Automated Test Generation for REST APIs: No Time to Rest Yet

Myeongsoo Kim
Georgia Institute of Technology
USA
mkim754@gatech.edu

Qi Xin
Wuhan University
China
qxin@whu.edu.cn

Saurabh Sinha
IBM T.J. Watson Research Center
USA
sinhas@us.ibm.com

Alessandro Orso
Georgia Institute of Technology
USA
orso@cc.gatech.edu

ABSTRACT

Modern web services routinely provide REST APIs for clients to access their functionality. These APIs present unique challenges and opportunities for automated testing, driving the recent development of many techniques and tools that generate test cases for API endpoints using various strategies. Understanding how these techniques compare to one another is difficult, as they have been evaluated on different benchmarks and using different metrics. To fill this gap, we performed an empirical study aimed to understand the landscape in automated testing of REST APIs and guide future research in this area. We first identified, through a systematic selection process, a set of 10 state-of-the-art REST API testing tools that included tools developed by both researchers and practitioners. We then applied these tools to a benchmark of 20 real-world open-source RESTful services and analyzed their performance in terms of code coverage achieved and unique failures triggered. This analysis allowed us to identify strengths, weaknesses, and limitations of the tools considered and of their underlying strategies, as well as implications of our findings for future research in this area.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Automated software testing, RESTful APIs

ACM Reference Format:
Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534401

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07.
https://doi.org/10.1145/3533767.3534401

1 INTRODUCTION

The last decade has seen a tremendous growth in the availability of web APIs—APIs that provide access to a service through a web interface. This increased popularity has been driven by various industry trends, including the advent of cloud computing, the broad adoption of microservices [60], and newer value propositions enabled by the "API economy". A majority of modern web APIs adhere to the REpresentational State Transfer (REST) architectural style [32] and are referred to as RESTful APIs, whose popularity is reflected in the availability of thousands of APIs in public directories (e.g., ProgrammableWeb [65] and APIs guru [4]).

Given the large number of applications that rely on web APIs, it is essential to test these APIs thoroughly to ensure their quality. It is therefore not surprising that many automated techniques and tools for REST APIs have been proposed in recent years [6, 9, 13–15, 27, 29, 35, 44, 51, 58, 74, 76, 79]. These techniques take as input a description of the API, in the OpenAPI specification format [62] or API Blueprint [1], and employ various strategies to generate test cases for exercising API endpoints defined in the specification.

Although these tools have been evaluated, these evaluations have been performed (1) in different settings (in terms of API benchmarks considered, experiment setup, and metrics used), (2) using benchmarks that are in some cases limited in size or closed-source, and (3) mostly in isolation or involving only limited comparisons. It is thus difficult to understand how these tools compare to one another. Recently, and concurrently to our effort, there has been some progress in this direction, via two empirical comparisons of black-box REST API testing tools [25, 38] and an evaluation of white-box and black-box REST API test generation [59]. These efforts are a step in the right direction but are still limited in scope. Specifically, and to the best of our knowledge, ours is the first study that (1) systematically identifies both academic and practitioners' tools to be used in the comparison, (2) analyzes the code of the benchmarks to identify code characteristics that affect the performance of the tools and differentiate them, (3) conducts an in-depth analysis of the failures revealed by the tools, and (4) identifies concrete and specific future research directions.

As benchmark for our study, we used a set of 20 RESTful services selected among those used in related work and through a search on GitHub, focusing on Java/Kotlin open-source services that did not excessively rely on external resources. To select the tools for our evaluation, we performed a thorough literature search, which resulted in 8 academic and 11 practitioners' tools. Among those,

we selected the tools that relied on commonly used REST API standards, such as OpenAPI, produced actual test cases, and were able to handle our benchmark of 20 services. The resulting set consisted of 10 tools overall: one white-box tool, EvoMasterWB [6, 9], and nine black-box tools, APIFuzzer [3], bBOXRT [51], Dredd [28], EvoMasterBB [10], RESTest [58], RESTler [14], RestTestGen [27, 79], Schemathesis [38], and Tcases [77]. In the paper, we provide a characterization of these tools along several dimensions, including their underlying input-generation approach, their support for stateful API testing, and the types of test oracles they use.

We applied these tools to our benchmark of 20 services and evaluated their performance in terms of code coverage achieved (lines, branches, and methods exercised) and different kinds of fault-detection ability (generic errors, unique failure points, and unique library failure points). Through a thorough analysis of the results, we also investigated the strengths and weaknesses of the tools and of their underlying test-generation strategies.

Overall, all tools achieved relatively low line and branch coverage on many benchmarks, which indicates that there is considerable room for improvement. Two common limitations of many tools, in particular, involve the inability of (1) generating input values that satisfy specific constraints (e.g., parameters that must have a given format), and (2) satisfying dependencies among requests (e.g., this endpoint must be called before these other endpoints). In general, we found that accounting for dependencies among endpoints is key to performing effective REST API testing, but existing techniques either do not consider these dependencies or use weak heuristics to infer them, which limits their overall effectiveness.

The paper also discusses lessons learned and implications for future research based on our results. For example, REST API testing techniques should leverage information embedded in the specification and the server logs to improve the quality of the test input parameters they generate. For another example, dependencies among services could be detected through static analysis, if the source code is available, and through natural language processing techniques applied to the service specification. Finally, in addition to discussing possible ways to improve REST testing approaches, we also present a proof-of-concept evaluation of these ideas that shows the feasibility of the suggested directions.

In summary, this work provides the following contributions:

- A comprehensive empirical study of automated test-generation tools for REST APIs that involves 10 academic and practitioners' tools and assesses their performance on 20 benchmarks in terms of code coverage and different types of fault-detection ability.
- An analysis of the strengths and weaknesses of the tools considered and their underlying techniques, with suggestions for improvement and a discussion of the implications for future research.
- An artifact with the tools and benchmarks that can allow other researchers to replicate our work and build upon it [12].

2 BACKGROUND

2.1 REST APIs

REST APIs are web APIs that follow the RESTful architectural style [32]. Clients can communicate with web services through their REST APIs by sending HTTP requests to the services and receiving responses. Clients send requests to access and/or manipulate resources managed by the service, where a *resource* represents data that a client may want to create, delete, or access. The request is sent to an API *endpoint*, which is identified by a resource path, together with an *HTTP method* that specifies the action to be performed on the resource. The most commonly used methods are POST, GET, PUT, and DELETE, which are used to create, read, update, and delete a resource, respectively. The combination of an endpoint plus an HTTP method is called an *operation*. In addition to specifying an operation, a request can optionally also specify HTTP headers containing metadata (e.g., the data format of the resource targeted) and a body that contains the payload for the request (e.g., text in JSON format containing the input values).

```
1  "/products/{productName}": {
2   "get": {
3    "operationId":
4       "getProductByName",
5    "produces":
6       ["application/json"],
7    "parameters": [{
8      "name": "productName",
9      "in": "path",
10     "required": true,
11     "type": "string"
12   }],
13   "responses": {
14    "200": {
15     "description":
16       "successful operation",
17     "schema": {
18      "$ref":
19        "#/definitions/Product"
20    },
21    "headers": {}
22 }}}}
```

Figure 1: An example OpenAPI specification.

After receiving and processing a request, the web service returns a response that includes, in addition to headers and possibly a body, an HTTP status code—a three-digit number that shows the outcome of the request. Status codes are organized into five suitably numbered groups. 1xx codes are used for provisional purposes, indicating the ongoing processing of a request. 2xx codes indicate successful processing of a request. 3xx codes indicate redirection and show, for instance, that the target resource has moved to a new URL (code 301). 4xx codes indicate client errors. For example, a 404 code indicates that the client has requested a resource that does not exist. Finally, 5xx codes indicate server errors in processing the request. Among this group, in particular, the 500 code indicates an Internal Server Error and typically corresponds to cases in which the service contains a bug and failed to process a request correctly. For this reason, many empirical studies of REST API testing tools report the number of 500 status codes triggered as the bugs they found (e.g., [6, 13, 14, 27, 29, 38, 44, 58]).

2.2 OpenAPI Specifications

The description of a service interface is typically provided by means of a specification that lists the available operations, as well as the input and output data types and possible return codes for each operation. A common, standard format for such specifications is the OpenAPI Specification [62] (previously known as Swagger). Other examples of specification languages include RAML [66] and API Blueprint [1].

Figure 1 shows a fragment of the OpenAPI specification for Features-Service, one of the benchmarks in our study. It specifies an API endpoint `/products/{productName}` (line 1), that supports HTTP method GET (line 2). The id for this operation is `getProductByName` (lines 3–4), and the format of the returned data is JSON (lines 5–6). To exercise this endpoint, the client *must* provide a value for the required parameter `productName` in the form of a path parameter (lines 7–12) (e.g., a request `GET /products/p` where `p` is the product

Table 1: Overview of REST API testing techniques and tools.

