Open Problems in Fuzzing RESTful APIs: A Comparison of Tools

MAN ZHANG, Kristiania University College, Norway
ANDREA ARCURI, Kristiania University College and Oslo Metropolitan University, Norway

RESTful APIs are a type of web service that are widely used in industry. In the last few years, a lot of effort in the research community has been spent in designing novel techniques to automatically fuzz those APIs to find faults in them. Many real faults were automatically found in a large variety of RESTful APIs. However, usually the analyzed fuzzers treat the APIs as black-box, and no analysis of what is actually covered in these systems is done. Therefore, although these fuzzers are clearly useful for practitioners, we do not know what are their current limitations and actual effectiveness. Solving this is a necessary step to be able to design better, more efficient and effective techniques. To address this issue, in this paper we compare seven state-of-the-art fuzzers on 18 open-source, one industrial and one artificial RESTful APIs. We then analyzed the source code of which parts of these APIs the fuzzers fail to generate tests for. This analysis points to clear limitations of these current fuzzers, listing concrete challenges for the research community to follow up on.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; **Search-based software engineering**.

Additional Key Words and Phrases: Automated test generation, SBST, fuzzing, REST, comparison

ACM Reference Format:
Man Zhang and Andrea Arcuri. 2022. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. 1, 1 (April 2022), 46 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

RESTful APIs are widely used in industry. Arguably, REST is the most common kind of web service, used to provide functionality (i.e., an API) over a network. Many companies worldwide provide services via a RESTful API, like Google [8], Amazon [1], LinkedIn [12], Twitter [23], Reddit [15], etc. The website *ProgrammableWeb* [14] currently lists more than 24 thousand APIs available on internet, where the vast majority of them is implemented with REST. Furthermore, REST APIs are also commonly used when developing backend enterprise applications using a microservice architecture [66].

Due to their wide use in industry, in recent years there has been a large interest in the research community [49] to design novel techniques to automatically test this kind of applications (e.g., [28, 38, 56, 62, 68, 69]). Compared to other kinds of applications (e.g., data parsers) and types of testing (e.g., unit testing), system testing of RESTful APIs has some unique challenges, like dealing with network communications (e.g., HTTP over TCP) and with accesses to databases (e.g., Postgres and MySQL).

Authors' addresses: Man Zhang, man.zhang@kristiania.no, Kristiania University College, Oslo, Norway; Andrea Arcuri, andrea.arcuri@kristiania.no, Kristiania University College and Oslo Metropolitan University, Oslo, Norway.

XXXX-XXXX/2022/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In the literature, different techniques have been proposed [49], which have been able to automatically find many faults in existing APIs (e.g., [28, 38, 56, 62, 68, 69]). However, so far, most of the investigated techniques are black-box, applied on remote web services on the internet. Although those fuzzers have been useful to detect several faults, it is unclear how effective they truly are at testing these APIs. For example, most studies do not report any form of code coverage, as the system under tests (SUTs) are typically remote black-boxes with no access to their source code. For example, it can be possible that most found faults are simply in the first layer of input validation of these SUTs, with little-to-no code of their businesses logic executed [59]. Without an in-depth analysis of which parts of the SUT code is not exercised by these fuzzers, it is not possible to understand their limitations. This hinders further research development for novel techniques to achieve better results.

To address these issues, in this paper we compare seven state-of-the-art fuzzers, namely (in alphabetic order) bBOXRT [56], EvoMaster [28], RESTest [62], RestCT [69], RESTler [38], RestTestGen [68] and Schemathesis [52]. We applied them on 13 RESTful APIs running on the JVM (i.e., 12 open-source and one industrial) and six open-source RESTful APIs running on NodeJS, for which we collected line coverage metrics using JaCoCo [10] (for JVM) and c8 [5] (for NodeJS). Experiments are carried out both for black-box and white-box testing. To compare the effectiveness of black-box fuzzers at finding faults, also a new artificial REST API was implemented specifically for this study.

This large set of experiments (which takes roughly 120 days if run in sequence) enables us to analyze in detail what are the current limitations of the state-of-the-art in fuzzing RESTful APIs. These tools achieve different degrees of code coverage, but there are still many issues that need to be addressed to achieve better results. These include for example how to deal with underspecified schemas, and how to deal with interactions with external services.

In this paper, we provide the following novel research contributions:

- We carried out one of the largest and most variegated to date empirical comparison of fuzzers for RESTful APIs.
- Instead of using black-box metrics, we report actual coverage results on the employed APIs.
- We provide in-depth analyses of the current challenges in this testing domain.

The paper is organized as follows. Section 2 provides background info on the seven compared tools. Related work is discussed in Section 3. Section 4 presents the details of the tool comparisons. Section 4.7 provides the main scientific contribution of this paper, with an analysis of the current open-problems identified by our empirical study (with in depth-analyses on each studied API provided in Appendix). Threats to validity are discussed in Section 5. Finally, Section 6 concludes the paper.

2 BACKGROUND: USED TOOLS

In this paper, we compare seven different fuzzers for RESTful APIs, namely bBOXRT, EvoMaster, RESTest, RestCT, RESTler, RestTestGen and Schemathesis. Here in this section they are briefly described, in alphabetic order. For the tool comparisons, we used their latest released versions, as per 23rd of May 2022.

To the best of our knowledge, this selection represents the current state-of-the-art in fuzzing RESTful APIs, as those are the most used, most popular (e.g., number of stars and download stats from GitHub) and cited work in the literature. For example, some tools in the literature like QuickRest [54] do not seem available online, whereas others like ApiTester [46] have no documentation and have not been updated in years [3]. The recent tool Morest [57] is not open-source, and its replication package seems that it provides no executable nor code to replicate its experiments[1], only the data analysis. So, no comparison seems possible with these other tools. In addition, there exist some open-source fuzzers made by practitioners in industry, such as APIFuzzer [2], Dredd [11] and Tcases [22]. However, since these fuzzers do not appear in the scientific literature (i.e., they do not have scientific articles describing the techniques they use to generate test cases), and considering that in a previous study [55] these tools achieve worse results compared to the other fuzzers, we do not include them in this study.

All these tools require a OpenAPI/Swagger [13] schema to operate. Based on such a schema, these tools send syntactically valid HTTP requests, using different strategies (e.g., random and search-based) to choose how to create the input data (e.g., query parameters and JSON body payloads).

bBOXRT [56] aims at black-box robustness testing of RESTful APIs. The tool is written in Java, and its source code is available online on the authors' institute pages [4], since 2020. We could not find any released version, so we used the latest version on its Git master branch (commit *7c894247*).

EvoMaster [28--31, 33--35, 72, 76] is a search-based tool that uses evolutionary computation to generate test cases for RESTful APIs. It supports both black-box and white-box testing [31] (but this latter only for programs running on the JVM and JavaScript/TypeScript). The tool is written in Kotlin, and open-sourced on GitHub [6], since 2016. For this study, we used version *1.5.0*. We are the authors of this tool.

RESTest [61--64, 67] is black-box testing tool, open-sourced on GitHub [17], since 2018. One of the main features of this tool is the ability to analyze inter-dependencies among input parameters. The tool is written in Java. We used its *1.2.0* version.

RestCT [69] uses Combinatorial Testing for black-box testing of RESTful APIs. It is open-sourced on GitHub [16], since 2022. The tool is written in Python. We used its *1.0* version.

RESTler [38, 39, 47, 48] is a black-box fuzzer for RESTful APIs. It is open-sourced on GitHub [18], since 2020. The tool is written in Python and F#. It does not have any published release, although its Git repository has tagged commits. We use the version of the repository with the latest tag *v8.5.0*.

RestTestGen [45, 68] is another black-box fuzzer for RESTful APIs. It is written in Java, but the original versions used in [68] are not open-source. Since late 2021, a new rewrite of the tool as open-source is available on Github [19]. Then, in this study, we use the version with the latest tag *v22.01*.

Schemathesis [52] is a black-box fuzzer which employs property-based testing techniques [58], whose development started in 2019. The fuzzer is capable of deriving structure and semantics of SUTs based on their API schemas, such as OpenAPI [13]. It is written in Python, and can be installed with `pip`. In this study, we use the version *3.15.2* [21].

These seven tools have different levels of maturity, usability and quality of documentation. On the one hand, achieving high code coverage and fault detection might be not so important for practitioners if a tool cannot be used because it is too complex to set up and does not have documentation. On the other hand, most of these tools are academic prototypes, primary meant to study research questions, and not for wide use by practitioners.

As we are the authors of EvoMaster, we are too biased to comment on its maturity and usability. We can only state that it is the oldest of these seven, and it is actively maintained.

Among the other six tools, Schemathesis is clearly the one that puts most emphasis on usage by practitioners, with its user-friendliness (e.g., GitHub Actions support) and extensive documentation. Tools like RESTler and RestTestGen for example require to clone their Git repositories and build the tools locally (e.g., in contrast to Schemathesis that can be installed with `pip`, or EvoMaster that provides installers for all major operating systems, such as `.msi` for Windows). RESTest

[1] https://anonymous.4open.science/r/morest-rest-8CAE/ currently gives a ''The repository is expired'' error

provides released jars and useful documentation, but in our experiments it was not as user-friendly as Schemathesis (e.g., it requires few manual steps to set it up). At the time of this writing, the other tools (i.e., bBOXRT, and RestCT) have not been updated for several months (according to the commit history in their Git repositories). Therefore, they might not be longer maintained.

3 RELATED WORK

To achieve better results (e.g., higher code coverage and fault detection), we need to understand what are the limitations of current testing techniques. Throughout the years, different studies have been carried out to provide this insight, like for example studying the limitations of Dynamic Symbolic Execution (DSE) [70, 71] and Search-Based Software Testing (SBST) for unit testing [25, 27, 51]. However, to the best of our knowledge, no work has been carried out to study what are the limitations of fuzzing RESTful APIs (most work is for black-box testing, where achieved code coverage is usually not reported, and no analysis of what was not covered is carried out).

Regarding tool comparisons for fuzzing RESTful APIs, there has been some work in the literature. For example, the authors of RestTestGen compared four black-box tools (RestTestGen, RESTler, bBOXRT and RESTest) on 14 APIs [44]. These APIs are written in different languages (e.g., Java, C#, PHP, JavaScript and Go), with the largest having up to 24044 lines of code. However, no code coverage was reported. From this comparison, RestTestGen seems the most effective tool, whereas RESTler is the most robust (i.e., could be used on more APIs without crashing).

The authors of RESTest and EvoMaster compared the two tools [60], aiming at studying the tradeoffs between black-box and white-box testing. For example, they studied the impact on performance of using ‘‘custom generators’’, with test data provided by the users.

The authors of RestCT, when they introduced their tool [69], compared it with RESTler on 11 APIs coming from two projects (i.e., GitLab and Bing Maps), showing better results for RestCT.

The authors of Morest [57] compared it against with black-box EvoMaster, RestTestGen and RESTler, on 6 APIs. They claimed Morest gives the best results. However, as Morest is not available for comparisons (in contrast to EvoMaster, RestTestGen and RESTler), such claims cannot be independently verified.

In previous work, we compared EvoMaster's black-box and white-box mode on 8 SUTs [31], 7 open-source and 1 industrial, showing better results for its white-box mode. The SUTs used in this paper are a super-set of the open-source SUTs used in [31] (i.e., for the experiments on the JVM we used the same SUTs plus another 5).

In parallel, at the same time of the first arXiv version of this paper [73], Kim et al. [55] made a comparison of tools for fuzzing RESTful APIs. A total of different 9 tools (the same as here, minus the recent RestCT, but plus the aforementioned APIFuzzer [2], Dredd [11] and Tcases [22]) were compared on 20 APIs running on the JVM (where half of them are the same as here, which come from our own EMB [7] repository of APIs we use to experiment with EvoMaster). To better differentiate from this existing work that was done in parallel, and to better generalize our results, here we include as well 6 JavaScript/TypeScript APIs running on NodeJS, as well one industrial API coming from one of our industrial partners. Furthermore, where the focus of [55] seems to be the comparison of tools, our focus is on the in-depth analysis of the *open problems* in fuzzing RESTful APIs (Section 4.7). The comparison of tools is only a means to identify the one that gives the best results, as the tests generated by such tool are the starting point of the in-depth analyses. In both studies [55, 73] black-box EvoMaster gives the best results, closely followed by Schemathesis, and white-box testing gives better results than black-box. Kim et al. [55] do not seem to be authors of any of the compared tools. On the one hand, their comparison is therefore unbiased. This is different from our case, as we are the authors of EvoMaster, which turned out to be the best in all these comparisons. One should always be a bit skeptical of studies where the tool of the authors gives the best results, especially if such results are not possible to be replicated by third-parties. This is one of the main reasons why EvoMaster is open-source, with all its experiment scripts stored in its Git repository, automatically updated on Zenodo for long term storage at each new release (e.g., version 1.5.0 [36]), as well preparing all the SUTs for experiments in a single repository (i.e., EMB [7]). On the other hand, being the authors of the most performant tool gives us a unique insight when analyzing and discussing the current open problems in fuzzing RESTful APIs (Section 4.7).

4 EMPIRICAL ANALYSIS

In this paper, we aim at answering the following research questions:

- **RQ1:** How do the seven compared black-box fuzzers fare in terms of line coverage?
- **RQ2:** What is the impact of the time budget on the performance of the black-box fuzzers?
- **RQ3:** What kind of faults can the compared black-box fuzzers find?
- **RQ4:** What line coverage and fault detection results are obtained with white-box fuzzing?
- **RQ5:** What are the main open problems currently hindering the results?

To answer these research questions, we carried out 3 different sets of experiments, on the case study described in Section 4.1. First set is for black-box testing aimed at code coverage (**RQ1** and **RQ2**, in Section 4.3 and 4.4, with common setup described in Section 4.2), then for fault finding (**RQ3**, Section 4.5), and then for white-box testing (**RQ4**, Section 4.6). From the results of these experiments, the in-depth analysis of the results for **RQ5** follows in Section 4.7.

4.1 Case Study

Table 1. Statistics on 18 RESTful APIs from EMB [7] and one industrial API.

SUT	Language	Endpoints	Files	File LOCs	c8/JaCoCo LOCs
cyclotron	JavaScript	50	25	5803	2458
disease-sh-api	JavaScript	34	57	3343	2997
js-rest-ncs	JavaScript	6	8	775	768
js-rest-scs	JavaScript	11	13	1046	1044
realworld-app	TypeScript	19	37	1229	1077
spacex-api	JavaScript	94	63	4966	3144
catwatch	Java	23	106	9636	1835
cwa-verification	Java	5	47	3955	711
features-service	Java	18	39	2275	457
gestaohospital-rest	Java	20	33	3506	1056
ind0	Java	20	75	5687	1674
languagetool	Java	2	1385	174781	45445
ocvn-rest	Java	258	526	45521	6868
proxyprint	Java	115	73	8338	2958
rest-ncs	Java	6	9	605	275
rest-news	Kotlin	7	11	857	144
rest-scs	Java	11	13	862	295
restcountries	Java	22	24	1977	543
scout-api	Java	49	93	9736	2673
rest-faults	Java	8	3	115	26
*Total (X, Y)**		778 (214,564)	2640 (203,2437)	285013 (17162,267851)	76448 (11488,64960)

* Note that, regarding *Total(X,Y)*, *Total* represents the total of the statistic on all case studies, whereas *X* represents the total on just the JavaScript/TypeScript APIs, and *Y* represents the total on the JVM APIs.

To carry out experiments in this paper, we used a collection of 19 RESTful APIs plus one artificial case study named *rest-faults* for the experiment on fault detection. Table 1 shows the statistics of these 20 APIs.

As we need to measure code coverage and analyze the source code to check which parts are not covered, we needed open-source APIs that we could run on a local machine. Furthermore, to simplify the collection of code coverage results, it is easier to use APIs written in the same programming language (e.g., Java), or at least use not too many different languages, as each programming language would need to configure its own code-coverage tool to analyze the test results. Considering that we wanted to do comparisons also with white-box testing, which currently only EvoMaster supports, and that requires some manual configurations (e.g., to set up bytecode instrumentation for the white-box heuristics), we decided to use the same case study that we maintain for EvoMaster. In particular, we maintain a repository of RESTful APIs called EMB [7], which is stored as well on Zenodo [37]. Note that one of these APIs is coming from one of our industrial partners, which of course we are not allowed to store on EMB.

