter of each operation.

5) Request content-type coverage: it measures the capability of a test suite to feed endpoints with request bodies of different content-type formats. It is the ratio of the number of tested content-types to the total number of accepted content-types as documented in the OpenAPI specification. A test suite reaches 100% request content-type coverage if there exists at least a test request for each accepted content-type. The original definition by Martin-Lopez et al. [3] does not consider scenarios of content-types with wildcards (e.g., `application/*`). Such cases turn the number of accepted content-types unbounded. In our adaptation, we assume that request content-type coverage can be computed only when operations content-types have no wildcards, in order for the metric value to be meaningful.

6) Operation flow coverage: it measures the capability of a test suite to apply different sequences of operations. It is defined as the ratio of the number of tested flows to the total number of meaningful flows, according to the application business logic. However, as also acknowledged by Martin-Lopez at al. [3], there is no standard definition of what are the meaningful flows for a REST API, and there is no way to document flows in the OpenAPI specification. Thus, considering that the definition of this metric is not operative, we decided not to include it in our framework.

B. Output coverage metrics

Four metrics are meant to measure the coverage of a test suite according to *responses* received from the REST API under test. These metrics are called *output coverage metrics*.

1) Status code class coverage: it measures the capability of a test suite to trigger responses with *correct* and *erroneous* status code classes. The OpenAPI specification does not provide primitives to formally document correct or erroneous status code classes. According to the metric definition by Martin-Lopez et al. [3], it is up to the test engineer to define which status codes belong to the *correct* class, and those belonging to the *erroneous* class, based on the semantic of the target API. To maintain a black-box point of view that assumes no knowledge about the semantic of the API under test, we consider the standard semantic provided by the HTTP protocol, i.e., the `2XX` class represents a correct execution and `4XX` and `5XX` classes represent an erroneous execution. A test suite reaches 100% status code class coverage when it is able to trigger both correct and erroneous status codes. Conversely, if it only triggers status codes belonging to the same class (either correct or erroneous), the reached coverage is 50%.

2) Status code coverage: it measures the capability of a test suite to trigger responses with different status codes. It is the ratio of the number of obtained status codes to the total number of status codes documented in the OpenAPI specification, for each operation. A test suite reaches 100% status code coverage if, for each operation, it is able to test all the status codes.

3) Response body properties coverage: it measures the capability of a test suite to trigger responses containing all the properties defined in their schema. A property is, for instance, a key-value pair of a JSON object. This metric is computed as the ratio of the number of obtained properties to the total number of properties defined in the OpenAPI specification schemas. A test suite reaches 100% response body properties coverage if it is able to trigger responses whose bodies contain all properties for all response objects. Parsing the response header is not enough to compute this metric and, in addition, the response bodies have to be parsed with different grammars according to the body content-type (e.g., JSON or XML). For this reason, we decided to skip this metric in this work, and focus on the other metrics, whose computation is less complex. However, we plan to implement also this metric as future work.

4) Response content-type coverage: it measures the capability of a test suite to trigger responses whose body covers different formats. It is the ratio of the number of obtained content-types to the total number of response content-types documented in the OpenAPI specification. A test suite reaches 100% response content-type coverage if there exists at least one test response whose body matches each documented content-type, for each operation. Similarly to the request content-type coverage metric, we will compute this metric only when specific content-types are defined with no wildcard.

C. Automatic metrics computation

To automatically compute the coverage metrics achieved by the object tools, we have developed Restats [15], a Python tool that implements the aforementioned measurement framework. Our implementation is tool-agnostic: it does not employ tool-specific log files to compute the coverage. Quite the opposite, it operates by reading a generic HTTP traffic log, composed by request-response pairs. In our validation, to log requests and responses we have routed all the HTTP traffic through a proxy before executing the object tools.

Restats reads the HTTP log and the OpenAPI specification of the target API, then it computes path, operation, parameter, parameter value, and status code coverage as originally defined by Martin-Lopez et al. [3]. It computes input and output content-type coverage, and status code class coverage according to our adaptation, as explained in the previous paragraphs.