Name	Website	Testing Approach	Test-Generation Technique	Stateful	Oracle	Parameter Generation	Version Used
EvoMasterWB	[30]	White-box	Evolutionary algorithm	Yes	Status code	Random, Mutation-based, and Dynamic	v1.3.0[§]
EvoMasterBB	[30]	Black-box	Random Testing	Yes	Status code	Random	v1.3.0[§]
RESTler	[70]	Black-box	Dependency-based algorithm	Yes	Status code and predefined checkers	Dictionary-based and Dynamic	v8.3.0
RestTestGen	[27][‡]	Black-box	Dependency-based algorithm	Yes	Status code and response validation	Mutation-based, Default, Example-based, Random, and Dynamic	v2.0.0
RESTest	[69]	Black-box	Model-based testing	No	Status code and response validation	Constraint-solving-based, Random, and Mutation-based	Commit 625b80e
Schemathesis	[73]	Black-box	Property-based testing	Yes	Status code and response validation	Random and Example-based	v3.12.3
Dredd	[28]	Black-box	Sample-value-based testing	No	Status code and response validation	Example-based, Enum-based, Default, and Dummy	v14.1.0
Tcases	[77]	Black-box	Model-based testing	No	Status code	Random or Example-based	v3.7.1
bBOXRT	[19]	Black-box	Robustness testing	No	Status code and behavioral analysis	Random and Mutation-based	Commit 7c894247
APIFuzzer	[3]	Black-box	Random-mutation-based testing	No	Status code	Random and Mutation-based	Commit e2b536f

[‡] We obtained this tool directly from its authors when it was a private tool, so our version is not in the website.

name). If a product with that name exists and the service can process the request correctly, the client would receive a response with status code `200` (line 14) and the requested product data. Lines 18–19 reference the definition of the product data type, which contains information about the product, such as its id, name, features, and constraints (we omit the definition for lack of space).

Two operations have a *producer-consumer* dependency relationship when one of the operations (producer) can return data needed as input by the other operation (consumer). For example, operation `GET /products/{productName}` has producer-consumer relationship with operation `DELETE /products/{productName}/constraints/{constraintId}` (not shown in the figure). This is because a request sent to the former can lead to a response containing a constraint id needed to make a request to the latter.

3 TESTING TOOLS USED IN THE STUDY

3.1 Tools Selection

For selecting tools for the study, we searched for REST-API-related papers published since 2011 in top venues in the areas of software engineering, security, and web technologies (e.g., ASE, CCS, FSE, ICSE, ISSTA, NDSS, S&P, TOSEM, TSE, USENIX, WWW, and WSDM).[1] We identified relevant papers via keyword search, using the terms "rest", "api", and "test" and by transitively following citations in the identified papers. Among the resulting set of papers, we selected those about techniques and tools that (1) rely on well-known REST API standards (i.e., OpenAPI, RAML, and API Blueprint), and (2) produce actual test cases.

This process resulted in 19 publicly-available tools: eight research tools—EvoMasterWB, EvoMasterBB, RESTler, RESTest, RestTestGen, bBOXRT, Schemathesis, and api-tester—and 11 practitioners' tools—fuzz-lightyear, fuzzy-swagger, swagger-fuzzer, APIFuzzer, TnT-Fuzzer, RESTApiTester, Tcases, gadolinium, restFuzzer, Dredd, and kotlin-test-client.[2] More recently, we found a technical report [38] that describes two additional tools, cats [22] and gotswag [37]. Because Schemathesis, which is included in our study, outperforms them significantly [38], we did not include these tools in our comparison. We then eliminated tools that either did not work at all or failed to run on our benchmark of 20 services. It is worth noting that the remaining 10 tools, which are listed in Table 1, were also the most popular based on stars, watches, and forks in their GitHub repositories.

[1] The latest search was performed in December 2021.
[2] We define as research tools those whose technical details are presented in research papers.

3.2 Tools Description

Table 1 presents an overview of the 10 tools we selected along several dimensions: the URL where the tool is available (column 2); whether the tool is white-box or black-box (column 3); test-generation strategy used by the tool (column 4); whether the tool produces test cases that exercise APIs in a stateful manner (column 5); the types of oracle used by the tool (column 6); the approach used by the tool to generate input parameter values (column 7); and the version of the tool (column 8).

EvoMaster [7] can test a REST API in either white-box or black-box mode. In the study, we used the tool in both modes. We refer to the tool in the black-box mode as *EvoMasterBB* and in the white-box mode as *EvoMasterWB*. Given a REST API and the OpenAPI specification, both tools begin by parsing the specification to obtain the information needed for testing each operation. EvoMasterBB performs random input generation: for each operation, it produces requests to test the operation with random values assigned to its input parameters. EvoMasterWB requires access to the source code of the API. It leverages an evolutionary algorithm (the MIO algorithm [8] by default) to produce test cases with the goal of maximizing code coverage. Specifically, for each target (uncovered) branch, it evolves a population of tests by generating new ones while removing those that are the least promising (i.e., have the lowest fitness value) for exercising the branch in each iteration until the branch is exercised or a time limit is reached. EvoMasterWB generates new tests through sampling or mutation. The former produces a test from scratch by either randomly choosing a number of operations and assigning random values to their input parameters (i.e., random sampling) or accounting for operation dependencies to produce stateful requests (i.e., smart sampling). The approach based on mutation, conversely, produces a new test by changing either the structure of an existing test or the request parameter values. EvoMasterBB and EvoMasterWB use an automated oracle that checks for service failures resulting in a 5xx status code. Recent extensions of the technique further improve the testing effectiveness by accounting for database states [11] and making better use of resources and dependencies [86].

RESTler [14] is a black-box technique that produces stateful test cases to exercise "deep" states of the target service. To achieve this, RESTler first parses the input OpenAPI specification and infers producer-consumer dependencies between operations. It then uses a search-based algorithm to produce sequences of requests that conform to the inferred dependencies. Each time a request is appended to a sequence, RESTler executes the new sequence to check its validity. It leverages dynamic feedback from such execution to prune the search space (e.g., to avoid regenerating invalid sequences that

were previously observed). An early version of RESTler relies on a predefined dictionary for input generation and targets finding 5xx failures. Newer versions of the technique adopt more intelligent fuzzing strategies for input generation [35] and use additional security-related checkers [15].

RestTestGen [27, 79] is another black-box technique that exploits the data dependencies between operations to produce test cases. First, to identify dependencies, RestTestGen matches names of input and output fields of different operations. It then leverages the inferred dependency information, as well as predefined priorities of HTTP methods, to compute a testing order of operations and produce tests. For each operation, RestTestGen produces two types of tests: nominal and error tests. RestTestGen produces nominal tests by assigning to the input parameters of an operation either (1) values dynamically obtained from previous responses (with high probability) or (2) values randomly generated, values provided as default, or example values (with low probability). It produces error tests by mutating nominal tests to make them invalid (e.g., by removing a required parameter). RestTestGen uses two types of oracles that check (1) whether the status code of a response is expected (e.g., 4xx for an error test) and (2) whether a response is syntactically compliant with the response schema defined in the API specification.

RESTest [58] is a model-based black-box input generation technique that accounts for inter-parameter dependencies [56]. An *inter-parameter dependency* specifies a constraint among parameters in an operation that must be satisfied to produce a valid request (e.g., if parameter A is used, parameter B should also be used). To produce test cases, RESTest takes as input an OpenAPI specification with the inter-parameter dependencies specified in a domain-specific language [55] and transforms such dependencies into constraints [40]. RESTest leverages constraint-solving and random input generation to produce nominal tests that satisfy the specified dependencies and may lead to a successful (2xx) response. It also produces faulty tests that either violate the dependencies or are not syntactically valid (e.g., they are missing a required parameter). RESTest uses different types of oracles that check (1) whether the status code is different from 5xx, (2) whether a response is compliant with its defined schema, and (3) whether expected status codes are obtained for different types of tests (nominal or faulty).

Schemathesis [38] is a black-box tool that performs property-based testing (using the Hypothesis library [39]). It performs negative testing and defines five types of oracles to determine whether the response is compliant with its defined schema based on status code, content type, headers, and body payload. By default, Schemathesis produces non-stateful requests with random values generated in various combinations and assigned to the input parameters. It can also leverage user-provided input parameter examples. If the API specification contains "link" values that specify operation dependencies, the tool can produce stateful tests as sequences of requests that follow these dependencies.