These 19 APIs are written in four different languages: Java, Kotlin, JavaScript and TypeScript. They run on 2 different environments/runtimes: the JVM and NodeJS. For each SUT, we report the number of total source files (i.e., ending in either `.java`, `.kt`, `.js` or `.ts`), and their number of lines (LOCs). As these include as well import statements, empty lines, comments and tests, for the APIs running on the JVM we also report the number of actual line targets for code coverage (measured with the tool JaCoCo [10]).

For the APIs running on the NodeJS, the code coverage is measured with the tool c8, which uses native V8 coverage. By default, the tool c8 will count code coverage only for the files which are loaded by the engine [5]. For instance, regarding *cyclotron*, different LOCs between Files and c8 are due to unreachable files (i.e., `api.analyticselasticsearch.js`, `api.analytics.js`, `api.statistics-elasticsearch.js`). However, all of the files will be only reached by manually modifying a configuration in `config.js`, i.e., the default value is `false`. Therefore, we report the number of line targets measured by c8 that could be loaded with the default SUT settings.

For the experiments on fault detection for the black-box fuzzers (Section 4.5), we created a small artificial API, written in Java, with 10 seeded faults. This API is open-source, currently available on GitHub[2] .

4.2 Black-Box Testing Experiment Settings: Code Coverage

For each SUT, we created Bash scripts to start them, including any needed dependency (e.g., as *ocvn-rest* uses a MongoDB database, this is automatically started with Docker). Each SUT is started with either JaCoCo (for JVM) or c8 (for NodeJS) instrumentation, to be able to collect code coverage results at the end of each tool execution.

Each of the seven compared tools was run on each of the 19 SUTs, repeated with different seeds a certain amount of times (e.g., repeated 10 times for a budget setting of 1 hour), to keep into account the randomness of these tools. Each script starts a SUT as a background process, and then one of the tools. Each script runs the SUT on a different TCP port, to enable running any of these scripts in parallel on the same machine.

The code coverage is computed based on all the HTTP calls done during the fuzzing process, and not on the output of the generated test files (if any). This was done for several reasons: not all tools generate test cases in JUnit on JavaScript format, the generated tests might not compile (i.e., bugs in the tools), and setting up the compilation of the tests and running them for collecting coverage would be quite challenging to automate (as each tool behaves differently). This also means that, if a

[2] https://github.com/EMResearch/rest-faults

tool crashes, we are still measuring what code coverage it achieves. If a tool crashes immediately at startup (e.g., due to failures in parsing the OpenAPI/Swagger schemas), we are still measuring the code coverage achieved by the booting up of the SUT.

All the experiments for this paper were run on a Windows 10 machine with a processor Intel(R) Xeon(R), 2.40GHz, 24 Cores with 192G of RAM. To avoid potential issues in running jobs in parallel (e.g., exhausting the resources of the operating system, which could lead to possible timeout issues), we only ran 8 jobs at a time. This means that each experiment had 3 cores (6 virtual threads) and 24G of RAM.

Regarding the selected seven fuzzers, there exist fuzzers which do not provide an option to configure a global time budget to terminate fuzzing (e.g., Schemathesis [20] and RESTest [17]). Also, although some fuzzers provide the option of a timeout (e.g., RestTestGen), they might terminate much earlier than the specified timeout value [45]. To make the comparison of the fuzzers more fair by applying the same time budget, given the same time budget X (e.g., 1 hour), for each fuzzer we run it in a loop with the same X timeout (i.e., if the fuzzer runs out of time, it would be terminated, and if the fuzzer completes but there is still some time remaining, it would be re-started to generate more tests). Thus, all coverage we collected are based on the same time budget for all the fuzzers.

To compare these tools, we use the line coverage reported by JaCoCo and c8 as metric. Another important metric would be fault detection. However, how to compute fault detection in unbiased way, applicable for all compared tools, is far from trivial. Each tool can self-report how many faults they find, but how such fault numbers are calculated might be very different from tool to tool, making any comparison nearly meaningless. Manually checking (possibly tens of) thousands of test cases is not a viable option either. Therefore, for this type of experiments line coverage was the most suitable metric for the comparisons. Still, fault detection is a very important metric needed to properly compare fuzzers. For the black-box fuzzers, this will be evaluated with an ad-hoc API, as explained in more details in Section 4.5. For the white-box experiments (Section 4.6), this is not a problem, as we use the same tool (i.e., EvoMaster).

Regarding experiment setup, as a black-box testing tool for fuzzing REST APIs, all tools are required to configure where the schema of the API is located. In all these SUTs used in our case study, the schemas are provided by the SUTs directly via an HTTP endpoint. But, we found that most of the tools do not support to fetch the schema with a given URL, such as http://localhost:8080/v2/api-docs. To conduct the experiments with these tools, after the SUT starts, we developed a Bash script which manages to fetch the schema and download it to the local file system, and then configure a file-path for the tools to specify where the schema can be accessed.

Regarding additional setups to execute the tools, EvoMaster, RestCT and Schemathesis were the simplest to configure, as they require only one setup step, as all of their options can be specified with command-line arguments. However, RestCT currently does not work directly on Windows [16]. So, for these experiments, we simply ran it via the Windows Subsystem for Linux (WSL). This might introduce some time delays compared to running it directly on a Linux machine. However, RestCT is the only tool that has constraints on the type of operating system it can run on, and practitioners in industry that use Windows would have to run it with WSL (or Docker) as well. RestTestGen requires having a JSON file (i.e., `rtg_config.json`) to configure the tool with available options [19]. RESTler requires multiple setup steps, e.g., RESTler needs to generate grammar files first, and then employ such grammars for fuzzing. However, with its online available documentation, we could write a Python script about how to run the tool. For RESTest, it requires a pre-step to generate a test configuration in order to employ the tool, and such generation could be performed automatically by a utility `CreateTestConf` provided by the tool. In order to use bBOXRT, a Java class file is required to load and set up the API specification.

At the time of writing this paper, there does not exist specific documentation about how to specify such Java class. However, in its open-source repository, there exist many examples that helped us to create these Java classes for the SUTs in our case study. Note that, for these experiments in this paper, all the above setups were performed automatically with our Bash scripts.

All the manual steps to configure the tools that we automated in our Bash scripts take some time. However, such time was not taken into account in the time budget (e.g., 1 hour) used in the experiments. In other words, each tool was run for the same amonut of time regardless of the time it takes to set it up. The are two main reasons for this. First, evaluating the cost of each manual step in a sound way would require empirical studies with human subjects, taking into account several different properties (e.g., experience/seniority of the participants, and familiarity with the existing fuzzers). But this kind of empirical study would be beyond the scope of this paper. Second, tool setups on an API are a one-time cost. The same API would be tested continuously throughout its life-cycle (e.g., on a Continuous Integration server), each time new changes/features are introduced. In this context, such one-time setup cost would be negligible. Furthermore, when dealing with several APIs to test (e.g., in a microservice architecture), it can well be that the setups are very similar, and it could be just a trivial matter of copy&paste when setting up the fuzzers for a new API (this is for example what we have experienced when applying EvoMaster in large companies such as Meituan [74]). As anyway those manual costs were small in terms of time (in order of minutes), but excluding the time to study the documentation of these tools, we do not consider this as a major threat to validity to our study.

The first time we ran the experiments, we could collect results only for EvoMaster, RESTler and Schemathesis. All the other tools failed to make any HTTP calls. This was due for example to mismatched schema format, or missing/misconfigured info in the schemas. More specifically, bBOXRT only allows a schema with YAML format. In the SUTs used in this study, there is only one specified with YAML (i.e., *restcountries*) out of the 19 schemas (the remaining ones use JSON). RestTestGen only accepts a schema with OpenAPI v3 in a JSON format [19]. There are only two (i.e., *cwa-verification* and *spacex-api*) out of the 19 SUTs which expose OpenAPI v3 in the JSON format. In addition, RESTest and RestTestGen need the protocol info (e.g., http or https with `servers/schemes` tag) in the OpenAPI/Swagger schema. But since the `servers/schemes` tag is not mandatory, such info might not be always available in the schema. For example, seven (i.e., *cyclotron*, *disease-sh-api*, *js-rest-ncs*, *js-rest-scs*, *realworld-app*, *features-service* and *restcountries*) out of the 19 SUTs have such protocol info specified in their schemas. In addition, to create HTTP requests, RestCT, RESTest and RestTestGen require info specified in `host` (for schema version 2) and `servers` (for schema version 3), but such info (typically related to TCP port numbers) might not be fully correct (e.g., the host and TCP port might refer to the production settings of the API, and not reflecting when the API is running on the local host on an ephemeral port for testing purposes). Those three tools do not seem to provide ways to override such info. For instance, in 19 SUTs , 10 SUTs (i.e., *cyclotron*, *disease-sh-api*, *js-rest-ncs*, *js-rest-scs*, *realworld-app*, *spacex-api*, *cwa-verification*, *features-service*, *languagetool* and *restcountries*) are specified with a hard-coded TCP port, and in one SUT (i.e., *scout-api*) the TCP port is unspecified. To avoid these issues in accessing the SUTs, we developed a utility using `swagger-parser` library that facilitates converting formats of schemas between JSON and YAML (only applied for bBOXRT and RestTestGen), converting OpenAPI v2 to OpenAPI v3 (only applied for RestTestGen), adding missing `schemes` info, and correcting/adding `host` and `servers` info in the schemas.

Once these changes in the schemas were applied, we repeated the experiments, to collect data from all the seven tools. Ideally, these issues should be handled directly by the tools. But, as they are rather minor and only required changes in the OpenAPI/Swagger schemas, we decided to address them, to be able to collect data from all the seven tools and not just from three of them.

In addition, we needed to configure authentication info for five APIs, namely *proxyprint*, *scout-api*, *ocvn-rest*, *realworld-app* and *spacex-api*. For *proxyprint*, *scout-api* and *spacex-api*, they need *static* tokens sent via HTTP headers. This was easy to setup in RestCT, EvoMaster and Schemathesis, just by calling these tools with a `--header` input parameter. RESTest required to rewrite the test configuration file by appending authentication configuration. RESTler and RestTestGen required to write some script configurations. bBOXRT has no documentation to setup authentication, but we managed to setup it up by studying the examples it provides in its repository.

Regarding *ocvn-rest* and *realworld-app*, for authentication, it requires to make a `POST` call to a form-login endpoint, and then use the received cookie in all following HTTP requests. Out of the seven compared tools, it seems RESTler, bBOXRT, Schemathesis, and RestTestGen could directly support this kind of authentication by setting up it with an executable script. Given the provided documentation, we did not manage to configure it, as it requires to write different scripts for different fuzzers to manually make such HTTP login calls then handle responses. Technically, by writing manual scripts, it could be possible to use EvoMaster, RestCT and RESTest as well, by passing the obtained cookie with `--header` option or the test configuration file. As doing all this was rather cumbersome, and considering that for this API the authentication is needed only for *admin* endpoints, we decided to do not spend significant time in trying to setup this kind *dynamic* authentication tokens.

Table 2. Experiment results of the seven black-box fuzzers on the 19 RESTful APIs. For each tool we report the average line coverage, as well as its [min,max] values out of the 10 runs. Each tool also has a *(rank)* based on its performance on each SUT. Ties are broken by reporting the average rank. For each SUT, the results of the best tools (e.g., rank (1)) are in bold. We also report the χ^2 and p-value of the Friedman test for variance analysis by the ranks.

SUT	bBOXRT	EvoMaster BB	RestCT	RESTler	RESTest	RestTestGen	Schemathesis
cyclotron	41.3 [41.3,41.3] (5)	**70.3 [70.1,72.3] (1)**	41.3 [41.3,41.3] (5)	41.3 [41.3,41.3] (5)	41.3 [41.3,41.3] (5)	41.3 [41.3,41.3] (5)	69.1 [67.7,69.7] (2)
disease-sh-api	56.4 [55.4,57.4] (3)	60.8 [60.8,60.9] (2)	48.4 [48.4,48.4] (6.5)	48.5 [48.5,48.5] (4)	48.5 [48.4,48.5] (5)	48.4 [48.4,48.4] (6.5)	**61.5 [61.4,61.6] (1)**
js-rest-ncs	70.2 [67.3,71.1] (5)	93.0 [89.8,95.8] (2)	92.2 [87.5,92.7] (3)	44.3 [44.3,44.3] (6.5)	44.3 [44.3,44.3] (6.5)	88.6 [85.8,93.0] (4)	**100.0 [100.0,100.0] (1)**
js-rest-scs	83.2 [83.0,83.5] (5)	**88.7 [87.5,89.5] (1)**	83.2 [83.1,83.2] (6)	54.1 [54.1,54.1] (7)	84.1 [83.3,84.6] (4)	86.4 [85.4,87.1] (2)	86.1 [85.9,87.4] (3)
realworld-app	64.2 [62.8,66.4] (4)	69.4 [69.4,69.4] (2)	59.7 [59.7,59.7] (6)	66.5 [66.5,66.5] (3)	59.7 [59.7,59.7] (6)	59.7 [59.7,59.7] (6)	**69.7 [69.1,69.8] (1)**
spacex-api	76.1 [76.1,76.2] (6)	84.7 [84.7,84.8] (2)	76.1 [76.1,76.2] (7)	76.3 [76.3,76.4] (3)	76.3 [76.3,76.3] (5)	76.3 [76.3,76.4] (4)	**85.4 [85.3,85.6] (1)**
catwatch	31.0 [29.2,31.8] (3)	**35.9 [34.3,36.9] (1)**	9.7 [9.7,9.7] (7)	17.3 [14.5,20.5] (5)	20.8 [15.9,23.8] (4)	15.1 [12.3,18.3] (6)	35.8 [33.6,39.1] (2)
cwa-verification	43.4 [42.9,43.5] (4)	49.4 [49.1,49.6] (2)	21.9 [21.9,21.9] (6)	21.9 [21.9,21.9] (6)	21.9 [21.9,21.9] (6)	43.8 [43.3,43.9] (3)	**49.5 [49.5,49.5] (1)**
features-service	35.7 [35.7,35.7] (4)	**59.7 [58.4,62.1] (1)**	21.0 [21.0,21.0] (6)	21.0 [21.0,21.0] (6)	21.0 [21.0,21.0] (6)	45.9 [45.1,46.8] (3)	52.0 [46.6,57.1] (2)
gestaohospital-rest	36.2 [36.2,36.2] (4)	50.8 [44.8,54.3] (3)	19.9 [19.9,19.9] (7)	21.5 [21.5,21.5] (6)	29.0 [28.1,32.0] (5)	57.2 [51.7,58.7] (2)	**58.7 [55.9,62.3] (1)**
ind0	**8.2 [8.2,8.2] (3)**	**8.2 [8.2,8.2] (3)**	7.6 [7.6,7.6] (6.5)	7.6 [7.6,7.6] (6.5)	**8.2 [8.2,8.2] (3)**	**8.2 [8.2,8.2] (3)**	**8.2 [8.2,8.2] (3)**
languagetool	1.7 [1.7,1.7] (6)	**32.6 [26.0,35.1] (1)**	1.5 [1.5,1.5] (7)	1.9 [1.9,1.9] (4.5)	2.5 [2.5,2.5] (2)	1.9 [1.9,1.9] (4.5)	2.2 [2.1,2.5] (3)
ocvn-rest	10.1 [10.1,10.1] (5)	**27.5 [27.5,27.6] (1)**	10.1 [10.1,10.1] (5)	10.1 [10.1,10.1] (5)	10.1 [10.1,10.1] (5)	10.1 [10.1,10.1] (5)	27.5 [27.5,27.7] (2)
proxyprint	4.2 [4.2,4.2] (5)	**34.0 [32.5,35.0] (1)**	4.2 [4.2,4.2] (5)	4.2 [4.2,4.2] (5)	4.2 [4.2,4.2] (5)	4.2 [4.2,4.2] (5)	4.4 [4.4,4.4] (2)
rest-ncs	55.0 [52.4,56.4] (5)	64.5 [64.4,64.7] (3)	85.5 [85.5,85.5] (2)	40.7 [40.7,40.7] (6)	5.1 [5.1,5.1] (7)	64.3 [64.0,64.7] (4)	**94.1 [93.1,94.5] (1)**
rest-news	34.9 [34.0,36.8] (5)	**69.4 [69.4,69.4] (1)**	13.9 [13.9,13.9] (6.5)	44.4 [44.4,44.4] (4)	13.9 [13.9,13.9] (6.5)	47.2 [47.2,47.2] (3)	68.8 [67.4,70.8] (2)
rest-scs	60.2 [59.3,61.4] (6)	**66.9 [64.4,70.2] (1)**	60.5 [60.3,61.0] (5)	58.3 [58.3,58.3] (7)	61.7 [61.4,62.4] (4)	65.3 [64.4,67.1] (2)	64.8 [64.4,65.1] (3)
restcountries	65.5 [63.7,68.5] (5)	**76.1 [76.1,76.1] (1)**	3.5 [3.5,3.5] (7)	50.6 [50.6,50.6] (6)	73.0 [71.5,74.2] (4)	75.4 [73.5,76.6] (2)	73.9 [72.4,75.0] (3)
scout-api	18.1 [18.1,18.1] (5)	**36.7 [32.8,41.1] (1)**	12.0 [12.0,12.0] (6.5)	26.5 [26.4,26.6] (2)	12.0 [12.0,12.0] (6.5)	23.0 [21.4,23.8] (4)	23.0 [22.1,25.1] (3)
Average	41.9 (4.6)	56.8 (1.6)	35.4 (5.8)	34.6 (5.1)	33.6 (5.0)	45.4 (3.9)	54.5 (1.9)
Friedman Test							χ^2 = **71.845**, p-**value** = **≤0.001**

4.3 RQ1: Black-Box Testing Experiments

To answer this research question, each of the seven compared tools was run on each of the 19 SUTs, repeated for 10 times to keep into account the randomness of these tools. This results in a total of $7 \times 19 \times 10 = 1330$ Bash scripts.