V. Experimental Settings

In this section, we provide an experimental evaluation of the performance of the state-of-the-art testing tools for REST APIs. In particular, our aim is to assess the capability of a tool to test real-world case studies, and to measure how effective are the generated tests suites. The complete package to replicate our experiment is available online [2].

A. Research Questions

Automated black-box RESTful APIs testing approaches are available as research prototypes and, thus, might not be as robust as commercial tools. They might fail or crash with certain API implementations. Before deciding which tool to adopt, a developer might be interested in knowing their maturity and solidity, so the first research question is meant to compare tools with respect to their ability to manage many real-world case studies.

RQ1: How *robust* are automated RESTful APIs test-case generation tools?

The extent of a REST API that can be tested by automated tools is the other important consideration when deciding which testing tool to adopt. So, the second research question is intended to compare tools with respect to the coverage that their test cases can achieve.

RQ2: What is the *coverage* of the test suites emitted by automated RESTful APIs test-case generation tools?

Our empirical investigation will be designed to answer these research questions.

B. Metrics

Our empirical evaluation of the performance of REST APIs testing tools is based on two different dimensions. First, we consider *robustness*, aiming at assessing to which extent a tool is ready to be effectively usable. This translates to checking how many APIs a tool is able to test out-of-the-box without unexpected errors.

Second, we consider *coverage*, aiming at assessing the adequacy of the test cases generated by the tools. Considering that all the tools start from the definition of the API interface (input, output and operations) and require no source code access, coverage will be computed with the same viewpoint. In particular, interface coverage means how much of the behavior of an API, as described in the specification, is tested by the tools. Coverage will be computed using the coverage metrics introduced in Section IV.

C. REST APIs Case Studies

For the comparison to be fair, object tools should operate on the same REST APIs, with the same initial conditions. Many publicly hosted APIs are available for free (such as those on *APIs.guru*[9]) and they have been used as case studies for assessing automated testing tools [5]. However, they are not appropriate for a direct comparison among several tools, because the state of these APIs can be changed by previous executions of testing tools or by other users accessing them. Hence, different testing tools might work with APIs at different starting state and this might affect the tool performance and, consequently, threaten the validity of our results.

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

TABLE II
List of the selected case studies.

Case Study	Language	Framework	Endpoints	Operations	# of lines
01-Slim	PHP	Slim	9	18	8,566
02-Airline	Java	Spring Boot	12	30	3,859
03-Streaming	Java	Spring Boot	5	5	1,780
04-Petclinic	Java	Spring Boot	17	47	8,550
05-Toggle	ASP.NET	.NET Core	8	16	2,363
06-Problems	Java	Spring Boot	5	9	2,174
07-Products	Java	Spring Boot	6	14	3,451
08-Widgets	Go	-	4	14	1,370
09-Safrs	Python	Flask	6	18	2,787
10-Realworld	PHP	Laravel	11	19	5,278
11-Crud	Node.js	Express	1	4	5,106
12-Order	PHP	Laravel	2	3	3,359
13-Users	TypeScript	Express	2	5	805
14-Scheduler	Node.js	Express	26	40	24,044

To overcome this state interference problem, we opted for case studies that we can download and run in a controlled local environment. To this aim, we searched for REST API implementations among the open-source projects on *GitHub*. With local instances of REST APIs, we can set a common starting point for the underlying database and restore a common initial state of the API before starting each testing iteration.

We started our search with the query strings "REST", "RESTful API", "OpenAPI" and "Swagger", to have an initial list of candidate case studies. Subsequently, we also added query strings that represent framework commonly used to implement REST APIs, such as "swagger-ui", "SpringFox", "swagger-jsdoc", "flask-swagger". Among these APIs, we kept those containing an OpenAPI specification, because black-box testing tools require it as input. Some services contain this specification directly in the project sources; for some others, the specification is not in the source code, but it can be automatically generated when their underlying frameworks support this feature. This is the case for some services implemented, for instance, using *Spring* [16] or *Flask* [17].