Dredd [28] is another open-source black-box tool that validates responses based on status codes, headers, and body payloads (using the Gavel library [34]). For input generation, Dredd uses sample values provided in the specification (e.g., examples, default values, and enum values) and dummy values.

Tcases [77] is a black-box tool that performs model-based testing. First, it takes as input an OpenAPI specification and automatically constructs a model of the input space that contains key characteristics of the input parameters specified for each operation (e.g., a characteristic for an integer parameter may specify that its value is negative). Next, Tcases performs *each-choice* testing (i.e., 1-way combinatorial testing [49]) to produce test cases that ensure that a valid value is covered for each input characteristic. Alternatively, Tcases can also construct an example-based model by analyzing the samples provided in the specification and produce test cases using the sample data only. For test validation, Tcases checks the status code based on the request validity (as determined by the model).

bBOXRT [51] is a black-box tool that aims to detect robustness issues of REST APIs. Given a REST API and its OpenAPI specification, the tool produces two types of inputs, valid and invalid, for robustness testing. It first uses a random approach to find a set of valid inputs whose execution can result in a successful (i.e., 2xx) status code. Next, it produces invalid inputs by mutating the valid inputs based on a predefined set of mutation rules and exercises the REST API with those inputs. The approach involves manual effort to analyze the service behavior and (1) classify it based on a failure mode scale (the CRASH scale [47]) and (2) assign a set of behavior tags that provide diagnostic information.

APIFuzzer [3] is a black-box tool that performs fuzzing of REST APIs. Given a REST API and its specification, APIFuzzer first parses the specification to identify each operation and its properties. Then, it generates random requests conforming to the specification to test each operation and log its status code. Its input-generation process targets the body, query string, path parameter, and headers of requests and applies random generation and mutation to these values to obtain inputs. APIFuzzer uses the generated inputs to submit requests to the target API and produces test reports in the JUnit XML format.

4 EMPIRICAL STUDY

In our study, we investigated three research questions for the set of tools we considered and described in the previous section:

- **RQ1**: How much code coverage can the tools achieve?
- **RQ2**: How many error responses can the tools trigger?
- **RQ3**: What are the implications of our findings?

To answer RQ1, we evaluated the tools in terms of the line, branch, and method coverage they achieve to investigate their abilities in exercising various parts of the service code.

For RQ2, we compared the number of 500 errors triggered by the tools to investigate their fault-finding ability, as is commonly done in empirical evaluations of REST API testing techniques (e.g., [6, 13, 14, 27, 29, 38, 44, 52, 58, 79, 85]). We measured 500 errors in three ways. First, to avoid counting the same error twice, we grouped errors by their stack traces, and reported *unique 500 errors*. Therefore, unless otherwise noted, 500 errors will be used to denote unique 500 errors in the rest of the paper. Second, different unique 500 errors can have the same failure point (i.e., the method-line pair at the top of the stack trace). Therefore, to gain insight into occurrences of such failure sources, we also measured *unique failure points*, which group stack traces by their top-most entries. Finally, we differentiated cases in which the unique failure point

Table 2: RESTful web services used in the empirical study.

Name	Total LOC	Java/Kotlin LOC	# Operations
CWA-verification	6,625	3,911	5
ERC-20 RESTful service	1,683	1,211	13
Features-Service	1,646	1,492	18
Genome Nexus	37,276	22,143	23
Languagetool	677,521	113,277	2
Market	14,274	7,945	13
NCS	569	500	6
News	590	510	7
OCVN	59,577	28,093	192
Person Controller	1,386	601	12
Problem Controller	1,928	983	8
Project tracking system	88,634	3,769	67
ProxyPrint	6,263	6,037	117
RESTful web service study	3,544	715	68
Restcountries	32,494	1,619	22
SCS	634	586	11
Scout-API	31,523	7,669	49
Spring boot sample app	1,535	606	14
Spring-batch-rest	4,486	3,064	5
User management	5,108	2,878	23
Total	977,296	207,609	675

was a method from the web service itself from cases in which it was a third-party library used by the service; we refer to the latter as *unique library failure points*.

To answer RQ3, we (1) provide an analytical assessment of the studied tools in terms of their strengths and weaknesses and (2) discuss the implications of our study for the development of new techniques and tools in this space.

Next, we describe the benchmark of web services we considered (§4.1) and our experiment setup (§4.2). We then present our results for the three research questions (§4.3–4.5). We conclude this section with a discussion of potential threats to validity of our results (§4.6).

4.1 Web Services Benchmark

To create our evaluation benchmark, we first selected RESTful web services from existing benchmarks used in the evaluations of bBOXRT, EvoMaster, and RESTest. Among those, we focused on web services implemented in Java/Kotlin and available as open-source projects, so that we could use EvoMasterWB. This resulted in an initial set of 30 web services. We then searched for Java/Kotlin repositories on GitHub using tags "rest-api" and "restful-api" and keywords "REST service" and "RESTful service", ranked them by number of stars received, selected the 120 top entries, and eliminated those that did not contain a RESTful specification. This additional search resulted in 49 additional web services. We tried installing and testing this total set of 79 web services locally and eliminated those that (1) did not have OpenAPI Specifications, (2) did not compile or crashed consistently, or (3) had a majority of operations that relied on external services with request limits (which would have unnaturally limited the performance of the tools). For some services, the execution of the service through the EvoMaster driver class that we created crashed without a detailed error message. Because this issue was not affecting a large number of services, we decided to simply exclude the services affected.

In the end, this process yielded 20 web services, which are listed in Table 2. For each web service considered, the table lists its name, total size, size of the Java/Kotlin code, and number of operations.

Table 3: Average line, branch, and method coverage achieved and 500 errors found in one hour (E: unique error, UFP: unique failure point, ULFP: unique library failure point).

Tool	Line	Branch	Method	#500 (E/UFP/ULFP)
EvoMasterWB	**52.76%**	**36.08%**	**52.86%**	**33.3** / **6.4** / **3.2**
RESTler	35.44%	12.52%	40.03%	15.1 / 2.1 / 1.3
RestTestGen	40.86%	21.15%	42.31%	7.7 / 2 / 1
RESTest	33.86%	18.26%	33.99%	7.2 / 1.9 / 1.1
bBOXRT	40.23%	22.20%	42.48%	9.5 / 2.1 / 1.3
Schemathesis	43.01%	25.29%	43.65%	14.2 / 2.8 / 2
Tcases	37.16%	16.29%	41.47%	18.5 / 3.5 / 2.1
Dredd	36.04%	13.80%	40.59%	6.9 / 1.5 / 0.9
EvoMasterBB	45.41%	28.21%	47.17%	16.4 / 3.3 / 1.8
APIFuzzer	32.19%	18.63%	33.77%	6.9 / 2.2 / 1.3

We obtained the API specifications for these services directly from their corresponding GitHub repositories.

4.2 Experiment Setup

We ran our experiments on Google Cloud e2-standard-4 machines running Ubuntu 20.04. Each machine had 4 2.2GHz Intel-Xeon processors and 16GB RAM. We installed on each machine the 10 testing tools from Table 1, the 20 web services listed in Table 2, and other software used by the tools and services, including OpenJDK 8, OpenJDK 11, Python 3.8, Node.js 10.19, and .NET framework 5.0. We also set up additional services used by the benchmarks, such as MySQL, MongoDB, and a private Ethereum network. This process was performed by creating an image with all the tools, services, and software installed and then instantiating the image on the cloud machines.

We ran each tool with the recommended configuration settings as described in the respective paper and/or user manual. Some of the tools required additional artifacts to be created (see also Section 4.6). For RESTest, we created inter-parameter dependency specifications for the benchmark applications, and for EvoMasterWB, we implemented driver programs for instrumenting Java/Kotlin code and database injection. For lack of space, we omit here details of tool set up and configuration. A comprehensive description that enables our experiments and results to be replicated is available at our companion website [12].

We ran each tool for one hour, with 10 trials to account for randomness. Additionally, we ran each tool once for 24 hours, using a fresh machine for each run to avoid any interference between runs. For the one-hour runs, we collected coverage and error-detection data in 10-minute increments to investigate how these metrics increased over time. For the 24-hour run, we collected the data once at the end. We collected coverage information using the JaCoCo code-coverage library [41].