In each script, each tool was run for 1 hour, for a total of 1330 hours, i.e., 55.4 days of computation efforts. Note: the choice of the runtime for each experiment might impact the results of the comparisons. The choice of 1 hour is technically arbitrary, but based on what practitioners might want to use these fuzzers in practice, and also not too long to make running all these experiments unviable in reasonable time.

Table 2 shows the results of these experiments. For each tool, we report the average (i.e., arithmetic mean) line coverage, as well as the min and max values out of the 10 runs. Each tool is then ranked (1 to 7) based on their average performance on each SUT (where 1 is the best rank). A Friedman test is conducted to analyze the variance of these techniques, based on ranks of their performance on the SUTs.

From these results we can infer few interesting observations. First, regarding the ranking, EvoMaster seems the best black-box fuzzer (best in 11 out of 19 SUTs, with an average coverage of 56.8%,), closely followed by Schemathesis (best in 7 SUTs, with an average coverage of 54.5%). The variance of the techniques is statistically significant at the significance level 0.05 (i.e., $p \leq 0.05$) with the Friedman test. EvoMaster-BB achieves the best average rank, i.e., 1.6, while Schemathesis has the second best average rank, i.e., 1.9. Then, the remaining tools can be divided in 2 groups: bBOXRT (average rank 4.6) and RestTestGen (average rank 3.9) with similar coverage 41.6-45.4%, and then RESTler (average rank 5.1), RESTest (average rank 5.0) and RestCT (average rank 5.8) with similar coverage 33.6-35.4%. These results confirm a previous study [44] showing that RestTestGen gives better results (in terms of black-box criteria) than RESTler and RESTest, as well as RestCT being better than RESTler [69] (although in this case the difference in average coverage is small, only 0.8%). Compared to the analyses in [55], interestingly the ranking of the tools is exactly the same (recall that, out of the combined 29 APIs between our and their study, only 10 APIs used in these empirical studies are the same).

On all APIs but 1, either EvoMaster or Schemathesis gives the best results. The exception is the industrial API, where 5 tools achieve the same coverage of 8.2%, on all their 10 runs. We will discuss this interesting case in more details in Section 4.7. In 12 APIs, either EvoMaster is the best followed by Schemathesis, or the other way round. There is no single API in which EvoMaster and Schemathesis were not at least the third best.

The other 7 APIs (including the industrial one) show some interesting behavior for the other 5 tools. For example, for *js-rest-scs*, RestTestGen gives the second-best results, with an average 86.4%, compared to the 86.1% of Schemathesis. The interesting aspect here is that, out of the 10 runs, RestTestGen has worse minimum coverage (85.4% *vs.* 85.9%) and worse maximum coverage (87.1% *vs.* 87.4%), although the average is higher (86.4% *vs.* 86.1%). This can happen when randomized algorithms are used. In *gestaohospital-rest*, RestTestGen and Schemathesis have similar performance (i.e., 57.2% and 58.7%), whereas EvoMaster is quite behind (i.e., 50.8%). Similarly, the performance of RestTestGen and Schemathesis are very similar on *rest-scs* (65.3% and 64.8%) and *restcountries* (75.4% and 73.9%), where EvoMaster is better only by a small amount (66.9% on *rest-scs* and 76.1% on *restcountries*). In *languagetool*, RESTest is better than Schemathesis, but the difference is minimal (only 0.3%). In *rest-ncs*, there is large gap in performance between EvoMaster (64.5%) and Schemathesis (94.1%), where the second best results are given by RestCT (85.5%). Finally, on *scout-api*, RESTler is better than Schemathesis (26.5% *vs.* 23.0%), although it is way behind EvoMaster (36.7%).

Another interesting observation is that there is quite a bit variability in the results of these fuzzers, as they use randomized algorithms to generate test cases. Let us highlight some of the most extreme cases in Table 2, for EvoMaster and Schemathesis. On *languagetool*, out of the 10 runs, EvoMaster has a gap of 9.1% between the best (35.1%) and worst (26.0%) runs. On *scout-api*, the gap is 8.3%. For Schemathesis, the gap on *features-service* is 10.5%, and 6.4% on *gestaohospital-rest*. This is yet another reminder of the peculiar nature of randomized algorithms, and the importance of how to properly analyze them. For example, doing comparisons based on a single run is unwise.

Table 3. p-values of the Mann-Whitney-Wilcoxon U-Test of EvoMaster's results compared to all the other tools, on each SUT (114 pairwise comparisons in total). Values lower than the $\alpha = 0.05$ threshold are reported in bold. *NaN* values happen when there is no difference in the compared datasets (they are exactly the same). They can be interpreted as p-value equal to 1.

SUT	bBOXRT	RestCT	RESTler	RESTest	RestTestGen	Schemathesis
cyclotron	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
disease-sh-api	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
js-rest-ncs	**0.001**	0.721	**0.001**	**0.001**	**0.003**	**0.001**
js-rest-scs	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
realworld-app	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
spacex-api	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
catwatch	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	0.909
cwa-verification	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	0.434
features-service	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
gestaohospital-rest	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
ind0	NaN	**0.001**	**0.001**	NaN	NaN	NaN
languagetool	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
ocvn-rest	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.033**
proxyprint	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
rest-ncs	**0.001**	**0.001**	**0.001**	**0.001**	0.179	**0.001**
rest-news	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	0.069
rest-scs	**0.001**	**0.001**	**0.001**	**0.001**	**0.039**	**0.007**
restcountries	**0.001**	**0.001**	**0.001**	**0.001**	0.417	**0.001**
scout-api	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**

Statistical tests [32] are needed when claiming with high confidence that one algorithm/tool is better than another one. In this particular case, EvoMaster BB achieved the overall best performance based on the significant p-value (i.e., less than 0.05) with Friedman test and the best average rank over all of the 19 APIs (i.e., 1.6) as shown in Table 2. In addition, we compare EvoMaster's performance with all the other tools, one at a time on each SUT (so $6 \times 19 = 114$ comparisons), and report the p-values of the Mann-Whitney-Wilcoxon U-Test in Table 3. Apart from very few cases, the large majority of comparisons are statistically significant at level $\alpha = 0.05$. Often, 10 repetitions might not be enough to detect statistically significant differences, and higher repetition values like 30 and 100 are recommended in the literature [32]. However, here the performance gaps are large enough that 10 repetitions were more than enough in most cases. Note that, as explained in more details in [32], we have not applied any p-value correction on these multiple comparisons (as they are controversial). We rather report the raw values (rounded up to 0.001 for readability), in case readers still want to apply such corrections when analyzing such data. For completeness, as

Table 4. p-values of the Mann-Whitney-Wilcoxon U-Test of Schemathesis's results compared to all the other tools, on each SUT (114 pairwise comparisons in total). Values lower than the $\alpha = 0.05$ threshold are reported in bold. *NaN* values happen when there is no difference in the compared datasets (they are exactly the same). They can be interpreted as p-value equal to 1.

SUT	bBOXRT	EvoMaster BB	RestCT	RESTler	RESTest	RestTestGen
cyclotron	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
disease-sh-api	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
js-rest-ncs	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
js-rest-scs	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	0.081
realworld-app	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
spacex-api	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
catwatch	**0.001**	0.909	**0.001**	**0.001**	**0.001**	**0.001**
cwa-verification	**0.001**	0.434	**0.001**	**0.001**	**0.001**	**0.001**
features-service	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
gestaohospital-rest	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	0.168
ind0	NaN	NaN	**0.001**	**0.001**	NaN	NaN
languagetool	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
ocvn-rest	**0.001**	**0.033**	**0.001**	**0.001**	**0.001**	**0.001**
proxyprint	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
rest-ncs	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**
rest-news	**0.001**	0.069	**0.001**	**0.001**	**0.001**	**0.001**
rest-scs	**0.001**	**0.007**	**0.001**	**0.001**	**0.001**	0.061
restcountries	**0.001**	**0.001**	**0.001**	**0.001**	**0.014**	**0.019**
scout-api	**0.001**	**0.001**	**0.001**	**0.001**	**0.001**	0.557

Schemathesis achieves the best results on few APIs, Table 4 reports the same kind of analysis, in which Schemathesis is pairwise compared with all the other tools.

When looking at the obtained coverage values, all these tools achieve at least a 30% coverage on average. Only two of them (i.e., EvoMaster and Schemathesis) achieve more than 50%. But no tool goes above 60% coverage. This means that, although these tools might be useful for practitioners, there are still several research challenges that need to be addressed (we will go in more details on this in Section 4.7). However, what level of coverage can be reasonably expected from black-box tools (which have no info on the source code of the SUTs) is a question that is hard to answer.

RQ1: *all compared tools achieve at least 30% line coverage on average, but none goes above 60%. Of the 7 compared tools,* EvoMaster *seems the one giving the best results, closely followed by* Schemathesis.

4.4 RQ2: Time Budget

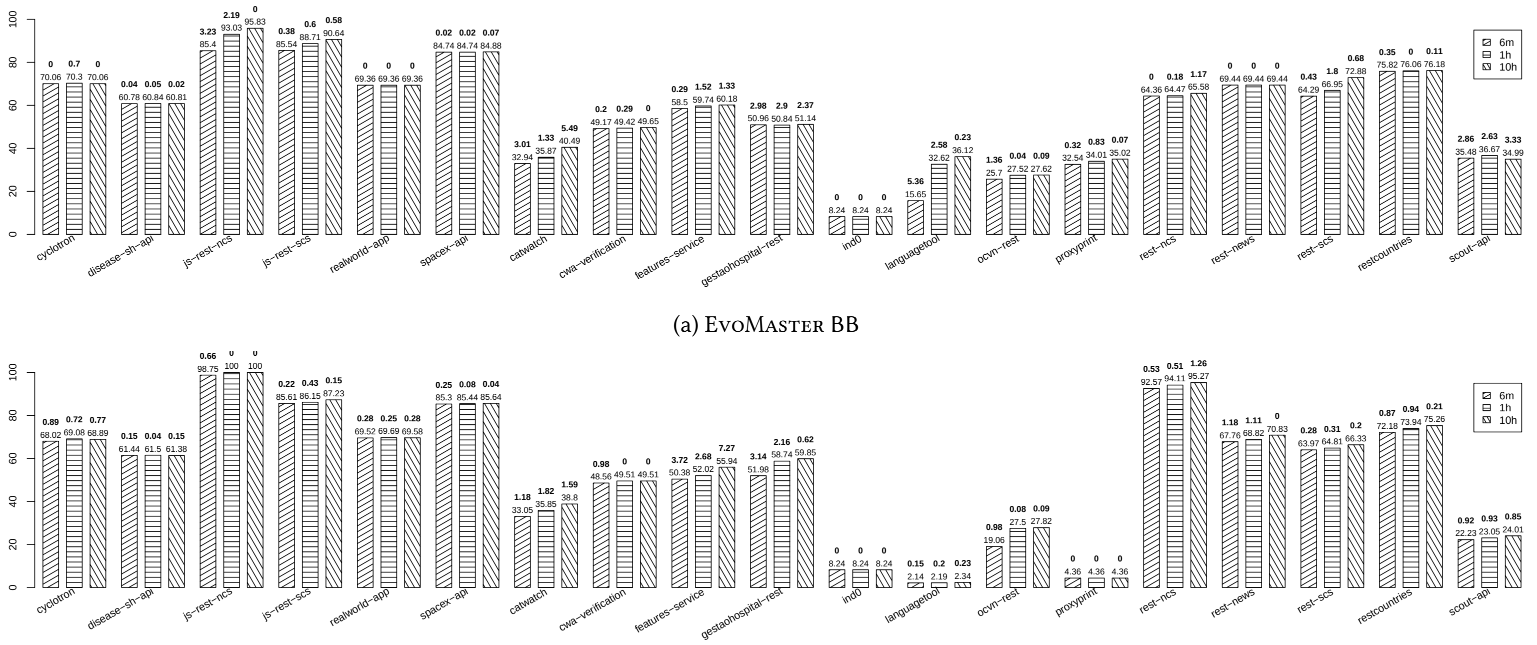
The choice of running experiments for 1 hour is arbitrary, and different results could be obtained when using different time budgets. In the end, what to use as time budget should be based on how practitioners in industry would actually use these fuzzers in their daily jobs. However, as far as we know, there is no scientifically sound data in the literature on how practitioners use these tools in practice. The choice of 1 hour is based on our anecdotal experience of collaborating and discussing with some engineers in industry.

Still, it is interesting to see how results would vary when using different time budgets. These would still represent realistic scenarios, such as engineers that would like to get some results

Table 5. Results of experiment on various time budget settings (i.e., {6m, 1h, 10h}). We report average, minimum and maximum of coverage (i.e., mean [min, max]) achieved by the best two black-box fuzzers (i.e., EvoMaster and Schemathesis) with each time budget setting on each of the 19 REST APIs .