These potential case studies have been downloaded, compiled and run to discard those that failed either in compiling or in running. After this last filtering, our final set of case studies consists of 14 REST APIs, for a total of 114 endpoints and 234 operations (more information in Table II). We consider these APIs as representative of real-world REST APIs because: (i) they are written in different programming languages (PHP, Java, Go, ASP.NET, Python, JavaScript and TypeScript); (ii) they are based on different frameworks and DBMSs; and (iii) they have different levels of complexity in terms of number of operations and dependencies. Applications are mostly query-intensive: their goal is to manage, for instance, an airline, restaurant orders, users, a library, a pet clinic, etc. Some APIs have many dependencies among operations (e.g., the airline management system with airports, planes, flights and routes), while others are simpler.

Among the 14 selected working case studies, 7 of them contained small errors in the specification, resulting invalid according to the official *Swagger Editor* [18]. Considering that all testing tools expect a valid specification, we manually

fixed these errors, paying attention not to alter the intended semantic. Indeed, we applied only minor changes for evident mistakes trivial to solve, such as removing non RFC3986-compliant characters from URLs, changing wrong syntax when OpenAPI version 2 (Swagger) syntax was used in OpenAPI version 3 files, moving fields in the right position when they were misplaced, and renaming operations with reused names when they were supposed to adopt unique naming.

D. Experimental Procedure

In order for the case studies to be testable, it might be necessary to initialize their state with some data. For instance, a "delete item" feature can be tested only when the "item" data does exist. Most of the case studies already came with a pre-initialized database, or with a procedure that fills it after installation. The initial database was completely empty only on few cases, so we adopted the following procedure to fill it. We manually interacted with these APIs, executing each documented feature at least once, thus providing some data. After the API has been moved to a testable state, we took a snapshot of the database, representing the initial state to be set when cleaning side effects produced during testing.

Note that, very few REST APIs provide a sandbox for testing purposes and, even if provided, a sandbox usually does not come with a full-reset mechanism. For this reason, we have created a custom sandbox for each REST API case study, encapsulated into an independent *Docker* container. In this way, we could isolate every service environment, making each test independent of the others. Furthermore, the use of containers allows us to easily restore the same starting point before running each testing tool.

Each testing tool has been configured with its default settings or, when available, with the settings that their authors deem the most effective in the corresponding paper. In particular, for RestTestGen and bBOXRT we used the default settings; for RESTler we set *BFS-Cheap* as test generation algorithm because showed as the best performer with reduced time budgets; for RESTest we used the CBT generator, although without providing any inter-parameter dependency constraint. Testing tools have been run on each case study with a time budget of 10 minutes. After each run, the case study database has been reset, to start each testing session from a clean baseline state. Test case generation has been repeated 10 times for each case study to control random variation of non-deterministic algorithms integrated in testing strategies. The execution log has been captured by a proxy and coverage metrics have been computed by Restats (see Section IV-C).

The experiment has been conducted on an Ubuntu 20.04 desktop computer equipped with an AMD® FX™-6300 six core CPU running at 3.5GHz and 16GB of primary memory.

E. Threats to Validity

We identified the following limitations as potential threats to the validity of our empirical results.

Conclusion validity. In order to draw correct conclusions, the measurements must be reliable. To limit this threat, we adopted an existing measurement framework, originally proposed by Martin-Lopez et al. [3], with only minimal changes to turn it operative.

Internal validity. To limit external factors that might influence our observations, case studies have been run locally, so that only testing tools could access them. No other end-users could access the case studies during the experiment and alter their state. Moreover, to give all the testing tools the same starting conditions, case studies databases have been reset before each testing session, thus canceling the footprint of previous executions. To make sure that measurements did not influence the testing results, testing tools have not been instrumented. Instead, coverage metrics have been computed by an external measurement tool, that just monitors the network traffic between case studies and testing tools.

Despite each testing tool reporting some kind of coverage statistics, there could be differences among the way these statistics are collected on different tools. To compare consistent data, coverage reported by testing tools have been ignored and coverage has been computed by a measurement tool contributed by us.

Construct validity. Considering that testing tools contain non-deterministic components, by chance rare events may have influenced our results. To limit this threat, we measured 10 independent runs and the average coverage has been reported.