4.3 RQ1: Coverage Achieved

Table 3 presents the results for line, branch, and method coverage achieved, and 500 errors, unique failure points, and unique library failure points detected by the 10 tools in one hour. The reported results are the average of the coverage achieved and errors found over all services and all ten trials. Unlike Table 3, which shows only average results, Figure 2 presents coverage data (top) and error-detection results (bottom) as box plots. In the figure, the horizontal axis represents the 10 tools, and the vertical axis represents, for each

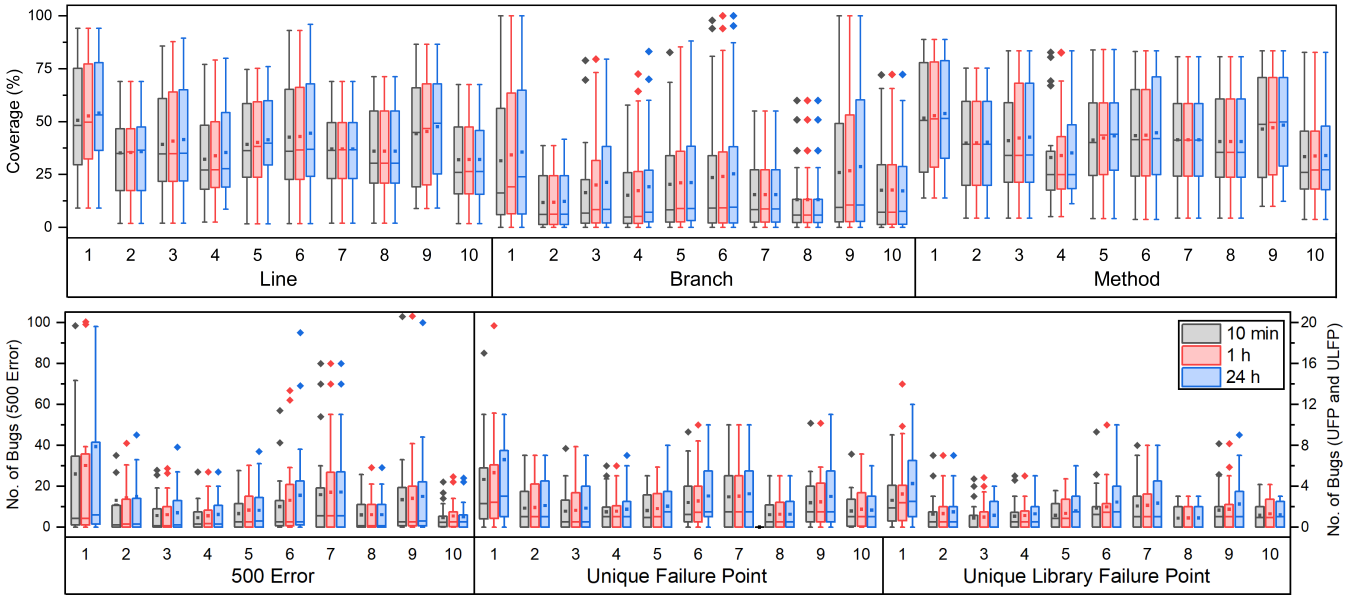

Figure 2: Code coverage achieved and number of unique 500 errors, unique failure points, and unique library failure points detected over all services by the ten tools in 10 minutes, 1 hour, and 24 hours (1: EvoMasterWB, 2: RESTler, 3: RestTestGen, 4: RESTest, 5: bBOXRT , 6: Schemathesis, 7: Tcases, 8: Dredd, 9: EvoMasterBB, 10: APIFuzzer).

tool, the results achieved by that tool after 10 minutes, 1 hour, and 24 hours. Each box represents average, median, standard deviation, min, and max for 20 data points, one per service considered. For the 10-minute and 1-hour runs, each point represents the coverage achieved (or number of errors revealed) averaged over 10 trials. For the 24-hour runs, the values reported are from single trials.

Overall coverage achieved. Table 3 shows that the tools did not achieve a high coverage in general. In one hour, the best-performing tool (EvoMasterWB) achieved less than 53% line coverage, less than 37% branch coverage, and less than 53% method coverage. The other black-box tools achieved lower coverage; the best-performing tool among them is EvoMasterBB, with 45.41% line coverage, 28.21% branch coverage, and 47.17% method coverage. We identified three key factors responsible for these low coverage rates.

- **Parameters value generation**. The tools generate many invalid requests that are rejected by the services and fail to exercise service functionality in depth. In most cases, this occurs when the parameters take domain-specific values or have data-format restrictions. As an example, for a parameter that takes email IDs as values, one of the services enforced the condition that the email should contain a '@' character. Tools wasted a significant amount of time attempting to pass that check, but they all failed in the one-hour and 24-hour runs in our experiment.
- **Operations dependency detection**. Identifying producer-consumer dependencies between operations is key to generating stateful tests. Existing tools either do not account for such dependencies or use a simple heuristic approach, which can lead to many false positives and false negatives in the dependencies computed. Failing to identify a dependency between producer A and consumer B can result in inadequate testing of B due to not testing A first. This is because B may require the execution of A to set the service state needed for its testing (e.g., by adding valid data to the database) or obtain valid parameter values. As an example, for Languagetool, most of the tools fail to identify a dependency between the two operations `POST /check` (consumer) and `GET /languages` (producer) in 24 hours. As a result, they did not leverage the output of the producer, which contained language information, and were unable to produce valid requests for the consumer.
- **Mismatch between APIs and their specifications**. The tools produce test cases by leveraging the API specifications, which are expected to faithfully reflect the API implementations and define, for each endpoint, supported HTTP methods, required input parameters, responses, and so on. Unfortunately, this is not always the case and, if the specification does not match the API implementation, the tools can produce incomplete, invalid, or even no requests at all for exercising the affected operations.

Existing tools fail to achieve high code coverage due to limitations of the approaches they use for generating parameter values and detecting operation dependencies. The effectiveness of the tools is also hindered by mismatches between API implementations and specifications.

Coverage increase over time. As we mentioned above, we collected coverage data in 10-minute increments, as well as for 24 hours. Figure 2 illustrates how the coverage rate increased from 10 minutes to 1 hour, and to 24 hours (more detailed results are available in our artifact [12]). As the figure shows, in many cases, the tools already achieved their highest level of code coverage in 10 minutes. However, there were several cases in which code coverage kept increasing over time. We investigated the 6 services for which the tools manifested this behavior—Features-Service, NCS, Restcountries, SCS, RESTful Web Service Study, and User management—and found that they all have some distinct characteristics (compared to the other 14 services). Specifically these services generally have simpler input parameters that do not take domain-specific or otherwise constrained values, so additional input generation tends to cover additional code. Conversely, for the services that have domain-specific or constrained parameter values (e.g., "email should contain @" or "year should be between 1900 and 2200"), the tools tend to hit a coverage wall relatively soon because they are inherently unable to generate inputs that satisfy these requirements.

> The coverage achieved by testing tools grows over time on services that have simpler input parameters with no domain-specific or constrained values.

4.4 RQ2: Error Responses Triggered

The ultimate goal of testing is to find bugs. In this section, we focus on comparing the testing tools in terms of their fault-finding ability, which we measured in terms of the numbers of unique 500 errors, failure points, and library failure points detected. Column 5 in the previously presented Table 3 provides this information averaged over the services considered and the trials performed. As the table shows, EvoMasterWB is the best performer by a wide margin, followed by Tcases, EvoMasterBB, RESTler, and Schemathesis. The box plot at the bottom of Figure 2 presents a more detailed view of these results by illustrating the distribution of errors detected across services and the increase in errors detected over time. The "500 Error" segment of the plot shows that EvoMasterWB outperforms all the other tools in terms of the median values as well, although with a larger spread.

> EvoMasterWB, by having access to source code and performing coverage-driven testing, achieves significantly higher coverage than black-box tools. However, as we will show in Section 4.5.2, there are cases in which EvoMasterWB cannot produce requests covering some service functionality, whereas black-box tools can.

The figure also shows that the number of unique failure points is considerably smaller than the number of unique errors—on average, there are 3 to 7 times fewer failure points than errors, indicating that several 500 errors occur at the same program points but along different call chains to those points. Another observation is that approximately half of the unique failure points occur in library methods. A more detailed analysis of these cases revealed that failure points in library methods could have more serious consequences on the functionality and robustness of a service, in some cases also leaving it vulnerable to security attacks. The reason is that failures in service code mostly originate at statements that throw exceptions after performing some checks on input values; the following fragment is an illustrative example, in which the thrown exception is automatically mapped to a 500 response code by the REST API framework being used:

```
1 if (product == null) {
2     throw new ObjectNotFoundException(name);
3 }
```

Conversely, in the case of library failure points, the root cause of the failures was often an unchecked parameter value, possibly erroneous, being passed to a library method, which could cause severe issues. We found a particularly relevant example in the ERC-20 RESTful service, which uses an Ethereum [84] network. An Ethereum transaction requires a fee, which is referred to as gas. If an invalid request, or a request with insufficient gas, is sent to Ethereum by the service, the transaction is canceled, but the gas fee is not returned. This is apparently a well-known attack scenario in Blockchain [72]. In this case, the lack of suitable checks in the ERC-20 service for requests sent to Ethereum could have costly repercussions for the user. We found other examples of unchecked values passed from the service under test to libraries, resulting in database connection failures and parsing errors.