SUT	EvoMaster BB			Schemathesis		
	6m	1h	10h	6m	1h	10h
cyclotron	70.1 [70.1,70.1]	70.3 [70.1,72.3]	70.1 [70.1,70.1]	68.0 [67.0,69.1]	69.1 [67.7,69.7]	68.9 [68.1,69.7]
disease-sh-api	60.8 [60.7,60.9]	60.8 [60.8,60.9]	60.8 [60.8,60.8]	61.4 [61.1,61.7]	61.5 [61.4,61.6]	61.4 [61.3,61.6]
js-rest-ncs	85.4 [82.4,90.4]	93.0 [89.8,95.8]	95.8 [95.8,95.8]	98.8 [98.4,100.0]	100.0 [100.0,100.0]	100.0 [100.0,100.0]
js-rest-scs	85.5 [85.1,86.3]	88.7 [87.5,89.5]	90.6 [90.0,91.2]	85.6 [85.2,86.1]	86.1 [85.9,87.4]	87.2 [87.1,87.4]
realworld-app	69.4 [69.4,69.4]	69.4 [69.4,69.4]	69.4 [69.4,69.4]	69.5 [68.9,69.8]	69.7 [69.1,69.8]	69.6 [69.3,69.8]
spacex-api	84.7 [84.7,84.8]	84.7 [84.7,84.8]	84.9 [84.8,84.9]	85.3 [84.8,85.6]	85.4 [85.3,85.6]	85.6 [85.6,85.7]
catwatch	32.9 [23.6,36.6]	35.9 [34.3,36.9]	40.5 [36.9,46.8]	33.1 [31.1,34.3]	35.8 [33.6,39.1]	38.8 [37.0,40.0]
cwa-verification	49.2 [49.1,49.6]	49.4 [49.1,49.6]	49.6 [49.6,49.6]	48.6 [47.3,49.5]	49.5 [49.5,49.5]	49.5 [49.5,49.5]
features-service	58.5 [58.4,59.5]	59.7 [58.4,62.1]	60.2 [59.3,61.7]	50.4 [46.4,57.5]	52.0 [46.6,57.1]	55.9 [51.4,64.3]
gestaohospital-rest	51.0 [44.8,54.5]	50.8 [44.8,54.3]	51.1 [48.8,53.5]	52.0 [48.5,57.5]	58.7 [55.9,62.3]	59.8 [59.2,60.4]
ind0	8.2 [8.2,8.2]	8.2 [8.2,8.2]	8.2 [8.2,8.2]	8.2 [8.2,8.2]	8.2 [8.2,8.2]	8.2 [8.2,8.2]
languagetool	15.6 [6.8,25.3]	32.6 [26.0,35.1]	36.1 [35.9,36.3]	2.1 [2.1,2.5]	2.2 [2.1,2.5]	2.3 [2.1,2.5]
ocvn-rest	25.7 [22.7,27.1]	27.5 [27.5,27.6]	27.6 [27.6,27.7]	19.1 [16.9,19.9]	27.5 [27.5,27.7]	27.8 [27.8,27.9]
proxyprint	32.5 [32.4,33.6]	34.0 [32.5,35.0]	35.0 [35.0,35.1]	4.4 [4.4,4.4]	4.4 [4.4,4.4]	4.4 [4.4,4.4]
rest-ncs	64.4 [64.4,64.4]	64.5 [64.4,64.7]	65.6 [64.7,66.9]	92.6 [92.4,93.8]	94.1 [93.1,94.5]	95.3 [94.5,96.7]
rest-news	69.4 [69.4,69.4]	69.4 [69.4,69.4]	69.4 [69.4,69.4]	67.8 [65.3,70.1]	68.8 [67.4,70.8]	70.8 [70.8,70.8]
rest-scs	64.3 [63.7,65.4]	66.9 [64.4,70.2]	72.9 [72.2,73.6]	64.0 [63.7,64.4]	64.8 [64.4,65.1]	66.3 [66.1,66.4]
restcountries	75.8 [74.8,76.1]	76.1 [76.1,76.1]	76.2 [76.1,76.2]	72.2 [71.1,73.8]	73.9 [72.4,75.0]	75.3 [75.1,75.5]
scout-api	35.5 [30.9,39.6]	36.7 [32.8,41.1]	35.0 [31.3,37.7]	22.2 [21.4,24.3]	23.0 [22.1,25.1]	24.0 [23.1,24.8]

quickly (e.g., in just a few minutes) or running fuzzing sessions on a Continuous Integration server for several hours during the night. To address this research question, we hence ran a new set of experiments, in which we used 1/10 of the budget (i.e., 6 minutes), and 10 times the budget (i.e., 10 hours). Unfortunately, running each experiment for 10 hours would take a huge amount of time, even when parallelizing the running of the experiments (unless a large number of machines can be used). For this reason, for these experiments we only used the two most performing tools (i.e., EvoMaster and Schemathesis), with the experiments repeated only 3 times. Still, with these reduced settings, these experiments took $2 \times 19 \times ((6 \times 10) + (600 \times 3))m = 49$ days of computational effort.



(a) EvoMaster BB

(b) Schemathesis

Fig. 1. Average (y-axis) and standard deviation (**bold value**) of coverage achieved by EvoMaster BB and Schemathesis with various time budgets (i.e., {6m, 1h, 10h}) on all of the 19 REST APIs

Table 5 shows the results of these experiments. A graphical representation is given in Figure 1. Statistical analyses in which the results of how the different time budgets compare with each other are reported in Table 6.

From this data, a few key observations can be made:

- In most cases, increasing the time budget does increase the achieved code coverage.
- Increases in coverage can be substantial (e.g., +20.5% for EvoMaster on *languagetool*), or very minimal (e.g., only +0.2% for *spacex-api*).
- When there is high variability in the results, experiments with longer time budgets might give worse results by chance. It is not common, but it can happen (e.g., this is the case for EvoMaster on *scout-api*).
- Relatively, already with low time budget (i.e., 6 minutes), reasonable coverage results can be achieved. This is important to take into account, considering that many faults in these REST APIs lay in the first layer of input validation [59].
- There are cases in which there is no difference in results even when using a 100 time larger time budget (i.e., 10 hours *vs.* 6 minutes). This is the case for the industrial API (i.e., *ind0*). Interestingly, white-box testing does achieve much better results than black-box techniques (we will show it in Section 4.6). Although giving more time does improve the performance of black-box fuzzers, there are cases in which it is not enough, and more advanced techniques like white-box testing are required.

RQ2: *Already with a small time budget of 6 minutes good results can be achieved. Longer time budgets can provide better results, but the magnitude of the improvements can vary significantly among the tested APIs.*

4.5 RQ3: Black-box Fault Detection

4.5.1 Issues in Measuring Fault Detection. As previously stated, each tool can self-report how many faults they find, but how such fault numbers are calculated might be very different from tool to tool, making any comparison nearly meaningless. Manually checking (possibly tens of) thousands of test cases is not a viable option either, especially when they are generated in different programming languages (e.g., Java and Python).

Some authors like in [55] use the number of unique exception stack traces in the SUT's logs as a proxy of detected faults. On the one hand, a tool that leads the SUT to throw more exceptions could be considered better. On the other hand, using such metric as proxy for fault detection has many shortcomings, like for example: (1) not all exceptions reported in the logs are related to faults; (2) not all SUTs actual print any logs by default (e.g., this is the case for several SUTs in our study); (3) crashes (which lead to responses with HTTP status code 500) are only one type of faults in RESTful APIs detected by these fuzzers [59], where for example all faults related to response mismatches with the API schema would leave no trace in the logs. For all these reasons, we did not do this kind of analysis on the logs, as we do not think they provide much more info compared to line coverage. Especially considering that the infrastructure to do such analysis would need to be implemented, which might not be a trivial task.

An alternative that we considered was to build a *proxy* HTTP server, to be used between the fuzzer and the SUT. With such a proxy, it would be possible to record all the responses from the SUT, e.g., to log all the endpoints returning a 500 HTTP status code, regardless of the employed fuzzer. We discarded this idea for two main reasons: (1) it could introduce a not trivial delay to all test case evaluations, which could bias the experiments (slower fuzzers would get a relative advantage if each HTTP call takes artificially longer to execute because going through a proxy); (2)

Table 6. Results of comparison on coverage achieved by various time budget settings (i.e., {6m, 1h, 10h}) applied on the best two black-box fuzzers (i.e., EvoMaster and Schemathesis) for all of the 19 REST APIs. To compare the settings of a and b (i.e., a *vs.* b where a is longer than b, e.g., 1h *vs.* 6m), we report p-value using Mann-Whitney-Wilcoxon U-tests and effect sizes ($\hat{A}_{12}$) using Vargha-Delaney statistics. Values in bold indicates that a is significantly better than b in statistics, i.e., p-value < 0.05 and $\hat{A}_{12} < 0.5$, and values in plain indicates that there is no significantly difference between a and b in statistics. Note that *NaN* values of p-value happen when a and b are exactly the same. For each of the fuzzer, among 19 SUTs, #Significance represents a number of the SUTs whose results are significantly improved with a greater time budget settings (i.e., a is significantly better than b). We also report difference and relative difference between average a and average b, i.e., *Diff%=mean(a)−mean(b)* and *Relative*= (*mean(a)−mean(b)*)/*mean(b)*.

						EvoMaster BB						
SUT	1h *vs.* 6m				10h *vs.* 1h				10h *vs.* 6m			
	$\hat{A}_{12}$	p-value	Diff%	Relative	$\hat{A}_{12}$	p-value	Diff%	Relative	$\hat{A}_{12}$	p-value	Diff%	Relative
cyclotron	0.60	0.168	+0.24	+0.35%	0.40	0.502	-0.24	-0.35%	0.50	NaN	0.00	0.00%
disease-sh-api	**0.82**	**0.010**	**+0.06**	**+0.10%**	0.33	0.392	-0.04	-0.06%	0.77	0.176	+0.02	+0.03%
js-rest-ncs	**0.99**	**≤0.001**	**+7.63**	**+8.93%**	**0.90**	**0.045**	**+2.80**	**+3.01%**	**1.00**	**0.013**	**+10.43**	**+12.21%**
js-rest-scs	**1.00**	**≤0.001**	**+3.17**	**+3.71%**	**1.00**	**0.007**	**+1.94**	**+2.18%**	**1.00**	**0.014**	**+5.11**	**+5.97%**
realworld-app	0.50	NaN	0.00	0.00%	0.50	NaN	0.00	0.00%	0.50	NaN	0.00	0.00%
spacex-api	0.50	1.000	0.00	0.00%	**0.98**	**0.004**	**+0.14**	**+0.17%**	**0.98**	**0.004**	**+0.14**	**+0.17%**
catwatch	**0.97**	**≤0.001**	**+2.94**	**+8.92%**	**0.98**	**0.013**	**+4.62**	**+12.87%**	**1.00**	**0.010**	**+7.55**	**+22.93%**
cwa-verification	**0.73**	**0.024**	**+0.26**	**+0.52%**	0.70	0.246	+0.23	+0.46%	**0.93**	**0.005**	**+0.48**	**+0.98%**
features-service	**0.77**	**0.007**	**+1.23**	**+2.11%**	0.65	0.487	+0.44	+0.73%	**0.96**	**0.001**	**+1.67**	**+2.86%**
gestaohospital-rest	0.48	0.939	-0.11	-0.22%	0.50	1.000	+0.29	+0.58%	0.48	1.000	+0.18	+0.35%
ind0	0.50	NaN	0.00	0.00%	0.50	NaN	0.00	0.00%	0.50	NaN	0.00	0.00%
languagetool	**1.00**	**≤0.001**	**+16.97**	**+108.44%**	**1.00**	**0.007**	**+3.50**	**+10.72%**	**1.00**	**0.003**	**+20.47**	**+130.79%**
ocvn-rest	**1.00**	**≤0.001**	**+1.82**	**+7.09%**	**0.87**	**0.048**	**+0.10**	**+0.35%**	**1.00**	**0.010**	**+1.92**	**+7.46%**
proxyprint	**0.95**	**≤0.001**	**+1.47**	**+4.52%**	**0.98**	**0.017**	**+1.01**	**+2.98%**	**1.00**	**0.010**	**+2.49**	**+7.64%**
rest-ncs	**0.65**	**0.036**	**+0.11**	**+0.17%**	**0.95**	**0.015**	**+1.10**	**+1.71%**	**1.00**	**≤0.001**	**+1.21**	**+1.88%**
rest-news	0.50	NaN	0.00	0.00%	0.50	NaN	0.00	0.00%	0.50	NaN	0.00	0.00%
rest-scs	**0.95**	**≤0.001**	**+2.66**	**+4.14%**	**1.00**	**0.014**	**+5.93**	**+8.86%**	**1.00**	**0.008**	**+8.60**	**+13.37%**
restcountries	**0.75**	**0.011**	**+0.24**	**+0.31%**	**0.83**	**0.010**	**+0.12**	**+0.16%**	**0.92**	**0.022**	**+0.36**	**+0.47%**
scout-api	0.60	0.448	+1.19	+3.35%	0.37	0.554	-1.68	-4.58%	0.50	1.000	-0.49	-1.38%
#Significance		12				10				12		
						Schemathesis						
SUT	1h *vs.* 6m				10h *vs.* 1h				10h *vs.* 6m			
	$\hat{A}_{12}$	p-value	Diff%	Relative	$\hat{A}_{12}$	p-value	Diff%	Relative	$\hat{A}_{12}$	p-value	Diff%	Relative
cyclotron	**0.86**	**0.007**	**+1.06**	**+1.56%**	0.47	0.932	-0.19	-0.27%	0.73	0.271	+0.87	+1.28%
disease-sh-api	0.71	0.096	+0.06	+0.10%	0.32	0.380	-0.12	-0.19%	0.37	0.540	-0.06	-0.09%
js-rest-ncs	**0.90**	**≤0.001**	**+1.25**	**+1.27%**	0.50	NaN	0.00	0.00%	**0.90**	**0.021**	**+1.25**	**+1.27%**
js-rest-scs	**0.93**	**≤0.001**	**+0.53**	**+0.62%**	**0.92**	**0.028**	**+1.08**	**+1.26%**	**1.00**	**0.012**	**+1.62**	**+1.89%**
realworld-app	0.70	0.097	+0.17	+0.25%	0.33	0.364	-0.11	-0.16%	0.53	0.931	+0.06	+0.08%
spacex-api	0.61	0.401	+0.13	+0.16%	**0.97**	**0.020**	**+0.21**	**+0.24%**	**0.97**	**0.022**	**+0.34**	**+0.40%**
catwatch	**0.96**	**≤0.001**	**+2.80**	**+8.46%**	**0.93**	**0.028**	**+2.95**	**+8.24%**	**1.00**	**0.010**	**+5.75**	**+17.39%**
cwa-verification	**0.93**	**≤0.001**	**+0.94**	**+1.94%**	0.50	NaN	0.00	0.00%	**0.93**	**0.022**	**+0.94**	**+1.94%**
features-service	0.64	0.238	+1.64	+3.26%	0.56	0.814	+3.93	+7.55%	0.74	0.231	+5.57	+11.06%
gestaohospital-rest	**0.94**	**≤0.001**	**+6.76**	**+13.01%**	0.62	0.611	+1.11	+1.89%	**1.00**	**0.007**	**+7.87**	**+15.14%**
ind0	0.50	NaN	0.00	0.00%	0.50	NaN	0.00	0.00%	0.50	NaN	0.00	0.00%
languagetool	0.41	0.468	+0.05	+2.46%	0.67	0.385	+0.15	+6.77%	0.69	0.323	+0.20	+9.39%
ocvn-rest	**1.00**	**≤0.001**	**+8.44**	**+44.28%**	**1.00**	**0.011**	**+0.32**	**+1.17%**	**1.00**	**0.010**	**+8.76**	**+45.96%**
proxyprint	0.50	NaN	0.00	0.00%	0.50	NaN	0.00	0.00%	0.50	NaN	0.00	0.00%
rest-ncs	**0.96**	**≤0.001**	**+1.54**	**+1.66%**	0.83	0.076	+1.16	+1.24%	**1.00**	**0.001**	**+2.70**	**+2.92%**
rest-news	**0.76**	**0.032**	**+1.06**	**+1.57%**	**0.95**	**0.023**	**+2.01**	**+2.93%**	**1.00**	**0.007**	**+3.08**	**+4.54%**
rest-scs	**0.97**	**≤0.001**	**+0.84**	**+1.32%**	**1.00**	**0.011**	**+1.51**	**+2.34%**	**1.00**	**0.007**	**+2.36**	**+3.68%**
restcountries	**0.94**	**≤0.001**	**+1.76**	**+2.44%**	**1.00**	**0.013**	**+1.32**	**+1.78%**	**1.00**	**0.009**	**+3.08**	**+4.27%**
scout-api	**0.82**	**0.010**	**+0.82**	**+3.70%**	0.83	0.112	+0.96	+4.15%	**0.90**	**0.037**	**+1.78**	**+8.00%**
#Significance		12				7				12		
#TotalSignificance		24				17				24		

such an approach would not be able to distinguish between different faults triggered in the same endpoint.

Another alternative was to use some Mutation Testing [53] tool. But we are aware of no such tool aimed at system testing (i.e., they all seem tailored for unit testing). Furthermore, there could be major technical challenges in using a Mutation Testing tool when the generated tests and the SUT are written in different programming languages (e.g., Python and Java). Designing a novel Mutation Testing tool for RESTful APIs would be useful, but out of the scope of this paper.

Existing work on comparing fuzzers in different domains, like for example for data parsers (e.g., [42]), use SUT *crashes* as oracle to detect faults. Those are easy to identify (the SUT is no longer running), although smart techniques need to be used to distinguish possible different faults in the same SUT when it crashes. This does not happen in RESTful APIs, as crashes in the business logic do not crash the HTTP servers they run on. The HTTP servers catch these exceptions, and return a HTTP response with status 500. Also, as already stated, this is only one type of faults that can be automatically detected in RESTful APIs [59]. Different kinds of automated oracles have been defined in the literature [50]. Only focusing on that type of fault could be too restrictive.