External validity. Although we have sampled real REST APIs in our experimental validation, written in different programming languages and with different frameworks, we cannot assume that our results hold for any other arbitrary REST API. Additional experiments on new case studies are needed to corroborate our findings.

VI. Experimental Results

This section presents the results of our experimental validation to compare REST APIs testing tools according to robustness and coverage, respectively.

A. RQ1: Analysis of Robustness

All the tools have been run on the same 14 case studies, monitoring crashes and failures. We tried our best to make tools work on all case studies, sometimes even contacting the authors in order to understand the possible reasons for the failures. Table III reports successful executions with tick-marks (✓) and failing executions with cross-marks (✗). Eventually, in the last line, the table reports the total number of successfully tested case studies for each automated testing tool.

RESTler resulted the most robust automated testing tool because it is the only one able to manage all case studies. Indeed, all other tools fail (or crash) while testing some services.

RestTestGen was the second most robust tool, as it could run on 11 case studies out of 14. During the testing of case study `02`, the tool got stuck in an endless loop while parsing a date. In case studies `03` and `14`, instead, the tool crashed in its initialization phase. RestTestGen makes use of the official *Swagger Codegen* [19] to build HTTP client classes starting

TABLE III
ROBUSTNESS: CASE STUDIES SUCCESSFULLY TESTED BY EACH TOOL.

Case study	RestTestGen	RESTler	bBOXRT	RESTest
01-Slim	✓	✓	✗	✓
02-Airline	✗	✓	✗	✗
03-Streaming	✗	✓	✗	✗
04-Petclinic	✓	✓	✗	✗
05-Toggle	✓	✓	✓	✗
06-Problems	✓	✓	✗	✗
07-Products	✓	✓	✓	✗
08-Widgets	✓	✓	✓	✓
09-Safrs	✓	✓	✓	✗
10-Realworld	✓	✓	✓	✗
11-Crud	✓	✓	✓	✗
12-Order	✓	✓	✓	✗
13-Users	✓	✓	✓	✗
14-Scheduler	✗	✓	✗	✗
Total:	**11**	**14**	**8**	**2**

from the OpenAPI specification, and the failure is due to this module which is executed at the very beginning.

bBOXRT could run on approximately half of the APIs (8 out of 14). In case study 01, the tool crash is caused by an unhandled Java Null Pointer Exception of the component responsible to write the Excel output file. In the other 5 non-working cases, bBOXRT crashes while parsing the OpenAPI specification, especially when resolving the defined schemas.

RESTest seems to be the least robust testing tool because it failed on most of the case studies. Indeed it could only test two of them: 01 and 08. The main limitation of the tool is its inability to test REST APIs that use body parameters (e.g., a JSON data structure in the body) when no body parameter examples are provided within the specification. Nevertheless, this is a quite common scenario in practice: in fact, this happens for 11 out of 14 case studies. In one other case (case study 09), the failure is due to malformed requests containing two *content-type* fields in the header. When building requests, RESTest sets the default content-type field in the header. A second content-type field is then appended, in case the specification explicitly documents it. However, the component in RESTest responsible for checking the request correctness detects the duplicate content-type field in the header and stops the program execution with an error message. Basically, the tool rejects the request generated by the tool itself.

Based on these results, we can answer RQ1 as follows:

RESTler is the most robust automated testing tool for REST APIs, because it could test all the 14 case studies. RestTestGen is the second most robust tool (11/14), followed by bBOXRT (8/14). RESTest is the least robust automated testing tool (2/14).

B. RQ2: Analysis of Coverage

We now compare the automated testing tools with respect to the coverage metrics. We have excluded RESTest from the comparison due to its very low robustness. Indeed, restricting the comparison on only the two case studies testable by RESTest would have made the coverage analysis meaningless. Therefore, in order for the comparison to be as fair as possible, we considered only the case studies that all the remaining tools (RestTestGen, RESTler and bBOXRT) could test successfully. Thus, the coverage comparison focuses on 8 case studies, namely 05 and from 07 to 13.

TABLE IV
COVERAGE: NUMBER OF "WINS" FOR THE TOOLS ON EACH METRIC.