> The severity of different 500 errors can vary considerably. Failure points in the service code usually occur at throw statements following checks on parameter values. Failure points outside the service, however, often involve erroneous requests that have been accepted and processed, which can lead to severe failures.

We further investigated which factors influence the error-detection ability of the tools. First, we studied how the three types of coverage reported in Table 3 correlate with the numbers of the 500 errors found using the Pearson's Correlation Coefficient [33]. The results showed that there is a strong positive correlation between the coverage and number of 500 errors (coefficient score is ~0.7881 in all cases). Although expected, this result confirms that it makes sense for tools to use coverage as a goal for input generation: if a tool achieves higher coverage, it will likely trigger more failures.

> There is a strong positive correlation between code coverage and number of faults exposed. Tools that achieve higher coverage usually also trigger more failures.

Second, we looked for patterns, not related to code coverage, that can increase the likelihood of failures being triggered. This investigation revealed that exercising operations with different combinations of parameters can be particularly effective in triggering service failures. The failures in such cases occur because the service lacks suitable responses for some parameter combinations. In fact, Tcases and Schemathesis apply this strategy to their advantage and outperform the other tools. Generating requests with different input types also seems to help in revealing more faults.

```
1 public static String subject(String directory, String file) {
2   int result = 0;
3   String[] fileparts = null;
4   int lastpart  = 0;
5   String suffix = null;
6   fileparts = file.split(".");
7   lastpart = fileparts.length - 1;
8   if (lastpart > 0) { ... } //target branch
9   return "" + result;
10 }
```

Figure 3: A method used for parsing the file suffix in the SCS web service.

> Exercising operations with various parameter combinations and various input types helps revealing more faults in the services under test.

4.5 RQ3: Implications of Our Results

We next discuss the implications of our results for future research on REST API testing and provide an analytical assessment of testing strategies employed by the white-box and black-box testing tools we considered.

4.5.1 *Implications for Techniques and Tools Development.*

Better input parameter generation. Our results show that the tools we considered failed to achieve high code coverage and could be considerably improved. Based on our findings, one promising direction in this context is better input parameter generation.

In particular, for white-box testing, analysis of source code can provide critical guidance on input parameter generation. In fact, EvoMasterWB, by performing its coverage-driven evolutionary approach, achieves higher coverage and finds more 500 errors than any of the black-box tools. There are, however, situations in which EvoMasterWB's approach cannot direct its input search toward better coverage. Specifically, this happens when the fitness function used is ineffective in computing a good search gradient. As an example, for the code shown in Figure 3, which parses the suffix of a file, EvoMasterWB always provides a string input `file` that leads to `lastpart` in line 8 evaluating to 0. The problem is that EvoMasterWB cannot derive a gradient from the condition `lastpart > 0` to guide the generation of inputs that exercise the branch in line 8. In this case, symbolic execution [17, 46] could help find a good value of `file` by symbolically interpreting the method and solving the constraint derived at line 8.

For black-box testing approaches, which cannot leverage information in the source code, using more sophisticated testing techniques (e.g., combinatorial testing at 2-way or higher interaction levels) could be a promising direction [43, 80, 85]. Also, black-box testing tools could try to leverage useful information embedded in the specification and other resources to guide input parameter generation.

Our analysis in Section 4.5.2 below shows that using sample values for input parameter generation can indeed lead to better tests. Therefore, another possible way to improve test generation would be to automatically extract sample values from parameter descriptions in the specification. For example, the description of the input parameter `language`, shown in the following OpenAPI fragment for Languagetool (lines 4–9), suggests useful input values, such as "en-US" and "en-GB":

```
1 "name": "language",
2 "in": "formData",
3 "type": "string",
4 "description": "A language code like `en-US`, `de-DE`, `fr`,
5 or `auto` to guess the language automatically
6 (see `preferredVariants` below). For languages with variants
7 (English, German, Portuguese) spell checking will
8 only be activated when you specify the variant,
9 e.g. `en-GB` instead of just `en`.",
10 "required": true
```

Another source of useful hints for input generation can be response messages from the server, such as the following one:

```
1 Received: "HTTP/1.1 400 Bad Request\r\nDate: Wed, 28 Apr 2021
2 00:38:32 GMT\r\nContent-length: 41\r\n\r\nError: Missing 'text'
3 or 'data' parameter"
```

To investigate the feasibility of leveraging input parameter descriptions provided in the REST API specifications to obtain valid input values, we implemented a proof-of-concept text-processing tool that checks whether there are suggested values supplied in the parameter description for a given input parameter. We applied the tool to two of the services, OCVN and Languagetool, for which the existing testing tools failed to obtain high coverage (less than 10% line code coverage for most cases). With the help of the tool, we identified developer-suggested values for 934 (32%) of the 2,923 input parameters. This preliminary result suggests that leveraging parameter descriptions for input generation may be a feasible and promising direction.

Along similar lines, we believe that natural-language processing (NLP) [54] could be leveraged to analyze the parameter description and extract useful information. For example, a technique may first perform token matching [83] to identify what parameters from the specification are mentioned in server messages and then use parts-of-speech tagging [81] and dependency parsing [48] to infer parameter values.

Also in this case, to investigate the feasibility of this approach, we implemented a proof-of-concept prototype that parses dependency information from natural-language descriptions in OpenAPI specifications and server messages and collects nouns, pronouns, conjunctions, and their dependencies. Our prototype detects parameter names with simple string matching on nouns and pronouns, and relationships between parameters via conjunctions and dependencies. As an example, the top part of Figure 4 shows the parsed dependencies for parameter `preferredVariants` of endpoint `/check`. By analyzing the tokens `with`, `language`, and `auto` with the connected dependencies (case, punct, nsubj, and dep), our prototype can determine that parameter `language` must be set to "auto". This simple approach was able to automatically detect 8 of the 12 unique inter-parameter dependencies that we manually found in the benchmark APIs. It was also able to detect useful parameter values. For example, none of the black-box tools studied could generate values for parameter `language`; yet, our prototype detected useful values such as "en-US", "de-DE", and "fr" (the parsed dependencies are shown in the bottom part of Figure 4). As before, these preliminary results show the feasibility of applying NLP techniques to descriptions in OpenAPI specifications and server messages for improving REST API testing.

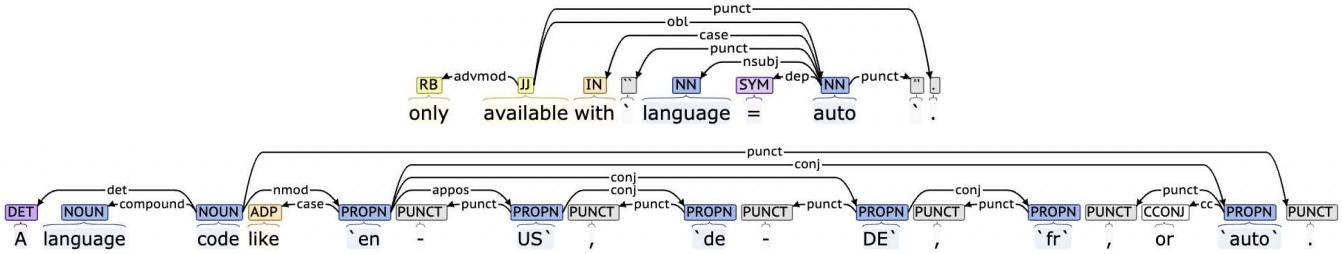

Figure 4: Parsed dependency graphs of `preferredVariants` parameter description (top) and `language` parameter description (bottom) from Languagetool's OpenAPI specification.

Better support for stateful testing. As discussed in Section 4.3 and also stressed in related work [7, 14], producing stateful tests by inferring producer-consumer relationships between operations is key to effective REST API testing. Current tools, such as RestTestGen and RESTler, rely on simple heuristics to infer the producer-consumer dependencies and are inaccurate in identifying such dependencies, which can easily lead to false positives and negatives. There is a need for more sophisticated approaches that address the flexibility issues that we discuss later in Section 4.5.2. These approaches should be able to account for responses dynamically generated and to match fields of different types while still being precise (e.g., by not identifying a dependency between two consumers that rely on the same parameters to execute).