4.5.2 Experiment Setup. In the end, for a more fair comparison on fault detection among the different fuzzers for RESTful APIs, we created an artificial API, with 10 distinctive types of artificial faults in it (as mentioned in Section 4.1). We then ran each black-box fuzzer on it, and *manually* evaluated their generated test cases to check if the faults were identified.

The API is designed in a way that triggering faults in it should be *trivial*. In other words, just calling an endpoint would trigger a fault, regardless of any input. The goal here is to check if the fuzzers are able to *identify* if the obtained responses are indeed faulty. In other words, we are empirically checking which range of automated oracles each fuzzer does employ.

Regardless of the achieved code coverage, a fuzzer could be extended with more automated oracles from the literature. But how to integrate them is not necessarily trivial. For example, consider *robustness* faults, in which wrong data (according to the schema) is sent, and a fuzzer then verifies if the HTTP response has status code in the 4xx family (i.e., representing a user error). Sending invalid data might negatively impact the efforts on maximizing code coverage in the SUT, as input validation is done in the first layer of the API. How often and when to send invalid inputs to check this type of automated oracle is hence a non-trivial design decision in a fuzzer.

In our artificial API, we have injected the following 10 different types of faults for which an automated oracle can be defined. Note, this is not meant to be an exhaustive list of all possible faults that can be automatically detected with an oracle. For example, we have not considered security related faults (e.g., based on access policies). This selection is only meant as a starting point, to shed more light on this important issue.

The 10 fault types are:

(1) Thrown exception in the business logic of the API, which results in a returned 500 HTTP status code.
(2) An endpoint returns a payload, where a field has wrong type, based on the schema.
(3) An endpoint returns a success status code (i.e., in the 2xx family) that is not declared in the schema.
(4) A payload requires a numeric field to be constrained within a certain maximum value, but this constraint is ignored in the API.
(5) A payload requires a string field to be constrained within a certain enumeration of values, but this constraint is ignored in the API.
(6) A payload requires a boolean field to be present, but this constraint is ignored in the API.

(7) A POST endpoint returns a *location* header (pointing to the newly created resource) which is invalid, resulting in a 404 status code if it is used in a GET call.
(8) A DELETE endpoint returns status code 200, but it does not delete anything (i.e., a GET on the same URL would still return the non-deleted resource).
(9) A PUT endpoint wrongly applies a partial update (with similar semantics to RFC 7396 *JSON Merge Patch*), instead of doing a full resource replacement (which can be detected by fetching the modified resource with a GET), as per HTTP semantics.
(10) A PUT endpoint wrongly applies a non-idempotent modification. Therefore, calling the same endpoint more than once leads to different results, which can be checked by fetching the modified resource with a GET. Recall that all HTTP verbs are idempotent, but POST and PATCH.

We ran each of the seven black-box fuzzers on the artificial API. As the endpoints of such API are trivial to cover, we ran each tool only once, for just 6 minutes. Then, we manually checked all the generated test cases, to see which of the 10 faults were correctly identified.

4.5.3 Experiment Results. Table 7 shows the results of these experiments. On this API, RestCT did not manage to generate any test case, as it seems like it has issues when dealing with endpoints not having query or path parameters. Although the API is trivial, RESTest and RestTestGen did not manage to achieve full coverage, in contrast to the other four fuzzers.

Of the 10 faults, only #1 is reliable found by most tools. Fault #2 was found only by bBOXRT. For fault #3, it was found only by EvoMaster and Schemathesis. None of the other 7 faults was properly identified by any of the fuzzers. However, there are some clarifications to make:

bBOXRT: according to [56], it should be able to find robustness testing faults, i.e., #4, #5 and #6, but it did not. It seems that it tried to send different kinds of faulty inputs, but with wrong payload types, where all these HTTP requests ended up with a 415 status code (i.e., *Unsupported Media Type*). However, it was the only fuzzer correctly identifying fault #2.

RESTler: according to [39], it should be able to find #8, but it did not.

RESTest: we were not able to determine if it finds any fault. It generated 121 JUnit files, containing 968 test cases. But none of the test cases shows what faults it detects in its source code. They need to be run, where faults are checked via custom log statements in the used HTTP library (i.e., RestAssured). Running such tests requires a library from RESTest, which is released on Maven Central. But such library, at the time of this writing, seems missing some transitive dependencies, and so it was not possible to run any of the generated tests.

RestTestGen: as bBOXRT, according to [45] it should be able to detect faults #4, #5 and #6. It might had, but it is unclear which faults it finds. For example, it can generate test sequences with 20 HTTP calls, with a final comment stating *"The erroneous test sequence was accepted as valid by the server"*. But there is no info on what makes the test sequence "erroneous".

The results of this experiment clearly show that current fuzzers only detect a small subset of possible fault types in RESTful APIs. Most fuzzers have been reported in the literature [49] finding hundreds/thousands of faults in existing APIs. This is typically done by detecting 500 HTTP status codes (i.e., like in fault #1). However, not all faults have the same severity. For example, many cases of 500 status code is simply wrong handling of invalid inputs [59]. Returning a 500 instead of 400 on wrong inputs is not as serious as a fault like #10. In this case, this latter type of fault can have catastrophic consequences, as it not only can corrupt the data of the API, but also it would be hard to reproduce and debug due to the non-determinism and low frequency of re-executing idempotent methods on the HTTP stack.

RQ3: *Existing fuzzers detect only a small number of types of faults in RESTful APIs.*

Table 7. Results of the experiment on fault detection using all selected black-box fuzzers on *rest-faults*. For each fuzzer, we report the achieved line coverage and the type of detected faults.

Fuzzer	Line Coverage %	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
bBOXRT	100.0	✓	✓								
EvoMaster BB	100.0	✓		✓							
RestCT	26.9										
RESTler	100.0	✓									
RESTest	61.5										
RestTestGen	92.3	✓			?	?	?				
Schemathesis	100.0	✓		✓							

4.6 RQ4: White-Box Testing Experiments

Table 8. Experiment results for white-box evolutionary search (WB) and and grey-box random search (RS), using EvoMaster on the 19 RESTful APIs. We report the average line coverage measured with EvoMaster itself (not JaCoCo, nor c8), as well as the number of faults detected in these APIs. When the differences are statically significant at $\alpha = 0.05$ level, the effect-sizes $\hat{A}_{12}$ are reported in bold.

SUT	Line Coverage %				# Detected Faults			
	RS	WB	$\hat{A}_{12}$	p-value	RS	WB	$\hat{A}_{12}$	p-value
cyclotron	31.2	31.3	0.69	0.158	32.2	30.9	**0.13**	0.004
disease-sh-api	18.1	19.5	**1.00**	0.001	30.0	36.2	**1.00**	0.001
js-rest-ncs	60.7	83.2	**1.00**	0.001	6.0	6.0	0.50	1.000
js-rest-scs	60.9	76.2	**1.00**	0.001	1.0	1.0	0.50	1.000
realworld-app	24.6	24.4	**0.03**	0.001	25.0	30.5	**1.00**	0.001
spacex-api	41.5	41.5	0.58	0.540	52.2	50.9	0.32	0.170
catwatch	41.1	50.0	**1.00**	0.001	18.0	23.7	**1.00**	0.001
cwa-verification	40.4	46.9	**1.00**	0.001	4.0	4.0	0.50	1.000
features-service	68.8	81.8	**1.00**	0.001	22.2	31.4	**1.00**	0.001
gestaohospital-rest	39.4	39.5	**0.72**	0.024	15.0	22.0	**1.00**	0.001
ind0	8.4	18.8	**1.00**	0.001	1.0	48.1	**1.00**	0.001
languagetool	28.9	39.5	**0.93**	0.001	16.6	12.8	0.55	0.731
ocvn-rest	35.3	24.8	**0.00**	0.001	279.9	257.9	**0.00**	0.001
proxyprint	53.3	51.6	0.28	0.120	84.1	86.3	0.59	0.531
rest-ncs	61.9	93.0	**1.00**	0.001	5.0	6.0	**1.00**	0.001
rest-news	55.4	66.5	**1.00**	0.001	5.0	7.8	**1.00**	0.001
rest-scs	63.3	86.2	**1.00**	0.001	1.0	12.0	**1.00**	0.001
restcountries	74.9	77.1	**1.00**	0.001	2.0	2.0	0.50	1.000
scout-api	54.7	53.4	0.37	0.346	94.0	88.6	**0.14**	0.008
Average	45.4	52.9	0.77		36.5	39.9	0.67	

Out of the seven compared tools, only EvoMaster supports white-box testing. EvoMaster uses evolutionary computation techniques, where the bytecode of the SUTs is instrumented to compute different kinds of heuristics. Due to possible conflicts with JaCoCo and c8 instrumentation, and due to the fact that EvoMaster uses its own driver classes (which need to be written manually) to start and stop the instrumented SUTs, this set of experiments was run differently compared to the black-box ones.

We ran EvoMaster on each SUT for 10 repetitions (so, 190 runs), each one for 1 hour (like for the black-box experiments). However, JaCoCo and c8 are not activated. After each run, EvoMaster generates statistic files, including information on code coverage and fault detection. However, as there can be differences on how line coverage is computed between EvoMaster and JaCoCo/c8, it would be difficult to reliably compare with the results in Table 2. Therefore, for the comparisons, we ran EvoMaster as well in grey-box mode (for another 190 runs). This mode generates test cases in exactly the same way as black-box mode, with difference that the SUT is instrumented, and coverage metrics are computed at each test execution. A further benefit is that, besides code coverage, we can reliably compare fault detection as well, as this metric is computed in exactly the same way (as it is the same tool). As EvoMaster is the black-box fuzzer that gives the highest code coverage (Section 4.3), it is not a major validity threat to compare white-box results only with EvoMaster. These 380 runs added a further 15.8 days of computational effort.

Table 8 shows these results. Few things appear quite clearly. First, on average, line coverage goes up by 7.5% (from 45.4% to 52.9%). Even with just 10 runs per API, results are statistically significant in most cases.

For APIs like *rest-ncs*, average coverage can go even higher than 90%. For other APIs like *features-service* improvements are more than 13% (e.g., from 68.8% to max 81.1%). In the case of *ind0*, although achieved coverage is relative low (i.e., less than 20%), it still more than double (i.e., from 8.4% to 18.8%).

Although results are significantly improved compared to black-box testing, for nearly half of these SUTs it was still not possible to achieve more than 50% line coverage. Also, there are two interesting cases in which results with white-box testing are actually significantly *worse*, i.e., for *realworld-app* and *ocvn-rest*. In the former case, the difference is minimal, just 0.2%. In the latter case, though, the difference is substantial, as it is 10.5% (from 35.3% down to 24.8%). This is not an unusual behavior for search algorithms, as it all depends on the quality of the *fitness function* and the properties of the *search landscape* [43]. If a fitness function gives no gradient to the search algorithm, it can easily get stuck in local optima. In such cases, a random search can give better results. EvoMaster uses several heuristics in its fitness function to try to maximize code coverage, but those do not work for *ocvn-rest*, as we will discuss in more details in Section 4.7. However, improving the fitness function in this case can be done (which we will address in future versions of EvoMaster).

Regarding fault detection, white-box testing leads to detect more faults. In the context of REST API testing, a 500 status code is regarded as potential fault occurred in the SUT, which can be identified by both black-box and white-box approaches, by extracting info from the HTTP responses. It is not unexpected that a white-box approach achieves better performance, as usually there is a strong correlation between code coverage and fault detection [42]. For example, you cannot detect a fault if the code it hides in is never executed. However, it also relies on the employed fitness functions and heuristics, e.g., whether they focuses more on fault detection than on other test criteria (such as code coverage and schema related coverage [49]) . The interesting aspect here is that the improvement is not much, just 3.4 more faults on average. In these experiments, random testing can find 36.5 faults on average, and so the relative improvement is less than 10%. For the problem domain of fuzzing RESTful APIs (and likely Web APIs in general), this is not surprising [59]. Many of these APIs do crash as soon as an invalid input is provided, instead of returning a proper user error response (e.g., with HTTP status code in the 4xx family). And random search is quite good at generating invalid input values. So, many faults can be easily found this way with a fuzzer at the very first layer of input validation in the SUT's code, even when the achieved code coverage is relatively low.

> **RQ4:** *White-box fuzzing leads to significantly higher results compared to black-box fuzzing, up to 7.5% more code coverage and 3.4 more detected faults on average. But, still, for many APIs coverage results were less than 50%.*

4.7 RQ5: Open Problems

To identify open problems, we performed detailed analysis based on results of existing fuzzers on real APIs (18 open source and one industrial). We ran seven tools/configurations on 19 RESTful APIs, with a budget of 1 hour per experiment. However, only in one single case it was possible to achieve 100% line coverage (i.e., Schemathesis on *js-rest-ncs*, recall Table 2). In general, it might not be possible to achieve 100% line coverage, as some code can be unreachable. This is for example a common case for constructors in static-method-only classes, and catch blocks for exceptions that cannot be triggered with a test. However, for several of these SUTs it was not even possible to reach 50% line coverage.

It is not in the scope of this paper to define what would be a good coverage target to aim for (80%? 90%?). However, clearly the higher the code coverage the better it would be for practitioners using these fuzzers. So, to improve those fuzzers, it is of paramount importance to understand what are their current limitations. To answer this question, we studied in detail the logs of the tools in Section 4.7.1, and large parts of the source code of the SUTs in Section 4.7.2 (recall that those are more than 280 thousand LOCs, see Table 1). An in-depth analysis of the current open-problems in fuzzing RESTful APIs is an essential scientific step to get more insight into this problem. This is needed to be able to design novel, more effective techniques.

Here, we do two different types of analyses. First, we look at the logs generated by the tools (Section 4.7.1), which helps in pointing out possible major issues in these fuzzers (e.g., when they crash they might generate log messages with stack traces). Second, we run the generated tests and manually check the code they execute (Section 4.7.2), to analyze what was not covered. This helps to provide hypotheses to explain why that was the case, which is needed to be able to design new techniques to achieve higher coverage. Finally, we summarize all these findings (Section 4.7.3).

It is important to stress out that the goal of these analyses is to identify general problems, or instances of problems that are likely going to be present in other SUTs as well. Designing novel techniques that just *overfit* for a specific benchmark is of little to no use. Ultimately, what is important will be the results that the practitioners can obtain when using these fuzzers on their APIs.

4.7.1 Analysis Of The Logs. From the tool logs, at least four common issues are worth to discuss. First, OpenAPI/Swagger schemas might have some *errors* (e.g., this is the case for *cyclotron*, *disease-sh-api*, *cwa-verification*, *features-service proxyprint*, and *ocvn-rest*). This might happen when schemas are manually written, as well as when they are automatically derived from the code with some tools/libraries (as those tools might have faults). In these cases, most fuzzers just crash, without making any HTTP call or generating any test case. It is important to warn the users of these issues with their schema, but likely fuzzers should be more *robust* and not crash, e.g., endpoints with schema issues could be simply skipped.

The second issue can be seen in *languagetool*. Most fuzzers for RESTful APIs support HTTP body payloads only in JSON format. But, in HTTP, any kind of payload type can be sent. JSON is the most common format for RESTful APIs [65], but there are others as well like XML and the `application/x-www-form-urlencoded` used in *languagetool*. From the results in Table 2, it looks

```
2022-06-12 12:30:28.112: Sending: 'GET //api/fisher/1/1/1.23 HTTP/1.1\r\nAccept: application/json\r\nHost: localhost:25900\r\nContent-Length: 0\r\nUser-Agent: restler/8.5.0\r\n\r\n'

2022-06-12 12:30:28.126: Received: 'HTTP/1.1 404 Not Found\r\nX-Powered-By: Express\r\nContent-Security-Policy: default-src \'none\'\r\nX-Content-Type-Options: nosniff\r\nContent-Type: text/html; charset=utf-8\r\nContent-Length: 159\r\nDate: Sun, 12 Jun 2022 10:30:28 GMT\r\nConnection: keep-alive\r\nKeep-Alive: timeout=5\r\n\r\n<!DOCTYPE html>\n<html lang="en">\n<head>\n<meta charset="utf-8">\n<title>Error</title>\n</head>\n<body>\n<pre>Cannot GET //api/fisher/1/1/1.23</pre>\n</body>\n</html>\n'
```

(a) RESTLER on *js-rest-ncs* built with NodeJS and Express. By default, a path with extra slash will not match the correct endpoint.

```
2022-06-11 22:55:56.651: Sending: 'GET //api/fisher/1/1/1.23 HTTP/1.1\r\nAccept: application/json\r\nHost: localhost:24850\r\nContent-Length: 0\r\nUser-Agent: restler/8.5.0\r\n\r\n'

2022-06-11 22:55:56.697: Received: 'HTTP/1.1 200 \r\nContent-Disposition: inline;filename=f.txt\r\nContent-Type: application/json;charset=UTF-8\r\nTransfer-Encoding: chunked\r\nDate: Sat, 11 Jun 2022 20:55:55 GMT\r\n\r\n38\r\n{"resultAsInt":null,"resultAsDouble":0.5328886540720141}\r\n0\r\n\r\n'
```

(b) RESTLER on *rest-ncs* built with Spring Boot 2.0.3 whose default path matching strategy allows the double slash.