Coverage metric	RestTestGen	RESTler	bBOXRT	Draw
Path	1	0	0	7
Operation	1	3	0	4
Parameter	1	0	0	4
Parameter value	1	0	2	0
Req. content-type	2	1	0	4
Status code class	4	4	0	0
Status code	5	3	0	0
Resp. content-type	3	2	0	2

Figure 1 shows the experimental data distributed over the 8 selected case studies, with a different box-plot for each metric. For instance, in the second box-plot of the first row we can compare values of the *Operation* coverage metric. While RestTestGen and RESTler score very high and similar values of Operation coverage, bBOXRT records lower values.

Overall, RestTestGen seems superior to the other tools with respect to *Parameter*, *Request content-type*, *Response content-type*, *Status code class* and *Status code* coverage metrics. RestTestGen and RESTler achieve similar values of *Path* coverage (they have the same median). Eventually, bBOXRT overcomes the other testing tools on *Parameter value* coverage. RESTler does not perform better than the other tools with respect to any coverage metric.

After these qualitative considerations on graph trends, we mean to present more quantitative comparison results. We recorded all the coverage metrics for all testing tools on all case studies, and we take the average value on 10 runs in order to avoid bias due to the nondeterministic components of the tools. We say that a testing tool hits a "win" for a coverage metric on a case study if the testing tool scores the highest value for that metric when testing that case study.

Table IV reports the number of "wins" for each testing tool on each coverage metric. For instance, the second line shows the results for the *Operation* coverage metric. We can observe that RestTestGen reported higher Operation coverage than RESTler and bBOXRT on 1 case study. Instead, RESTler reported the highest value of Operation coverage on 3 case studies. bBOXRT never reported a higher value for this metric. For the remaining 4 case studies no testing tool is a clear winner, because at least two other tools reported an equally high value (*Draw* column). Thus, we can claim that RESTler is preferable when considering Operation coverage (the corresponding number of "win" is highlighted in the table).

Not all rows of Table IV add up to 8, since a metric may be not computable for a specific case study. For instance, *Request content-type* coverage can be computed only when operations content-types have no wildcard (see Section IV for details).

According to these data, RestTestGen is preferable for *Path*, *Parameter*, *Request content-type Status code* and *Response*

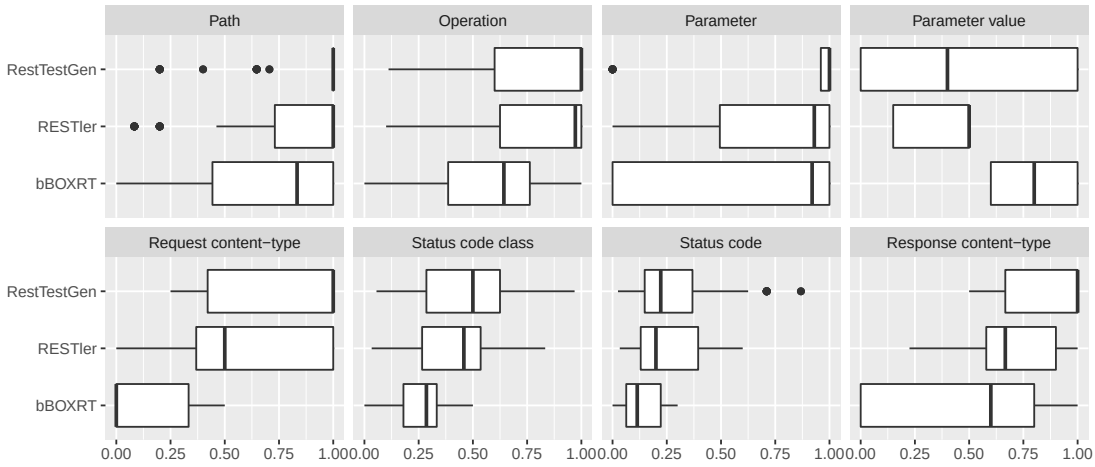

Fig. 1. Box-plots of coverage metrics on the 8 selected case studies.

content-type coverage. RESTler recorded the best results for *Operation* coverage, while bBOXRT for *Parameter value* coverage. RestTestGen and RESTler achieved equally high results (4 case studies) for *Status code class* coverage.