To determine whether two operations have a dependency relationship, one possible approach would be to check whether they have "related" properties, such as input parameters and response fields, that have the same or similar names. To explore the feasibility of this direction, we performed a quick case study in which we manually identified each operation that uses path parameters and has a dependency relationship with at least another operation. For each of them, we then identified its path parameter(s) and compared its textual similarity with other operations' response fields using the NLTK's scoring library [61]. In this way, by considering the top-3 matches for each parameter, we were able to correctly identify almost 80% of the operations involved in a dependency relationship. Based on our findings in RQ1, we expect that a similar approach could considerably help tools to achieve higher code coverage.

We also believe that machine learning [42] could help identify producer-consumer relationships and related dependencies. For example, one could train, via supervised learning [87], a classifier that accounts for a variety of features related to operation dependencies (e.g., HTTP methods, shared input parameters, field matching on dynamic object, field types) and then use this classifier to make predictions on potential dependencies.

4.5.2 *Analytical Comparison of White-Box and Black-Box Tools.*

Next, we present an analytical assessment of the tools with respect to strengths and weaknesses of their approaches for generating input parameter values and sequences of API requests. We also provide illustrative examples taken from the benchmark services.

White-box vs. black-box tools. Among the tools we considered, EvoMasterWB is the only one that performs white-box testing. By having access to the API source code and performing coverage-driven testing, EvoMasterWB achieves higher coverage than the other tools—according to Table 3, it achieves ~53% line coverage, ~36% branch coverage, and ~53% method coverage.

```
1  if ("male".equals(sex)) {
2      if ("mr".equals(title) || "dr".equals(title) ||
3          "sir".equals(title) || "rev".equals(title) ||
4          "rthon".equals(title) || "prof".equals(title)) {
5              result = 1;
6      }
7  } else if ("female".equals(sex)) {
8      if ("mrs".equals(title) || "miss".equals(title) ||
9          "ms".equals(title) || "dr".equals(title) ||
10         "lady".equals(title) || "rev".equals(title) ||
11         "rthon".equals(title) || "prof".equals(title)) {
12             result = 0;
13 }}
```

Figure 5: Sample code from the SCS web service.

To illustrate, Figure 5 shows an if-statement in which the two branches (lines 5 and 12) are exercised only by test cases produced by EvoMasterWB. The if-statement is responsible for handling requests for operation `GET /api/title/{sex}/{title}` of the SCS web service. To cover these branches, a tool must produce valid requests with relevant string values for parameters `sex` and `title`. Using an evolutionary algorithm with a fitness function that measures branch distance, EvoMasterWB successfully generates values "male" for `sex` and "dr" for `title` to exercise line 5, and values "female" and "prof" for those parameters to cover line 12. The black-box tools are unable to do this by using random and/or sample values.

Another benefit of EvoMasterWB's testing strategy is that once it produces a test case exercising a branch A, it will not generate similar test cases to exercise A again. This is due to its MIO algorithm [8], which handles test case generation for the target branches separately and focuses on uncovered branches.

However, there are also cases in which EvoMasterWB fails to create a sequence of requests for covering some API functionality, whereas a black-box tool is able to. Consider the code fragment shown in Figure 6, which handles requests for operation O_2: `GET /products/{productName}/configurations` of Features-Service. To cover line 6, a request must specify a product that exists and has configurations associated with it. None of the requests created by EvoMasterWB satisfy both conditions: operation O_1: `POST /products/{productName}/configurations/{configurationName}` must be called before O_2 to associate configurations with a product, and EvoMasterWB fails to recognize this producer-consumer relation. In contrast, Schemathesis uses a testing strategy that orders O_1 before O_2 and leverages sample values from the API specification to link two operations with the same input values. It can therefore generate a sequence that creates a product, adds configurations to it, and retrieves the configurations—a sequence that covers line 6.

```
public List<String>
  getConfigurationsNamesForProduct(String productName) {
  List<String> configurationsForProduct=new ArrayList<String>();
  for (ProductConfiguration productConfiguration :
    productsConfigurationsDAO.findByProductName(productName)) {
    configurationsForProduct.add(productConfiguration.getName());
  }
  return configurationsForProduct;
}
```

Figure 6: A method of Features-Service used to find configuration names associated with a product.

Assessment of black-box tools. Although black-box tools seems to be less effective than EvoMasterWB in terms of coverage achieved, which can also often result in fewer faults triggered, they have wider applicability by virtue of being agnostic to the language used for the service implementation. Among the black-box tools, EvoMasterBB and Schemathesis achieve better coverage than the other tools in terms of all the three metrics considered (unique 500 errors, failure points, and library failure points).

EvoMasterBB uses an evolutionary algorithm to generate successful requests, create sequences from them, and discover in this way operation dependencies. This way of operating allows it to find more valid request sequences than other tools that just use a randomized approach.

Schemathesis is the next best black-box tools. One characteristic of this tool is that it considers example values provided in the API specification, which lets it leverage the domain knowledge encoded in such examples. Moreover, it reuses values in creating request sequences, which enables it to successfully create covering sequences such as that for the loop body in Figure 6.

RestTestGen also has similar features, but we empirically found that its heuristic algorithm, which relies upon the matching of response fields and parameter names to infer producer-consumer dependencies between operations, yields many false positives and false negatives. This issue weakens the effectiveness of its stateful testing and leads to slightly lower code coverage than EvoMasterBB.

RESTler tries to infer producer-consumer relationships by leveraging feedback from processed requests. An important limiting factor for RESTler, however, is that it relies on a small set of dictionary values for input generation, which hinders its ability to exercise a variety of service behaviors.

Dredd does not perform random input generation but uses dummy values and input examples (provided in the specification) to produce requests. This prevents Dredd from generating invalid requests exercising uncommon behaviors in a service.

Finally, the other tools' random-based approaches are unlikely to produce valid test cases needed for input mutation, which limits their overall effectiveness.

Overall, black-box tools fail to achieve high coverage because they largely rely on random testing and/or leverage a limited set of sample data for input generation. Among these tools, Schemathesis and EvoMasterBB performed better than the others in our experiments due to some specific characteristics of the input generation approaches they use.

4.6 Threats to Validity

Like any empirical evaluation, our study could suffer from issues related to internal and external validity. To mitigate threats to internal validity, we used the implementations of the testing tools provided by their authors (the tool versions used are listed in Table 1). Our implementation consists of Python code for analyzing the log files of the web services to compute unique 500 errors, failure points, and library failure points. We thoroughly tested and spot checked the code and manually checked the testing results (coverage and faults) for a sample of the tools and web services to gain confidence in the validity of our results.

As for the threats to external validity, because our evaluation is based on a benchmark of 20 RESTful services, the results may not generalize. We note, nevertheless, that our benchmark includes a diverse set of services, including services that have been used in prior evaluations of REST API testing techniques.

Another potential threat is that we ran the services locally, which may result in different behavior than a real-world deployment. For example, we set up a private Ethereum network instead of using the Ethereum main network for the ERC-20 service. However, we believe that this is unlikely to affect our results in any significant way. Furthermore, we manually checked that the locally-installed services behaved as expected, including for the ERC-20 deployment.

In our empirical study, we used the OpenAPI specifications provided with each web service in the benchmark. We noticed that, for some services, the specification is incomplete, which may limit the ability of the tools to achieve higher coverage or finding more bugs. To investigate the prevalence of this phenomenon, we investigated over 1,000 specifications available on https://apis.guru/ and checked them with Schemathesis; we found that a vast majority (almost 99%) of the specifications contain some mismatch. In other words, it seems to be a common situation to have imperfect specifications, so the ones we used are representative of what the tools would have to rely upon in real-world scenarios.

RESTest requires additional information over what is in OpenAPI specifications. Specifically, it requires inter-parameter dependencies information [56]. We therefore studied the parameters of each API endpoint for the web services in our benchmarks and added such dependencies (if any) as annotations to the OpenAPI specifications. Although it was simple to create such annotations, and it typically took us only a few minutes per specification, the quality of the specifications might have affected the performance of the tool. Unfortunately, this is an unavoidable threat when using RESTest.

Finally, EvoMasterWB requires, for each service under test, a test driver that performs various operations (e.g., starting and stopping the web service). Building such driver can require a non-trivial amount of manual effort. For the 10 web services we selected from previous work, we obtained the existing drivers created by the EvoMasterWB author. For the remaining 10 web services, however, we had to create a driver ourselves. Also in this case, this was a fairly trivial task, which we were able to complete in just a few minutes for each web service. And also in this case, the only way to avoid this threat would be to exclude EvoMasterWB from the set of tools considered, which would considerably decrease the value of our comparative study.