Fig. 2. Different handling of requests with different techniques

like only EVOMASTER supports this format. On this API, EVOMASTER achieves between 26% and 35.1% code coverage, whereas no other fuzzer achieves more than 2.5%.

The third issue is specific to *scout-api*, which displays a special case of the JSON payloads. Most tools assume a JSON payload to be a *tree*: e.g., a root object `A` that can have fields that are object themselves (e.g., `A.B`), and so on recursively (e.g., `A.B.C.D` and `A.F.C`), in a tree-like structure. However, an OpenAPI/Swagger schema can define objects that are *graphs*, as it is the case for *scout-api*. For example, an object `A` can have a field of type `B`, but `B` itself can have a field of type `A`. This recursive relation creates a graph, that needs to be handled carefully when instantiating `A` (e.g., optional field entries can be skipped to avoid an infinite recursion, which otherwise would lead the fuzzers to crash).

The fourth issue is related to the execution of HTTP requests towards the SUTs. To test a REST API, the fuzzers build HTTP requests based on the schema, and then use different HTTP libraries to send the requests toward the SUT, e.g., `JerseyClient` is used in EVOMASTER. By analyzing the logs, we found that for several case studies, some fuzzers seem to not have problems in parsing schemas, but they failed to execute the HTTP requests. For instance, `javax.net.ssl.SSLException: Unsupported or unrecognized SSL message` was thrown when RESTTESTGEN processed *realworld-app* with `OkHttpClient`. For *js-rest-ncs*, *js-rest-scs* and *ind0*, we found that `404 Not Found` responses were always returned when fuzzing them with RESTLER v8.5.0. By checking the logs obtained by RESTLER, we found that it might be due to a problem in generating the right URLs for making the HTTP requests. For the SUT whose `basePath` is `/`, RESTLER seems to generate double slash (i.e., `//`) in the URL of the requests. However, whether to accept the double slash to match a real path depends on the SUTs, i.e., *js-rest-ncs*, *js-rest-scs* and *ind0* do not allow it. In Figure 2, we provide the logs obtained by RESTLER, representing the processed requests which contain the double slash and responses returned by *js-rest-ncs* (Figure 2a) and *rest-ncs* (Figure 2b).

Table 9. Main issues preventing higher coverage, reported for each SUT.

SUT	Authentication	Databases	External Services	Schema Inconsistencies	String Constraints	Unreachable Code
catwatch			✓			
cwa-verification				✓	✓	
cyclotron		✓		✓		✓
disease-sh-api						✓
features-service		✓				
gestaohospital-rest		✓				
ind0					✓	
js-rest-ncs						
js-rest-scs					✓	
languagetool				✓		
ocvn-rest				✓	✓	✓
proxyprint	✓			✓	✓	✓
realworld-app		✓				
rest-ncs						
rest-news		✓				✓
rest-scs					✓	
restcountries						✓
scout-api	✓				✓	✓
spacex-api	✓	✓				
Total	3	6	1	5	7	7

4.7.2 Analysis Of the Source Code. For each SUT, we manually ran the best (i.e., highest code coverage) test suite (which are the ones generated by EvoMaster) with code coverage activated. Then, in an IDE, we manually looked at which branches (e.g., `if` statements) were reached by the test case execution, but not covered. As well as looking at the cases in which the test execution is halted in the middle of a code block due to thrown exceptions. This is done to try to understand what are the current issues and challenges that these fuzzers need to overcome to get better results.

To be useful for researchers, these analyses need to be ‘‘low-level’’, with concrete discussions about the source code of these APIs. No general theories can be derived without first looking at and analyzing the single instances of a scientific/engineering phenomenon. As such software engineering discussions might depend on different groups of readers who might target different levels of detail in the software, or considering readers that might not be currently active in the development of a fuzzer, here we provide only a summary. Full analyses for each SUT are provided in Appendix A.

Table 9 summarizes all the identified *main issues*, for each SUT. There can be many reasons for why higher coverage was not achieved. Here, based on our analyses, we have categorized 6 main issues, namely:

Authentication. In some cases, there is the need to provide authentication info with specific roles, using specific types of authentication (e.g., an API can provide different ways to authenticate).

Databases. The execution might depend on the data returned by database queries, but those return nothing, as constraints are not satisfied (e.g., the content of `WHERE` clauses in SQL `SELECT` queries). It might be hard to generate the right data to satisfy those constraints.

External Services. The API depends on specific interactions with external services (e.g., other APIs on the internet). But, from the point of view of the fuzzers, there is no way to control what those external services return.

Schema Inconsistencies. The schema of the API might be underspecified (e.g., some data constraints are missing, and some query parameters used in the API might not be described), or also providing wrong info (e.g., wrong types for input parameters).

String Constraints. String input data might need to match specific formats (e.g., based on regular expressions), which is not simple to infer.

Unreachable Code. Some code can be simply unreachable via the entry-points of the API. It might be dead-code, or code executed by other scripts or functionality (e.g., a GUI and background threads).

Note that these are only the main issues we *currently* face. Solving them does not imply maximizing code coverage. As more code is executed, more issues could be identified. A concrete example is *cwa-verification*. Such API does connect to an external service, but the code doing this is currently not executed by any of the fuzzers. Dealing with external services would hence currently give no improvement on *cwa-verification*. However, when the other issues in *cwa-verification* are solved (recall Table 9), then dealing with external services will be needed to improve coverage results further.

4.7.3 Discussion. Based on the analyses of the logs and tests generated for the 19 APIs, some general observations can be made:

- Many research prototypes are not particularly robust, and can crash when applied on new SUTs. For example, we have faced this issue with EvoMaster many times, like when adding a new API to EMB [7]. Although we add new APIs to EMB each year, EMB has been available as open-source since 2017, and anyone can use it for their empirical studies and make sure their tools do not crash on it. However, it is important to stress out that, in this paper, we have compared *tool implementations* rather than *techniques*. For example, a tool with low performance (e.g., due to crashes) could still feature novel techniques that could be very useful, e.g., if integrated or re-implemented in a more mature tool.
- Like software, also schemas can have faults, and/or omissions (e.g., constraints on some inputs might be missing). And this issue seems quite common, especially when schemas are written manually. Although this problem could be addressed by white-box testing (currently supported only by EvoMaster) by analyzing the source code of the SUTs, it looks like a major issue for black-box testing, which might not have a viable solution.
- Interactions with databases are common in RESTful APIs. To execute the code to fetch some data, such data should be first present in the database. The data could be created with endpoints of the API itself (e.g., `POST` requests), as well as inserted directly into the database with SQL commands. Both approaches have challenges: e.g., how to properly link different endpoints that work on the same resources, and how to deal with data constraints that are specified in the code of the API and not in the SQL schema of the database. The compared fuzzers provide different solutions to address this problem, but it is clear that more still need to be done.

- Currently, no fuzzer deals with the mocking of external web services (e.g., using libraries such as WireMock). Testing with external live services has many shortcomings (e.g., the generated tests can become flaky), but it might be the only option for black-box strategies. For white-box strategies, mocking of web services will be essential when testing industrial enterprise systems developed with a microservice architecture (as in this class of software interactions between web services are very common).
- Authentication information needs to be provided when fuzzing APIs that require different users to login. Even with a white-box fuzzer, it might not be feasible to automatically create different users with different roles. For example, typically passwords in databases are hashed, and reverse engineering how to create valid hashed passwords (e.g., when creating data into the database as part of the tests [33]) is out of reach from current fuzzers. To reach specific parts of the code, there might be the need of specific users with specific roles. If those are not provided/setup before the fuzzing is done, then such code cannot be executed. A tester might provide different user profiles for fuzzing, but some important roles/setups might be missing from these manual configurations.
- Constraints on strings input seem quite common in RESTful APIs. Those constraints might be missing in the schema (e.g., a specific string input needs to satisfy a given regular expression). White-box fuzzers can analyze the code to see how such strings are manipulated, but it is not trivial. Black-box fuzzers could try to infer these constraints with natural language processing (e.g., based on the names of these inputs, and possibly their description, if any is provided).

Many of the challenges we have identified (e.g., dealing with databases and underspecified schemas) are specific for RESTful APIs, although they would likely apply to the fuzzing of other types of web services as well (e.g., GraphQL [40, 41] and RPC [74]). These challenges would not be meaningful in different testing contexts, e.g., *unit test* generation or fuzzing of parser libraries.

Issues with providing manual auth configurations would likely apply to any kind of software that needs authentication information. Dealing with constraints on string data is likely the most general challenge, as it applies as well for example to unit testing.

This means that solving these challenges would not only help the testing of RESTful APIs, but likely it can have positive effects on other testing domains as well.

> **RQ5:** *There are several open challenges in fuzzing RESTful APIs, including for example how to deal with underspecified schemas, and how to deal with interactions with their environment (e.g., databases and other external APIs).*

5 THREATS TO VALIDITY

Internal Validity. Besides writing the scaffolding to run and analyze the experiments, before running the experiments we did not make code modifications in any existing tool for this study. Although we are the authors of EvoMaster, to try to be fair, we used the latest release before running these experiments, without re-running them after fixing any new issues. However, as we have used all these APIs in our previous studies, EvoMaster did not crash on any of them.

The use of EMB [7] as case study might bias the results towards EvoMaster. To avoid EvoMaster overfit on it, EMB has been extended each year with new APIs, of different size and complexity. Furthermore, as EMB has been open-source for a few years, anyone can use it as case study to evaluate the effectiveness of their fuzzers.

We used seven existing tools, with their latest release versions when possible. However, there is a possibility that we might have misconfigured them, especially considering that some of those tools have minimal documentation. To avoid such possibility, we carefully look at their execution logs, to see if there was any clear case of misconfiguration. Furthermore, we release all our scripts as open-source in the repository of EvoMaster, so anyone can review them, and replicate the study if needed.

Any comparison of tools made by the authors of one of these tools is bound to be potentially biased, especially if such tool turns out to give the best results (as in our case with EvoMaster). However, the relative performance of the other six tools among them would be not affected by this issue. Likewise, the in-depth analysis of the SUTs is not affected by this issue either.

All the compared fuzzers use randomized algorithms. To take this into account, each experiment was repeated either 3 times (10 hour budget) or 10 times (6 minutes and 1 hour budgets), and we analyzed them with the appropriate statistical tests.

External Validity. The chosen seven fuzzers are arguably representing the state-of-the-art in testing RESTful APIs. However, results on 19 APIs might not generalize to other APIs as well, although we selected different programming runtimes (e.g., JVM and NodeJS) as well including one industrial API. This kind of system testing experiments are expensive (more than 120 days of computational effort in our case), which makes using more APIs challenging. Furthermore, finding RESTful APIs in open-source repositories is not so simple, and considerable effort might then be needed to configure them (e.g., setup external dependencies like databases and find out how/if they use any form of authentication).

6 CONCLUSIONS

RESTful APIs are widely used in industry, and so several techniques have been developed in the research community to automatically test them [49]. Several reports in the literature show the usefulness in practice of these techniques, by reporting actual faults automatically found with these fuzzers. However, not much has been reported on how the different techniques compare, nor on how effective they are at covering different parts of the code of these APIs. This latter is usually the case due to the testing of remote APIs, for which researchers do not have access to their source code.

To address these issues, in this paper we have compared the state-of-the-art in fuzzing RESTful APIs, using seven fuzzers to test 20 APIs (18 open-source, 1 industrial, and 1 artificial used only for fault analysis). This totaled more than 280 thousands of lines of code (for their business logic, but not including the millions of lines of code of all the third-party libraries they use). Each fuzzer was run for 1 hour, and each experiment was repeated 10 times, to take into account the randomness of these tools. To get a better understanding of the results, the best fuzzers were also run for 6 minutes and 10 hours (but this latter repeated only 3 times).

The results show different degrees of coverage for the different black-box testing tools, where EvoMaster seems the tool giving the best results (i.e., highest code coverage on average) on this case study, closely followed by Schemathesis. However, no black-box fuzzer was able to achieve more than 60% line coverage on average. Furthermore, the experiments show that white-box testing gives better results than black-box testing. Still, large parts of these APIs are left uncovered, as the fuzzers do not manage to generate the right data to maximize code coverage. Also, only a small number of types of faults were detected by these fuzzers. Although these fuzzers are already useful for practitioners in industry, more needs to be done to achieve better results.

To address this issue, this large empirical analysis has then be followed by an in-depth analysis of the source code of each of these 19 APIs, to understand what are the main issues that prevent the fuzzers from achieving better results. Several issues were identified, including for example how to deal with underspecified schemas, and how to deal with interactions with external services (e.g., other APIs and databases). This provides a useful list of common problems that researchers can use to drive new research effort in improving performance on this problem domain.

For few of these issues, we discussed possible solutions, but those will need to be empirically validated. Future work will aim at using the new knowledge and insight provided in this paper to design new variants of these tools, to be able to achieve better results (in terms of code coverage and fault detection). In particular, the insight provided in this study is shaping the current research efforts in EvoMaster, pointing to clear research issues that need to be prioritized.

Comparing existing tools provides insight on the current limitations of fuzzing RESTful APIs. Although comparing results on code coverage is not particularly complex, there are major issues when trying to do fair comparisons based on fault detection, especially in an automated way. Future work will be needed to design automated code infrastructure and scaffolding to run experiments based on fault detection.

All the tools and APIs used in this study are available online. To enable the replicability of this study, all our scripts used in our experiments are published as open-source, available online in the repository of EvoMaster, at www.evomaster.org. Documentation is provided to specify how to use them (e.g., "Replicating studies" section). These scripts get automatically stored on Zenodo at each new release (e.g., version 1.5.0 [36]).

ACKNOWLEDGMENTS

This work is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EAST project, grant agreement No. 864972).

APPENDIX

A SOURCE-CODE ANALYSIS

In this Appendix, we will discuss the execution of test cases on each SUT, in alphabetic order, one at a time. Hypotheses are then provided to explain why higher coverage was not achieved.

Note that, when we refer to the code coverage achieved by white-box EvoMaster, we refer to the results in Table 8. These coverage values are computed with the instrumentation of EvoMaster itself, and they are not exactly the same as what other coverage tools like JaCoCo and c8 would report (as there can be differences on how coverage is computed). For example, coverage results reported by c8 might be significantly higher, as those are computed only on the script files that are loaded (based on all the experiments), whereas for EvoMaster all source files were considered. In other words, if on each run i a tool cover X_i lines out of N_i total reported, then the coverage percentage c_i is computed as $c_i = X_i/N$, where $N = max(N_i)$. Furthermore, a current limitation of EvoMaster for JavaScript is that the recorded coverage does not include the coverage achieved at boot-time, but only from when the search starts. This was an issue that has been fixed for the JVM, but not for JavaScript yet. Therefore, the numbers in Table 8 are not directly comparable with the numbers in Table 2.

A.1 catwatch

With an average line coverage of 50%, this can be considered a non-trivial API. The main issue is that this SUT makes a call to an external service (more specifically, to the GitHub APIs to fetch project info). But such call seems to get stuck for a long time (and possibly timeout), which leads to none of the code parsing the responses (and do different kind of analyses) being executed. Calling external services is a problem for fuzzers. External services can go down or change at any time. They can return different data at each call, making assertions in the generated tests become flaky. Although testing with the actual external services is useful and should be done, the generated tests would likely not be suitable for regression testing. This is a known problem in industry, and there are different solutions to address it, like *mocking* the external services (e.g., in the JVM ecosystem,

```
@PostMapping(value = TAN_ROUTE,
    consumes = MediaType.APPLICATION_JSON_VALUE,
    produces = MediaType.APPLICATION_JSON_VALUE
)
public DeferredResult<ResponseEntity<Tan>> generateTan(
          @Valid @RequestBody RegistrationToken registrationToken,
          @RequestHeader(value = "cwa-fake", required = false) String fake) {
    if ((fake != null) && (fake.equals("1"))) {
      return fakeRequestService.generateTan(registrationToken);
    }
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    Optional<VerificationAppSession> actual
      = appSessionService.getAppSessionByToken(registrationToken.
    getRegistrationToken());
    if (actual.isPresent()) {
```

Fig. 3. A function snippet from the `ExternalTanController` class in the API *cwa-verification*.