With these results, we can answer RQ2 as follows:

> *RestTestGen is the automated testing tool producing test suites with higher coverage, because when generating test cases for 8 case studies it overtakes the other testing tools on 5 coverage metrics, while RESTler and bBOXRT have been superior to the other tools according to 1 coverage metric each.*

C. Considerations

Based on the experimental results, we could formulate the following subjective considerations.

Research prototypes robustness. The four tools under analysis are all research prototypes, so, it is not surprising that they may fail in testing some real-world case studies. Nevertheless, RESTler is the most mature tool, able to test without errors all the considered APIs. The different robustness degrees among the tools poses a first obstacle to the comparison of coverages. Indeed, while we were able to compare RESTler, RestTestGen and bBOXRT on a sufficiently large set of case studies, a comparison between all four tools would have led to only 2 common case studies, resulting in a not effective comparison.

Enumeration affects testing budget. Each tool that we considered in our empirical comparison adopts a distinct approach for assembling sequences of operations into test scenarios. An exhaustive enumeration of sequences (RESTler) seems to be very time-consuming and less time-effective than a well-thought-out single sequence (RestTestGen). This highlights that when a large amount of requests (and consequently a large amount of time) cannot be spent in testing, focused approaches (such as the one adopted by RestTestGen) are preferable. Conversely, when a lot of resources can be allocated to testing, possibly in the cloud, testing an exhaustive enumeration of interaction sequences is probably acceptable.

Input generation Vs sequence enumeration. When generating test cases, search budget might be optimized across different dimensions, either to test with many different interaction sequences, or focusing on few sequences but with many different input data. According to our empirical observations, both these dimensions are important, but, especially when testing time is limited, focusing on exploring new input data (RestTestGen) seems to achieve higher generic coverage than exploring new interaction sequences (RESTler).

Multiple input generation algorithms. When generating test input data, some approaches focus on single strategies (such as using random values, dictionaries, mutations, etc.) or a combination of these. However, rather than just adopting a few data generation strategies, we observed that higher generic test coverage is achieved when more of those strategies are integrated and combined, including also data that have been observed as output of previously executed tests. In fact, test outputs might represent valid actual data from the database of the REST API under test. Investigating novel input data generation strategies is a promising research direction to deliver more effective testing approaches.

Coverage according to input Vs output metrics. High coverage with respect to *input* metrics seems to be easier to obtain, rather than high coverage with respect to *output* metrics. In fact, each tool scored very high values of *Operation*, *Path* and *Parameter* coverage, which are all input metrics. Indeed, obtaining 100% coverage for these input metrics is relatively easy: it just requires a tool to exercise once those operations, paths and input parameters that are documented in the OpenAPI specification.

Conversely, obtaining 100% coverage for output metrics is more challenging. While requests and input data are selected by testing tools, responses and output data are not directly under control of testing tools. For instance, a tool can not simply decide to cause a response with either a *correct* status code (e.g., `2XX`) or an *error* status code (e.g., `5XX`). To obtain distinct status codes, a testing tool has to guess both valid

and invalid input data, which is challenging, and it might require several attempts. Hence, output coverage metrics can be considered harder to satisfy, because they require out-of-specification knowledge about the service under test, and they can be considered a more reliable indicator of deeper testing than input coverage metrics.

VII. Related Work

Existing commercial test authoring tools, like [20]–[25], help developers to *manually* write tests that can be then automatically run by the tool. These approaches are not fully automatic, as the tools we have considered in the present work.

Concerning automatic tests generation for REST APIs, the research community proposes some interesting solutions, following mainly two different lines of work. One consists in *white-box* approaches, that rely on the availability of APIs source code to perform static analysis, or to instrument it to collect execution traces and metric values. In this context, Arcuri [14] proposes a fully automated solution to generate test cases with evolutionary algorithms, that requires the OpenAPI specification and the access to the Java bytecode of the REST API to test. This approach has been implemented as a tool prototype called EvoMaster, extended with the introduction of a series of novel testability transformations aimed at providing guidance in the context of commonly used API calls [26].