5 RELATED WORK

Several recent studies have compared the effectiveness of testing techniques for REST services. Concurrent to ours, in particular, are three studies performed by Corradini et al. [25], Martin-Lopez, Segura, and Ruiz-Cortés [59], and by Hatfield-Dodds and Dygalo [38].

Corradini et al. [25] compared four black-box testing tools in terms of tool robustness and API coverage [26]. Our study includes all the tools they used and six additional tools. The code coverage metrics used in our study directly measure the amount of service code exercised and are different from the API coverage criteria [57] computed by the Restats tool [26] and used in their study. For example, the parameter value metric can only be applied to Boolean or enumeration types. Moreover, our study performs a more thorough assessment of the techniques' bug-finding ability by counting the number of unique 5xx errors, failure points, and library failure points instead of simply checking whether a 5xx status code is returned, as was done in their study. Finally, we provide a detailed discussion of the strengths and weaknesses of the testing strategies employed by the tools and the implications of our findings for future technique and tool development in this area.

Martin-Lopez, Segura, and Ruiz-Cortés compared EvoMasterWB, RESTest, and a hybrid of the two on a set of four REST APIs. Our study is different in that we compared EvoMasterWB with a set of 9 black-box tools (including RESTest) on a benchmark of 20 APIs. They showed that RESTest achieves better coverage than EvoMasterWB and that their bug-finding abilities are comparable. This is different from our result, which indicates that EvoMasterWB outperforms RESTest in terms of both coverage achieved and number of failures triggered. We believe this is partially because their study was based on only four services, which are all from the RESTest's benchmark dataset. Also, the manual work performed for tool configuration could have affected the results. In our study, we tried to avoid this issue by following the configuration method provided in the tools' papers and manuals [12]. We also collected tools and services in a systematic way, as explained in Sections 3 and 4.1. Compared to their study, ours also contains a more detailed analysis of code coverage achieved and 500 errors found, as well as an additional discussion of the implications of our results.

In their study, Hatfield-Dodds and Dygalo [38] compared Schemathesis to other black-box tools. Similar to Corradini et al.'s work, their work does not contain an in-depth analysis of the code coverage achieved and of the errors found by the tools beyond counting the number of errors triggered. Rather than providing a comparison of existing tools to assess their effectiveness, their work focused on the evaluation of Schemathesis.

In Section 3, we provided a summary of 10 state-of-the-art techniques and tools that perform automated testing of REST APIs and that we considered in our evaluation. We also discussed why we did not include in our study some of the tools we found. We note that there are other techniques that also target REST API testing. Among these, the approach proposed by Segura et al. [74] leverages metamorphic relationships to produce test cases. We do not include this work in our study because their testing approach is not automated: it requires a tester to provide low-level testing details, including identifying metamorphic patterns and identifying problem-specific metamorphic relations, which is non-trivial. Godefroid, Lehmann, and Polishchuk [36] introduced an approach that performs differential testing to find regressions in an evolving specification and in the corresponding service. Chakrabarti and Rodriquez [24] developed an approach that tests the connectedness (i.e., resource reachability) of RESTful services. The approaches proposed by Chakrabarti and Kumar [23] and by Reza and Gilst [71] rely on an XML-format specification for REST API testing. The approaches developed by Seijas, Li, and Thompson [50], by Fertig and Braun [31], and by Pinheiro, Endo, and Simao [63] perform model-based testing. We did not include these techniques in our study because they use a tool-specific REST API specification [23, 71] that requires non-trivial information to be provided by the user, rely on formal specifications [31, 50, 63], or target specific problems [24, 36].

Recently, after this work was finalized, two additional techniques and tools were proposed that perform combinatorial testing [85] and model-based testing [52] of RESTful APIs. Although we could not consider them in our current comparison, we will evaluate these tools for inclusion in an extended version of this work.

Numerous tools and libraries provide support for REST API testing, such as Postman [64], REST Assured [68], ReadyAPI [67], and API Fortress [2]. We did not include them in our study because they do not support automated testing. There are also tools that focus on tracking, fuzzing, and replaying API traffic such as AppSpider [5] and WAS [82]. We did not include these tools as they target fuzzing and require pre-recorded API usage information.

Finally, there exist many techniques and tools designed to test SOAP web services with WSDL-based specifications (e.g., [16, 18, 53, 75, 78]) and more broadly target the testing of service-oriented architectures (e.g., [20, 21]). We do not include these techniques and tools in our study, as we focus on REST API testing.

6 CONCLUSION AND FUTURE WORK

To gain insights into the effectiveness of existing REST API testing techniques and tools, we performed an empirical study in which we applied 10 state-of-the-art techniques to 20 RESTful web services and compared them in terms of code coverage achieved and unique failures triggered. We presented the results of the study, along with an analysis of the strengths and weaknesses of the techniques, summarized the lessons learned, and discussed implications for future research. Our experiment infrastructure, data, and results are publicly available [12]. In future work, we will extend our evaluation by using mutation to further evaluate the tools' fault-detection ability. We will also leverage the insights gained from our study and preliminary investigations to develop new techniques for testing REST APIs that perform better input parameter generation and consider dependencies among operations. Specifically, we will investigate ways to extract meaningful input values from the API specification and server logs, study the application of symbolic analysis to extract relevant information from the code, and research the use of NLP-based techniques to infer producer-consumer dependencies between operations.

DATA-AVAILABILITY STATEMENT

Data and code for reproducing our results are available on Zenodo [45]. Updated information about the project can be found at https://bit.ly/RESTTestToolsStudy.

ACKNOWLEDGMENTS

This work was partially supported by NSF, under grant CCF-0725202, DARPA, under contract N66001-21-C-4024, DOE, under contract DE-FOA-0002460, and gifts from Facebook, Google, IBM Research, and Microsoft Research.