WireMock [24] is a popular library to do that). Enhancing fuzzers to deal with mocked services (e.g., to setup the mock data they should return) is going to be an important venue of future research.

A.2 cwa-verification

On this API, achieved coverage is less than 50% (i.e., 46.9%). There are two interesting cases to discuss here, which have major impact on the achieved coverage.

First, Figure 3 shows a snippet of code for one of the endpoint handlers (i.e., `ExternalTanController`, but the same issue happens as well in `ExternalTestStateController` and `InternalTanController`). Here, the condition `actual.isPresent()` is never satisfied, which leads to miss large part of the code. A token is given as input as part of the body payload, and then a record matching such token in the database is searched for. However, none is found, and EvoMaster is unable to create it directly. In theory, such case should be trivial to handle with SQL support [33], but it was not the case. The reason is the peculiar properties of such token. Such token has the given constraint in the OpenAPI schema:

```
"registrationToken": {
 "pattern": "^[a-f0-9]{8}-[a-f0-9]{4}-4[a-f0-9]{3}-[89aAbB][a-f0-9]{3}-[a-f0
    -9]{12}$",
 "type": "string"
}
```

This results in the token having a length of 36 characters. EvoMaster has no problem in sampling strings that match such a regular expression, like for example: `bfe8b80b-dca1-458d-B312-96ecae446be0`. However, such constraint is not present in the SQL database, where the column for these tokens is simply declared with:

```
- column:
    name: registration_token_hash
    type: varchar(64)
```

When EvoMaster generates data directly into the database, they would be random strings that do not satisfy such regular expression. Again, this should not a problem thanks to taint analysis [34]. The reason it is not working is due to the length of the string, which is 36 characters. By default,

```
@PostMapping(value = TELE_TAN_ROUTE,
    produces = MediaType.APPLICATION_JSON_VALUE
)
public ResponseEntity<TeleTan> createTeleTan(
    @RequestHeader(JwtService.HEADER_NAME_AUTHORIZATION) @Valid AuthorizationToken
     authorization,
    @RequestHeader(value = TELE_TAN_TYPE_HEADER, required = false) @Valid
    TeleTanType teleTanType) {

    List<AuthorizationRole> requiredRoles = new ArrayList<>();

    if (teleTanType == null) {
      teleTanType = TeleTanType.TEST;
      requiredRoles.add(AuthorizationRole.AUTH_C19_HOTLINE);
    } else if (teleTanType == TeleTanType.EVENT) {
      requiredRoles.add(AuthorizationRole.AUTH_C19_HOTLINE_EVENT);
    }
```

Fig. 4. A function snippet from the `InternalTanController` class in the API *cwa-verification*.

EvoMaster does not generate random strings with more than 16 characters. Even if with taint analysis we could inject the right string into the database, currently this does not happen due to 36 being greater than 16.

Note that having a constraint on the length of strings is essential. Sampling unbound random strings with billions of characters would have too many negative side effects. The choice of the value 16 is arbitrary: could had been higher, or lower. Still, whatever limit is chosen, an SUT could need strings longer than such limit, as it happens in this case for *cwa-verification*.

A solution to address this issue is to distinguish between 2 different max-length limits: an *hard* one that should never be violated (e.g., a constraint in the OpenAPI or SQL schemas), and a *soft* one (e.g., 16). The soft limit would be used when sampling random strings, but could be violated in some specific circumstances (like for example in taint analysis).

The second interesting case to discuss is present in the endpoint handler `InternalTanController`, shown in Figure 4. Here, an HTTP header with value `TELE_TAN_TYPE_HEADER="X-CWA-TELETAN-TYPE"` can be provided as input. However, such info is missing from the OpenAPI schema. Therefore, the object `teleTanType` is always null. This is an example of underspecified schema.

A.3 cyclotron

On this API, there is not much difference between white-box and grey-box testing. There are at least 3 major issues that are worth to discuss.

First, a non-negligible amount of code is executed only if some configuration settings are *on*. But those are *off* by default (like for example `config.analytics.enable` and `config.enableAuth` in `config.js`). All the code related to these functionality cannot be tested via the RESTful endpoints.

Second, like for several other APIs in this study, the schema here is not fully correct/complete. For example, Figure 5 shows a snippet for the handler of the endpoint `/data/{key}/upsert`, where the two fields `data` and `keys` are read from the input object in the HTTP body payload of the request. However, the schema has no formal definition about those fields, as they are just mentioned as a comment (see Line 17 in Figure 6).

```
exports.upsertData = function (req, res) {
    if (req.body == null) {
        return res.status(400).send('Missing data.');
    }

    var upsertData = req.body.data;
    var keys = req.body.keys;

    if (upsertData == null) {
        return res.status(400).send('Missing data.');
    }
     if (keys == null) {
        return res.status(400).send('Missing keys.');
    }
```

Fig. 5. A function snippet from the api.data.js file in the API *cyclotron*, for the endpoint /data/{key}/upsert.

```
"/data/{key}/upsert": {
  "post": {
    "summary": "Upserts an object in the Data Bucket Data",
    "description": "Upserts (inserts or updates) an object in the data for a given
     Data Bucket. The rev property will be incremented.",
    "tags": [
       "Data"
     ],
     "parameters": [{
          "name": "key",
          "in": "path",
          "description": "The Data Bucket key.",
          "required": true,
          "type": "string"
        }, {
          "name": "data",
          "in": "body",
          "description": "An object containing 'keys' and 'data'.",
          "required": true
        }],
```

Fig. 6. Snippet of the OpenAPI schema for the API *cyclotron*.

Third, the API does several accesses to a MongoDB database. Several endpoints start by retrieving data from the database, like for example with commands like `Dashboards.findOne({ name: dashboardName })`. If the data is missing, no following code is then executed. But it is hard to get the right id (e.g., `dashboardName` in this example) by chance. Although EvoMaster has support for SQL databases (i.e., to analyze all executed queries at runtime), it does not for MongoDB, and not for NodeJS (i.e., SQL support is currently only implemented for the JVM).

```
private EvaluationResult addEvaluationsToResult(
        EvaluationResult result,
        ProductConfiguration configuration,
        Set<FeatureConstraint> featureConstraints){
 for(FeatureConstraint featureConstraint
                              : featureConstraints) {
   result = featureConstraint.evaluateConfiguration(
                                result,configuration);
 }

 if(result.isValid && !result.derivedFeatures.isEmpty()){
   for(String derivedFeature : result.derivedFeatures) {
       configuration.active(derivedFeature);
       result.derivedFeatures.remove(derivedFeature);
       result = this.addEvaluationsToResult(
```

Fig. 7. A function snippet from the `ConfigurationEvaluator` class in the API *features-service*.

A.4 disease-sh-api

On this API, there is not much difference between EvoMaster (both black-box and white-box) and Schemathesis. After an analysis of the generated tests, practically all achievable coverage is obtained, as the remaining code is not reachable through the REST endpoints in the API.

First, the schema refers only to version `v3` of the API, but in the source code there is still all the implementation of the endpoints for version `v2`. Those endpoints are not executable based on the info in the schema. Second, large part of the code deals with the fetching of disease records from online databases, and store them in a local Redis instance. But such code is executed only via scripts, and not from the RESTful endpoints. Note that, when the experiments are run, the database was populated with a selection of valid data, as there is no REST endpoint to add it.

A.5 features-service

With an average line coverage of 81.8%, *features-service* can be considered one of the easier APIs. Quite a bit of code cannot be reached with any generated system test, like for example getters/setters that are never called, or some functions that are only used by the manually written unit tests. There are still some branches that are not covered, related to properties of data in the database. In other words, when a `GET` is fetching some data, some properties are checked one at a time, but, to be able to pass all these checks, a previous `POST` should had created such data. Figure 7 shows one such case, where the code inside `if` statement does not get executed. Tracing how database data is impacting the control flow execution, and which operations should be first called to create such data, is a challenge that needs to be addressed.

A.6 gestaohospital-rest

On this API, there is no much difference between white-box and grey-box EvoMaster (i.e., average coverage 39.5% *vs.* 39.4%, Table 8). Furthermore, RestTestGen and Schemathesis give better results than EvoMaster (Table 2).

One specific property of this API compared to most of the other SUTs in our empirical study is that such API uses a MongoDB database. In contrast to SQL databases, white-box EvoMaster has no support for NoSQL databases (such as MongoDB). As this API does many interactions with the database (e.g., there is a lot of calls like `service.findByHospitalId(hospital_id)`), not much coverage is achieved if such accessed data is not present in the database, and the evolutionary search has no gradient to create it. In this regard, it seems that RestTestGen and Schemathesis do a better job at creating resources with POST commands and use such created data for their following GET requests. Still, although quite good, the coverage is at most 62.3% (measured with JaCoCo for Schemathesis), which means there still some challenges to overcome.

We could not analyze in detail the output of these tools in an IDE for the Java program *gestaohospital-rest* (e.g., using a debugger), as RestTestGen generated only JSON files with the descriptions of the tests (and no JUnit file), and the outputs of Schemathesis are in YAML for the VCR-Cassette format (which is mainly for Ruby and Python, where the port for Java seems has not been under maintenance for several years).

A.7 ind0

This is a closed-source API provided by one of our industrial partner. Therefore, we cannot provide any code example for it, but still we can discuss it at a high level. This API is a service in a microservice architecture. Although in terms of size it is not particularly big (recall Table 1), it is the most challenging API in our case study. No black-box fuzzer achieves more than 8.2% line coverage, and white-box testing goes only up to 18.8% (on average).

Out of 20 endpoints, all but one endpoint require string inputs satisfying a complex regular expression, but such info is not present in the OpenAPI schema. So, a random string is extremely unlikely to satisfy such constraint. Furthermore, the constraint is expressed in a @ annotation, which is handled by the Spring framework before any of the HTTP handlers are executed.

Thanks to taint-analysis, EvoMaster can handle these cases [34], even when constraints are evaluated in third-party libraries (and not just in the business logic of the SUT). Still, there is quite a bit of variability in the results, i.e., given the average coverage 18.8%, the standard variation is 7.0, with coverage going from a minimum of 11.9% to a maximum of 32.6% (out of the 10 repeated experiment runs). EvoMaster manages to solve these constraints, but not in all runs, as such strings not only get matched with a regular expression, but then are farther checked with other constraints. However, as the regular expression is exactly the same for all these 19 endpoints, applied on a URL path variable with the same name, it should be possible to design strategies to re-use such information to speed up the search, instead of resolving the same constraints each time for each endpoint. Even with the run with highest coverage 32.6%, many endpoints are not covered due to this issue.

A.8 js-rest-ncs

This is an artificial API, using numerical-computation functions (e.g., Triangle Classification [26]) behind REST endpoints. As this is a re-implementation in JavaScript of *rest-ncs*, we will defer to Section A.14 for the analyses.

One thing to notice though is that the coverage values are higher for a very simple reason: the fetching of the OpenAPI schema is part of business logic of the API, and code coverage is computed for it (as the schema is inside a JavaScript file as a string inside an HTTP endpoint declaration). On the other hand, for Java, the schema is handled as a JSON resource.

A.9 js-rest-scs

Similarly to the case of *js-rest-ncs*, *js-rest-scs* is a JavaScript re-implementation of *rest-scs*, which involved string-based computation functions. We therefore will defer to Section A.16 for the analyses.

As for *js-rest-ncs*, the fetching of the schema impacts the computed code coverage. However, for white-box EvoMaster, coverage is worse for *js-rest-scs* than for *rest-scs*. The reason is rather

```java
private void handleCheckRequest(
        HttpExchange httpExchange,
        Map<String, String> parameters,
        ErrorRequestLimiter errorRequestLimiter,
        String remoteAddress) throws Exception {
 AnnotatedText aText;
 if(parameters.containsKey("text") &&
            parameters.containsKey("data")) {
    throw new IllegalArgumentException(
      "Set only 'text' or 'data' parameter, not both");
 } else if (parameters.containsKey("text")) {
    aText = new AnnotatedTextBuilder()
        .addText(parameters.get("text")).build();
 } else if (parameters.containsKey("data")) {
    ObjectMapper mapper = new ObjectMapper();
    JsonNode data = mapper.readTree(
                                  parameters.get("data"));
```

Fig. 8. A function snippet from the ApiV2 class in the API *languagetool*.

simple: the support for white-box testing of JavaScript code in EvoMaster is a much more recent addition [75], and not all features that are implemented for the JVM (e.g., different forms of taint analysis [34]) are currently supported.

A.10 languagetool

On this API, coverage is up to 39.5%. This is the largest and most complex API in our study (recall Table 1). This is a CPU bound (e.g., no database) application, doing complex text analyses. This API has only two endpoints: /v2/languages which is a simple GET with no parameters (and so it is trivially covered by just calling it once), and /v2/check which is a POST with 11 input parameters. However, looking at the source code of ApiV2.handleRequest it seems there are some more endpoints, although those are not specified in the OpenAPI/Swagger schema. Without an extensive analysis of this SUT, it is unclear how much of its code would not be coverable due to this issue.

Following the execution from the entry point /v2/check, there are few missed branches that are worthy to discuss. In Figure 8, the last statement crashes due to mapper.readTree throwing an exception (and so all statements after it cannot be executed). Here, when a form in a application/x-www-form-urlencoded payload is received, one out of the 11 parameters is treated as JSON data (i.e., the input called data). Such info is written in the schema as a comment, although the type is registered as a simple string. It is unlikely that a random string would represent a valid JSON object.

Another issue is when entering in the class TextChecker. Several if statements refer to input parameters that are undefined in the schema, such as textSessionId, noopLanguages, preferredLanguages, enableTempOffRules, allowIncompleteResults, enableHiddenRules, ruleValues, sourceText, sourceLanguage and multilingual. Dealing with underdefined schemas is a major issue with fuzzers, especially black-box ones. Technically, these can be considered as *"faults"* in the schemas, which should be fixed by the users. However, schemas can be considered as *documentation*, and issues in the documentations might receive less priority compared to faults in the API implementation. Still, as fuzzers heavily rely on such schemas, a more widespread use of fuzzers might lead in changes in industrial practice by giving compelling reasons to timely update and fix those schemas.

As there are tens of thousands of line that were not covered in this SUT, there are many other missed branches that would be interesting to discuss in detail. But it is unclear how related or independent they are from the aforementioned issues. Once the issue of dealing of underdefined schemas is solved, it will be important to re-run these experiments to identify which major issues are still left.

A.11 ocvn-rest

On this API, with black-box testing line coverage was up to 35.3%, which is significantly higher than the 24.8% achieved by white-box testing. It is the second largest API in our study, following *languagetool*, based on number of lines of code. But, in terms of endpoints, it is the largest with 192 of them. It would not be possible, given the space, to discuss each single of these 192 endpoints. To analyze the achieved coverage, we hence used the test suites generated with both white-box and black-box testing. These have hundreds of test cases. Even calling each endpoint just once would result in at least 192 HTTP calls in the generated tests. Considering the complexity of this SUT, maybe using a larger search budget over 1 hour could be recommended.

More than 35% of the codebase (packages `org.devgateway.ocvn` and `org.devgateway.ocds.persistence`) deal with connections to databases such as SQL and MongoDB, but such code does not seem to be executed. This might happen if the endpoint executions fail before writing/reading from the databases (e.g., due to input validation). The definitions of HTTP handlers in the package `org.devgateway.ocds.web.rest.controller` takes more than 30% of the codebase. But only half of it (i.e., around 15%) gets covered by the tests. Few endpoints (e.g., in the class `UserDashboardRestController`) were not covered because they require admin authentication, which was not setup for these experiments (recall the discussion about authentication in Section 4.3). The class `CorruptionRiskDashboardIndicatorsStatsController` defines several HTTP endpoints, but none of them appears in the OpenAPI/Swagger schema. So, they cannot be called by the fuzzers.