On a complementary direction, *black-box* approaches do not require any source code, which is often the case when using closed-source components and libraries. *Fuzzers* [27]–[31] are black-box testing tools that generate new tests starting from previously recorded API traffic: they fuzz and replay new traffic in order to find bugs. Some of these also exploit the OpenAPI specification of the service under test [28]–[30]. Godefroid et al. [32] propose a methodology to fuzz body payloads intelligently using JSON body schemas and advanced fuzzing rules (as done in RESTler [6]). Although they are automatic black-box tools, their goal is to generate input values to tests, so they cannot be used as standalone testing tools (except for the approach of Godefroid et al. [32] that has been implemented in RESTler). Ed-douibi et al. [13] propose a model-based approach for black-box automatic test case generation of REST APIs. A model is extracted from the OpenAPI specification of a REST API, to generate both nominal test cases (with input values that match the model) and faulty test cases (with input values that violate the model). However, we did not manage to install their proof-of-concept implementation (due to some errors in the source code), hence we had to exclude the work from our comparison. Karlsson et al. [12] propose QuickREST, a tool for property-based testing of RESTful APIs. Starting from the OpenAPI specification, they generate test cases with the aim at verifying whether the API under test complies with some properties (i.e., definitions) documented in the specification (e.g., status codes or schemas). Unfortunately, we had to exclude QuickREST from our work because the version we found online was incompatible with the case studies we randomly selected for our comparison (it did not manage to test any of our case studies). Segura et al. [33] propose another black-box approach, where the oracle is based on metamorphic relations among requests and responses. For instance, they send two queries to the same REST API, where the second query has stricter conditions than the first one (e.g., by adding a constraint). The result of the second query should be a proper subset of entries in the result of the first query. When the result is not a sub-set, the oracle reveals a defect. However, this approach only works for search-oriented APIs. Moreover, this technique is only partially automatic, because the user is supposed to manually identify the metamorphic relation to exploit and what input parameters to test.

To the best of our knowledge, the only black-box testing approaches for REST APIs which provide an implementation, i.e., a usable testing tool, are the one we have taken into account in our comparison (RestTestGen [5], RESTler [6], bBOXRT [7] and RESTest [8], presented in Section III). Regarding test coverage, the only work proposing a systematic approach to assess the coverage of REST APIs testing tools is the framework of Martin-Lopez et al. [3], that we have taken as basis for our comparison.

VIII. Conclusion

Despite several approaches and automated tools are available to test cases generation for REST APIs, the literature is still missing an explicit comparison of them. In this paper, we defined an experimental framework that includes a benchmark of REST APIs case studies and a coverage measurement infrastructure. We adopted this framework to carry out a comparison of four state-of-the-art automated black-box REST APIs testing tools (RestTestGen, RESTler, bBOXRT, and RESTest) in terms of robustness and coverage.

RESTler appears to be the most robust tool, being able to test all the case studies without incurring in crashes or failures. Instead, the strategy of RestTestGen, based on data dependencies among operations, appears to be the most effective, as it overtakes the other approaches in several coverage metrics.

Based on our experimental results, we formulated some considerations that might guide developers in making an informed decision on which tool to adopt.

As a future work, we plan to evolve the comparison with new testing tools, along with the updated versions of the already considered tools. Another interesting aspect we plan to investigate is how the specification-based coverage we adopted correlates with code coverage. Indeed, even if code coverage is inevitably more accurate, sometimes a "black-box coverage" is the only viable option. With a study correlating the two approaches we can asses in which situations, or under which assumptions, the specification-based coverage is still an acceptable solution. Finally, we intend to extend our experimental framework, that currently just focuses on test coverage, to also measure defect detection capabilities.

Acknowledgment

This paper has been partially supported by project MIUR 2018-2022 "Dipartimenti di Eccellenza".

REFERENCES

[1] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, "Are rest apis for cloud computing well-designed? an exploratory study," in *Service-Oriented Computing*, Q. Z. Sheng, E. Stroulia, S. Tata, and S. Bhiri, Eds. Cham: Springer International Publishing, 2016, pp. 157–170.