REFERENCES

[1] APIBlueprint 2021. API Blueprint. https://apiblueprint.org/ Accessed: Jun 3, 2022.
[2] APIFortress 2022. API Fortress. https://apifortress.com Accessed: Jun 3, 2022.
[3] APIFuzzer 2022. APIFuzzer. https://github.com/KissPeter/APIFuzzer Accessed: Jun 3, 2022.
[4] apisguru 2022. APIs.guru API Directory. https://apis.guru/ Accessed: Jun 3, 2022.
[5] AppSpider 2022. AppSpider. https://www.rapid7.com/products/appspider Accessed: Jun 3, 2022.
[6] Andrea Arcuri. 2017. RESTful API automated test case generation. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, Prague, Czech Republic, 9–20.
[7] Andrea Arcuri. 2018. Evomaster: Evolutionary multi-context automated system test generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Västerås, Sweden, 394–397.
[8] Andrea Arcuri. 2019. Many Independent Objective (MIO) Algorithm for Test Suite Generation. *CoRR* abs/1901.01541 (2019), 3–17. arXiv:1901.01541 http://arxiv.org/abs/1901.01541
[9] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 1–37.
[10] Andrea Arcuri. 2020. Automated Black-and White-Box Testing of RESTful APIs With EvoMaster. *IEEE Software* 38, 3 (2020), 72–78.
[11] Andrea Arcuri and Juan P. Galeotti. 2019. SQL data generation to enhance search-based system testing. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO July 13-17, 2019*, Anne Auger and Thomas Stützle (Eds.). ACM, Prague, Czech Republic, 1390–1398. https://doi.org/10.1145/3321707.3321732
[12] Artifact 2022. Companion page with experiment infrastructure, data, and results. bit.ly/RESTTestToolsStudy Accessed: Jun 3, 2022.
[13] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations. arXiv:2005.11498 [cs.SE]
[14] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 748–758.
[15] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020. Checking Security Properties of Cloud Service REST APIs. In *13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Porto, Portugal, 387–397.
[16] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. 2005. WSDL-based automatic test case generation for web services testing. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE)*. IEEE, Beijing, China, 207–212.
[17] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
[18] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. 2009. WS-TAXI: A WSDL-based testing tool for web services. In *International Conference on Software Testing Verification and Validation (ICST)*. IEEE, Denver, CO, USA, 326–335.
[19] bBOXRT 2022. bBOXRT. https://git.dei.uc.pt/cnl/bBOXRT Accessed: Jun 3, 2022.
[20] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. 2013. Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability* 23, 4 (2013), 261–313.
[21] Gerardo Canfora and Massimiliano Di Penta. 2007. Service-oriented architectures testing: A survey. In *Software Engineering*. Springer, Berlin, Heidelberg, 78–105.
[22] Cats 2022. Cats. https://github.com/Endava/cats Accessed: Jun 3, 2022.
[23] Sujit Kumar Chakrabarti and Prashant Kumar. 2009. Test-the-rest: An approach to testing restful web-services. In *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*. IEEE, Athens, Greece, 302–308.
[24] Sujit Kumar Chakrabarti and Reswin Rodriquez. 2010. Connectedness Testing of RESTful Web-Services. In *Proceedings of the 3rd India Software Engineering Conference* (Mysore, India) *(ISEC '10)*. Association for Computing Machinery, New York, NY, USA, 143–152. https://doi.org/10.1145/1730874.1730902
[25] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Empirical comparison of black-box test case generation tools for RESTful APIs. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Luxembourg, 226–236.
[26] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Restats: A test coverage tool for RESTful APIs. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Luxembourg, 594–598.
[27] Davide Corradini, Amedeo Zampieri, Michele Pasqua, Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2022. Automated black-box testing of nominal and error scenarios in RESTful APIs. *Software Testing, Verification and Reliability* (2022), e1808.
[28] Dredd 2022. Dredd. https://github.com/apiaryio/dredd Accessed: may 1, 2022.
[29] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic generation of test cases for REST APIs: a specification-based approach. In *22nd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 181–190.
[30] EvoMaster 2022. EvoMaster. https://github.com/EMResearch/EvoMaster Accessed: Jun 3, 2022.
[31] Tobias Fertig and Peter Braun. 2015. Model-driven testing of restful apis. In *Proceedings of the 24th International Conference on World Wide Web*. 1497–1502.
[32] Roy T Fielding. 2000. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine.
[33] David Freedman, Robert Pisani, and Roger Purves. 2007. *Statistics (international student edition)*. WW Norton & Company.
[34] Gavel 2022. Gavel. https://github.com/apiaryio/gavel.js Accessed: Jun 3, 2022.
[35] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 725–736.
[36] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 312–323.
[37] GotSwag 2018. GotSwag. https://github.com/mobilcom-debitel/got-swag Accessed: Jun 3, 2022.
[38] Zac Hatfield-Dodds and Dmitry Dygalo. 2021. Deriving Semantics-Aware Fuzzers from Web API Schemas. *arXiv preprint arXiv:2112.10328* (2021).
[39] Hypothesis 2022. Hypothesis. https://hypothesis.works/ Accessed: Jun 3, 2022.
[40] IDLReasoner 2022. IDLReasoner. https://github.com/isa-group/IDLReasoner Accessed: May 1, 2022.
[41] JaCoCo 2021. JaCoCo. https://www.eclemma.org/jacoco/ Accessed: Jun 3, 2022.
[42] Michael I Jordan and Tom M Mitchell. 2015. Machine learning: Trends, perspectives, and prospects. *Science* (2015), 255–260.
[43] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. Automatic Property-based Testing of GraphQL APIs. *arXiv preprint arXiv:2012.07380* (2020).
[44] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. In *13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 131–141.
[45] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. *Automated Test Generation for REST APIs: Replication Package*. https://doi.org/10.5281/zenodo.6534554
[46] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
[47] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. 1997. Comparing operating systems using robustness benchmarks. In *Proceedings of 16th IEEE Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 72–79.
[48] Sandra Kübler, Ryan McDonald, and Joakim Nivre. 2009. Dependency parsing. *Synthesis lectures on human language technologies* (2009), 1–127.
[49] D Richard Kuhn, Raghu N Kacker, and Yu Lei. 2013. *Introduction to combinatorial testing*. CRC press.
[50] Pablo Lamela Seijas, Huiqing Li, and Simon Thompson. 2013. Towards property-based testing of RESTful web services. In *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*. 77–78.
[51] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A Black Box Tool for Robustness Testing of REST Services. *IEEE Access* (2021), 24738–24754.
[52] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: Model-based RESTful API Testing with Execution Feedback. *arXiv preprint arXiv:2204.12148* (2022).
[53] Chunyan Ma, Chenglie Du, Tao Zhang, Fei Hu, and Xiaobin Cai. 2008. WSDL-based automated test data generation for web service. In *2008 International Conference on Computer Science and Software Engineering*, Vol. 2. IEEE, 731–737.
[54] Christopher Manning and Hinrich Schutze. 1999. *Foundations of statistical natural language processing*. MIT press.
[55] Alberto Martin-Lopez, Sergio Segura, Carlos Muller, and Antonio Ruiz-Cortes. 2021. Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. *IEEE Transactions on Services Computing* (2021), 1–1. https://doi.org/10.1109/TSC.2021.3050610
[56] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. A catalogue of inter-parameter dependencies in RESTful web APIs. In *International Conference on Service-Oriented Computing*. Springer, 399–414.
[57] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. Test Coverage Criteria for RESTful Web APIs. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 15–21.

[58] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In *International Conference on Service-Oriented Computing*. Springer, 459–475.

[59] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?. In *Proceedings of the 32nd International Symposium on Software Reliability Engineering*. to appear.

[60] Sam Newman. 2015. *Building Microservices* (1st ed.). O'Reilly Media.

[61] NLTK 2021. NLTK. https://www.nltk.org/ Accessed: Jun 3, 2022.

[62] OpenAPI 2022. OpenAPI Specification. https://swagger.io/specification/ Accessed: Jun 3, 2022.

[63] Pedro Victor Pontes Pinheiro, Andre Takeshi Endo, and Adenilso Simao. 2013. Model-based testing of RESTful web services using UML protocol state machines. In *Brazilian Workshop on Systematic and Automated Software Testing*. Citeseer, 1–10.

[64] Postman 2022. Postman. https://getpostman.com Accessed: Jun 3, 2022.

[65] progweb 2022. ProgrammableWeb API Directory. https://www.programmableweb.com/category/all/apis Accessed: Jun 3, 2022.

[66] raml 2022. RESTful API Modeling Language. https://raml.org/ Accessed: Jun 3, 2022.

[67] ReadyAPI 2022. ReadyAPI. https://smartbear.com/product/ready-api/overview/ Accessed: Jun 3, 2022.

[68] RESTAssured 2022. REST Assured. https://rest-assured.io Accessed: Jun 3, 2022.

[69] RESTest 2022. RESTest. https://github.com/isa-group/RESTest Accessed: Jun 3, 2022.

[70] RESTler 2022. RESTler. https://github.com/microsoft/restler-fuzzer Accessed: Jun 3, 2022.

[71] Hassan Reza and David Van Gilst. 2010. A framework for testing RESTful web services. In *2010 Seventh International Conference on Information Technology: New Generations*. IEEE, 216–221.

[72] Muhammad Saad, Jeffrey Spaulding, Laurent Njilla, Charles Kamhoua, Sachin Shetty, DaeHun Nyang, and Aziz Mohaisen. 2019. Exploring the attack surface of blockchain: A systematic overview. *arXiv preprint arXiv:1904.03487* (2019).

[73] schemathesis 2022. schemathesis. https://github.com/schemathesis/schemathesis Accessed: Jun 1, 2022.

[74] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2017. Metamorphic testing of RESTful web APIs. *IEEE Transactions on Software Engineering (TSE)* (2017), 1083–1099.

[75] Harry M Sneed and Shihong Huang. 2006. Wsdltest-a tool for testing web services. In *2006 Eighth IEEE International Symposium on Web Site Evolution (WSE'06)*. IEEE, 14–21.

[76] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2021. Improving Test Case Generation for REST APIs Through Hierarchical Clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 117–128.

[77] tcases 2022. tcases restapi tool. https://github.com/Cornutum/tcases/tree/master/tcases-openapi Accessed: Jun 3, 2022.

[78] Wei-Tek Tsai, Ray Paul, Weiwei Song, and Zhibin Cao. 2002. Coyote: An xml-based framework for web services testing. In *Proceedings of 7th IEEE International Symposium on High Assurance Systems Engineering*. IEEE, 173–174.

[79] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RestTestGen: automated black-box testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 142–152.

[80] Diba Vosta. 2020. Evaluation of the t-wise Approach for Testing REST APIs.

[81] Atro Voutilainen. 2003. Part-of-speech tagging. *The Oxford handbook of computational linguistics* (2003), 219–232.

[82] WAS 2022. Qualys Web Application Scanning (WAS). https://www.qualys.com/apps/web-app-scanning/ Accessed: Jun 3, 2022.

[83] Jonathan J Webster and Chunyu Kit. 1992. Tokenization as the initial phase in NLP. In *COLING 1992 Volume 4: The 14th International Conference on Computational Linguistics*.

[84] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[85] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial Testing of RESTful APIs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*.

[86] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2021. Resource and dependency based test case generation for RESTful Web services. *Empirical Software Engineering* (2021), 1–61.

[87] Xiaojin Zhu and Andrew B Goldberg. 2009. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning* (2009), 1–130.