[2] Anonymous, "Replication package: An empirical comparison of state-of-the-art black-box test case generation tools for restful apis," in *to be disclosed after acceptance*.

[3] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Test coverage criteria for restful web apis," in *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 15–21. [Online]. Available: https://doi.org/10.1145/3340433.3342822

[4] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000, vol. 7.

[5] E. Viglianisi, M. Dallago, and M. Ceccato, "RESTTESTGEN: Automated black-box testing of RESTful APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 142–152.

[6] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API fuzzing," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 748–758. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00083

[7] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A black box tool for robustness testing of REST services," *IEEE Access*, vol. 9, pp. 24 738–24 754, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3056505

[8] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Black-box constraint-based testing of RESTful web APIs," in *Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings*, ser. Lecture Notes in Computer Science, E. Kafeza, B. Benatallah, F. Martinelli, H. Hacid, A. Bouguettaya, and H. Motahari, Eds., vol. 12571. Springer, 2020, pp. 459–475. [Online]. Available: https://doi.org/10.1007/978-3-030-65310-1_33

[9] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Checking security properties of cloud service REST APIs," in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 387–397. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00046

[10] A. Martin-Lopez, S. Segura, and A. Ruiz-Corts, "A catalogue of inter-parameter dependencies in restful web apis," 10 2019, pp. 399–414.

[11] A. Martin-Lopez, S. Segura, C. Muller, and A. Ruiz-Cortes, "Specification and automated analysis of inter-parameter dependencies in web APIs," *IEEE Transactions on Services Computing*, pp. 1–1, 2021.

[12] S. Karlsson, A. Causevic, and D. Sundmark, "QuickREST: Property-based test generation of OpenAPI-described RESTful APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 131–141.

[13] H. Ed-Douibi, J. L. C. Izquierdo, and J. Cabot, "Automatic generation of test cases for REST APIs: A specification-based approach," *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 181–190, 2018.

[14] A. Arcuri, "RESTful API automated test case generation with Evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, p. 3, 2019.

[15] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, "Restats: A test coverage tool for RESTful APIs," in *37th IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg City, Luxembourg, September 27 - October 1*, 2021.

[16] Spring.io, "Sring," https://spring.io/web-applications.

[17] The Pallet Projects, "Flask," https://palletsprojects.com/p/flask.

[18] Swagger.io, "Swagger-editor," https://editor.swagger.io/.

[19] ——, "Swagger-codegen," https://github.com/swagger-api/swagger-codegen.

[20] Postman, Inc., "Postman," https://www.getpostman.com/.

[21] SmartBear Software, "SoapUI," https://www.soapui.org/.

[22] Optimizory Technologies Pvt. Ltd., "vREST," https://vrest.io/.

[23] Borvid, "HttpMaster," http://www.httpmaster.net.

[24] A. Fortress, "API Fortress," http://apifortress.com.

[25] J. Haleby, "REST Assured," http://rest-assured.io/.

[26] A. Arcuri and J. P. Galeotti, "Testability transformations for existing APIs," in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 153–163. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00025

[27] API Fuzzer, "API Fuzzer," https://github.com/KissPeter/APIFuzzer.

[28] Fuzz-Lightyear, "Fuzz-Lightyear," https://github.com/Yelp/fuzz-lightyear.

[29] Fuzzy-Swagger, "Fuzzy-Swagger," https://github.com/namuan/fuzzy-swagger.

[30] Swagger-Fuzzer, "Swagger-Fuzzer," https://github.com/Lothiraldan/swagger-fuzzer.

[31] TnT-Fuzzer, "TnT-Fuzzer," https://github.com/Teebytes/TnT-Fuzzer.

[32] P. Godefroid, B. Huang, and M. Polishchuk, "Intelligent REST API data fuzzing," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 725–736. [Online]. Available: https://doi.org/10.1145/3368089.3409719

[33] S. Segura, J. Parejo, J. Troya, and A. Ruiz-Corts, "Metamorphic testing of RESTful web APIs," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2018.