Around 5% of the codebase seems to deal with the generation of Excel charts, but that code is not covered. Out of the 14 HTTP endpoints dealing with this functionality, they all fail on the very first line (and so a substantial part of the codebase is not executed, possibly much higher than 5%), which is a variation of the following statement with different string inputs as second parameter:

```
final String chartTitle = translationService.getValue(
                filter.getLanguage(),
                "charts:cancelledFunding:title");
```

Here, the language is given as input in the HTTP requests, which then gets matched with a regular expression. This makes the `translationService.getValue` call crash when the regular expression is not satisfied. White-box testing can analyze these cases, but it can be hard for black-box techniques if the info on such regular expressions is not available in the OpenAPI/Swagger schemas.

Another large part of the code that is not covered (around another 5%) is in the package `org.devgateway.ocds.web.spring`. However, most of this code seems only be called by the frontend of the OCVN application (and the jar file for *ocvn-rest* does not include such module). From the point of view of the HTTP endpoints, this can be considered dead-code, because it cannot be reached from those endpoints.

It might be surprising that white-box testing gives significantly worse results than grey-box testing, by a large margin (i.e., 10.5%). In a combinatorial optimization problem this can happen

```java
@ApiOperation(value = "Count the tenders and group the results by year. The year is calculated from "
    + "tender.tenderPeriod.startDate.")
@RequestMapping(value = "/api/countTendersByYear", method = { RequestMethod.POST, RequestMethod.GET },
    produces = "application/json")
public List<DBObject> countTendersByYear(@ModelAttribute @Valid final YearFilterPagingRequest filter) {
```

Fig. 9. A function snippet from the `CountPlansTendersAwardsController` class in the API *ocvn-rest*.

```java
@EachRange(min = MIN_REQ_YEAR, max = MAX_REQ_YEAR)
protected TreeSet<Integer> year;

@EachRange(min = MIN_MONTH, max = MAX_MONTH)
protected TreeSet<Integer> month;

@EachPattern(regexp = "^[a-zA-Z0-9]*$")
private TreeSet<String> bidTypeId;

@EachPattern(regexp = "^[a-zA-Z0-9]*$")
private TreeSet<String> notBidTypeId;

@EachPattern(regexp = "^[a-zA-Z0-9]*$")
private TreeSet<String> procuringEntityId;

@EachPattern(regexp = "^[a-zA-Z0-9]*$")
private TreeSet<String> notProcuringEntityId;

@EachPattern(regexp = "^[a-zA-Z0-9]*$")
private TreeSet<String> contrMethod;

@Min(0)
protected Integer pageNumber;

@Range(min = 1, max = MAX_PAGE_SIZE)
protected Integer pageSize;
```

Fig. 10. All fields with declared constraints (9 in total) in the class `YearFilterPagingRequest` in the API *ocvn-rest*.

when the fitness function provides little to no gradient to the search, generating the so called *fitness plateaus*. Mutation operators that only do small changes to the chromosome of an individual (i.e., an evolved test case in this context) would have lower chances to escape from such local optima. EvoMaster has some advance mechanism to address these cases, like for example *adaptive hypermutation* [72]. However, it was not enough to handle *ocvn-rest*. The problem here is that most endpoints take a JSON object as input, with several constraints that must be satisfied. But those constraints are not evaluated in the business logic of the SUT, but rather in a third-party library. These constraints are not specified in the OpenAPI schema. Figure 9 shows an example, in which the JSON input is unmarshalled into an `YearFilterPagingRequest` instance. Because this input is marked with the annotation `@Valid`, its validity is evaluated before the method `countTendersByYear()` is called. If it is not valid, then the Spring framework returns an HTTP response with status 400 (i.e., user error), without calling the method `countTendersByYear()`.

As shown in Figure 10, the class `YearFilterPagingRequest` has 9 fields on which `javax.validation` constraints are declared. Only need 1 (out of 9) violated constraint to invalidate the whole object. Because the constraints are evaluated in a third-party library, the fitness function has no gradient to guide the search to solve all of them. Looking at the results, it seems like that generating valid instances at random is feasible, albeit with low probability. On the other hand, small mutations on an existing invalid instance have little to no chance to make the object valid.

To address these issues, there could be at least three strategies:

- (White-Box) Compute the fitness function to optimize the coverage (using different search heuristics) on all the code, including third-party libraries, and not just the business logic of the SUT. In this way, there would be gradient for generating tests in which the code that evaluates `@Valid` constraints returns true. Although in theory possible, this does not sound like a very promising approach (unless only the needed code in the third-party libraries is instrumented). In most cases, the code of the business logic of the SUT is just tiny compared to the source code of all used third-party libraries, which includes application frameworks such as Spring, HTTP servers such as Tomcat, and ORM libraries such as Hibernate. Instrumenting everything to compute advance heuristics on all third-party libraries would likely have huge performance overhead. This is particularly the case for all code that is executed before the business logic, e.g., the unmarshalling of incoming HTTP requests.
- (White-Box) Natively support `@Valid` in the fitness function, by creating a branch distance for each field, with the objective of minimizing the distance for all fields. Different branch distances can be computed for the different constraints which are part of the standard `javax.validation` (e.g., `javax.validation.constraints.Min`). A new testing target (to optimize for) could be created for each `@Valid` annotation in the business logic of the SUT. However, one major challenge here is that `javax.validation` allows to create custom constraint annotations. This is for example the case for `@EachPattern` and `@EachRange` used in Figure 10.
- (Black-Box) Even if the info on the field constraints are missing in the OpenAPI schema, it can be detected in the schema that the same object type is used as input in more than one endpoint. If that is the case, and if for an endpoint it is difficult to create a valid instance (e.g., all evaluated HTTP calls so far return a response with HTTP status 400), then a possible strategy could be to re-use as input an instance created for another endpoint for which a 2xx status code was returned (if any has been created so far during the search).

A.12 realworld-app

On this API, there is a small difference between EvoMaster black-box and Schemathesis (c8 instrumentation: 69.4% *vs.* 69.7%), and between grey-box and white-box EvoMaster (EvoMaster instrumentation without bootstrap coverage: 24.6% *vs.* 24.4%).

Most of the code is covered. What is left are either unfeasible branches, or branches related to fetching data from the database (MySQL in this case) with some given properties. Figure 11 shows one such example, where 3 queries into the database are executed, based on the 2 inputs `id` and `slug`. The condition of `if` statement at the end of that code snippet is never satisfied, as it depends

```
unFavorite(id, slug) {
  return __awaiter(this, void 0, void 0, function* () {
    let article = yield this.articleRepository.findOne({ slug });
    const user = yield this.userRepository.findOne(id);
    const deleteIndex = user.favorites.findIndex(_article => _article.id ===
  article.id);
    if (deleteIndex >= 0) {
```

Fig. 11. A function snippet from the `article.service.js` file in the API *realworld-app*.

```
@Secured({"ROLE_USER"})
@RequestMapping(value = "/consumer/requests",
                method = RequestMethod.GET)
public String getRequests(Principal principal) {
  JsonObject response = new JsonObject();
  Consumer consumer = consumers.findByUsername(
                          principal.getName());
  if(consumer == null) {
      response.addProperty("success", false);
      return GSON.toJson(response);
  }
```

Fig. 12. Snippet of function handler for the endpoint `/consumer/request` in *proxyprint*.

on these 3 SQL queries. Recall that, although EvoMaster has support for SQL query analyses (which could help in this case), it is currently only for the JVM, and not NodeJS.

A.13 proxyprint

With a line coverage of up to 53.3%, this is an API in which reasonable results are achieved, but more could be done. Several functions in this API are never called, and so they are impossible to cover, like for example the methods `sendEmailFinishedPrintRequest` and `sendEmailCancelledPrintRequest` in the class `MailBox`, and the private methods `singleFileHandle` and `calcBudgetsForPrintShops` in the class `ConsumerController`. So, 100% coverage is not possible on this API.

This API provides a selection of various branches that are not covered. However, it is not straightforward to find out the interesting challenges here. One problem is that the number of HTTP calls on this API is low, and so not much search is actually carried out compared to the other SUTs when using a 1 hour budget. For example, the generated statistic files of EvoMaster report an average of 37 393 HTTP calls per experiment on *proxyprint*, whereas for example for *rest-news* it is 372 733, i.e., nearly 10 times as more. So, many of these missed branches could be covered if running EvoMaster for longer. Still, some of these branches seem quite unlikely to be covered even with larger search budgets. Let us discuss a few of them. Some are related to authentication. For example, in Figure 12, the code inside the `if` statement is never executed. To reach that statement, an HTTP call with valid authentication information needs to be provided. Such authentication info is validated with the database when the `Principal` object is instantiated by the framework (Spring Security in this case). Authentication information needs to be available when the SUT starts (unless the API has some way to register new users directly on-the-fly from the API itself). Therefore, to deal with security (especially when involving hashed passwords),

```
@RequestMapping(value="paypal/ipn/consumer/{consumerID}",
                method=RequestMethod.POST)
protected void consumerLoadUpConfirmation(
      @PathVariable(value = "consumerID") long cid,
      HttpServletRequest request,
      HttpServletResponse response
      ) throws ServletException, IOException {
  Map<String,String> configurationMap =
                           Configuration.getConfig();
  IPNMessage ipnlistener = new IPNMessage(
                                     request,
                                     configurationMap);
  boolean isIpnVerified = ipnlistener.validate();
  String transactionType = ipnlistener
                                 .getTransactionType();
  Map<String,String> map = ipnlistener.getIpnMap();

  String payerEmail = map.get("payer_email");
  Double quantity = Double.valueOf(map.get("mc_gross"));
```

Fig. 13. Snippet of function handler for the endpoint /paypal/ipn/consumer/{consumerID} in *proxyprint*.

some initialization script is required, e.g., to register a set of users with valid username/password information. In the case of *proxyprint*, also some other data in other tables is created as well, e.g., in the Consumer table. To be able to cover such branch, either a fuzzer should find a way to create new valid users, or modify the existing data in the database (e.g., EvoMaster can add new data, but not modify the existing one [33]).

Figure 13 shows an example in which the line defining the variable quantity throws an exception, due to Double.valueOf being called on a null input. Here, an HTTP object request is passed as input to the constructor of IPNMessage, which is part of PayPal SDK library. Inside such library, request.getParameterMap() is called to extract all the parameters of the HTTP request, which are used to populate the map object returned by ipnlistener.getIpnMap(). However, as such parameters are read dynamically at runtime, the OpenAPI/Swagger schema has no knowledge of them (as for this SUT the schema is created automatically with a library when the API starts). Therefore, there is no info to use an HTTP parameter called mc_gross of type double. As such parameter is used directly without being transformed, testability transformations with taint analysis might be able to address this problem [34], but such techniques would need to be extended to support getParameterMap() and Map.get.

Figure 14 shows another interesting example, where the value of a string field in a JSON payload is used directly to instantiate an enum value, i.e., the statement Item.RingType.valueOf(rti-.getRingType()). This fails, as a random string is extremely unlikely to represent a valid value from a restricted set. This might be handled by providing such info in the OpenAPI/Swagger schema (which supports defining enumerations on string fields), or also by handling valueOf() in enumeration by extending the techniques in [34] to support it.

The handler calcBudgetForPrintRequest for the endpoint /consumer/budget is never called (and so all the business logic related to this endpoint remains uncovered by the tests). Such endpoint requires as input a payload with type multipart/form-data, but the schema wrongly specifies the application/json type. A fuzzer might be able to handle this case automatically, but it could

```
@Secured("ROLE_MANAGER")
@RequestMapping(value="/printshops/{id}/pricetable/rings", method=RequestMethod.PUT)
public String editRingsItem(
        @PathVariable(value = "id") long id,
        @RequestBody RingTableItem rti) {
    PrintShop pshop = printshops.findOne(id);
    JsonObject response = new JsonObject();

    if(pshop!=null) {
      BindingItem newBi = new BindingItem(
                    Item.RingType.valueOf(
                                 rti.getRingType()),
                                 rti.getInfLim(),
                                 rti.getSupLim());
```

Fig. 14. Snippet of function handler for the endpoint `/printshops/id/pricetable/rings` in *proxyprint*.

be considered as a major issue in the schema, that should be fixed by the users like a fault in the SUT. In other words, there is a big difference between not providing all information (e.g., missing enum value constraints like in the case of `/printshops/{id}/pricetable/rings`) and providing *wrong* information (i.e., wrong payload type as in the case of `/consumer/budget`).

A.14 rest-ncs

On this SUT, white-box testing with EvoMaster achieves an average of 93%. An analysis of the non-covered lines shows that those are not possible to reach, i.e., dead-code. An example is checking twice if a variable is lower than 0, and return an error if so. In such case, it is impossible to make the second check true. Therefore, as the maximum achievable coverage is obtained in each single run, this can be considered as a *solved* problem.

Regarding black-box testing, all but 2 fuzzers achieve more than 50% line coverage, with Schemathesis achieving 94.1% coverage (computed with JaCoCo), followed by RestCT achieving 85.5% average coverage. On this API, black-box EvoMaster gives worse results, i.e., 64.4%.

This is a rather interesting phenomenon, which is strongly dependent on some design choices these fuzzers make. For example, some branches in this API depend on whether 2 or more integer inputs are equal (e.g., in the case of Triangle Classification). Given two 32-bit integers X and Y, there is only a $1/2^{32}$ chance that $X = Y$. On average, it will need to sample more than 2 billion values before obtaining $X = Y$. With such constraints, it will be very difficult for a black-box fuzzer to cover this kind of branches if integer inputs are sampled at random. However, a fuzzer does not need to sample from the whole spectrum of all possible integers. For example, it could sample from a restricted range, e.g., $[-100, 100]$. On the one hand, the shorter the range, the highest chances to sample $X = Y$. On the other hand, a short range could make impossible to cover branches that require values outside such range (e.g., 12345). This is a tradeoff that needs to be made.

A.15 rest-news

On this API, it was possible to achieve up to 66.5% line coverage. As for the other SUTs, most of the missing coverage is due to dead-code, which is impossible to execute. However, there are some branches that should be possible to cover, but they were not. These seem all related to the same issue, which is the update of existing data in the database. Databases allow to define constraints

```
public static String subject(String directory, String file){
    int result = 0;
    String[] fileparts = null;
    int lastpart = 0;
    String suffix = null;
    fileparts = file.split(".");
    lastpart = fileparts.length - 1;
    if (lastpart > 0) {
```

Fig. 15. Snippet of `subject` function from the `FileSuffix` class in the API *rest-scs*.

```
List<Country> countries = CountryService.getInstance().getByCodeList(codes);
if (!countries.isEmpty()) {
    return parsedCountries(countries, fields);
}
return getResponse(Response.Status.NOT_FOUND);
```

Fig. 16. Code snippet from the `CountryRestV2` class in the API *restcountries*.

on their data (e.g., a number should be within a certain range), using SQL commands such as `CHECK`. White-box EvoMaster can generated test data directly into SQL databases [33], taking into account all these constraints (as all this info is available in the schema of the database). If any constraint is not satisfied, then the `INSERT` SQL commands will fail. The problem here is that the SUT might define further constraints on such data. In JVM projects, this is commonly done with `javax.validation` annotations when using ORM libraries such as Hibernate [9]. But EvoMaster does not seem to handle this, and generates data that is valid for the database (as all the `CHECK` operations pass and the `INSERT`s do not fail), but not for these further constraints. So, in an update endpoint, invalid (from the point of view of `javax.validation` constraints) data can be read from the database, which then the SUT fails to write it back (as these `javax.validation` constraints are check on each write operation) when the update endpoint has done its computations.

A.16 rest-scs

With a line coverage of up to 86.2%, this is an API in which white-box testing is highly effective. There were two types of branches that were not covered: the ones involving regex checks with `Pattern.matches`, and the other involving the dot character ".". In this latter case, it is due to EvoMaster not using such character when sampling random strings, which make the `if` statement in Figure 15 impossible to cover. This might sound like something rather simple to fix. However, as soon as having special characters used in random strings, extra care needs to be taken into consideration when outputting test cases (e.g., the character '$' has special meaning in Kotlin, and would need to be escaped when used in strings, which is not the case for Java). Character escaping rules can be very different based on the context in which they are used in a test case (e.g., inside a URL, inside a JSON object passed as HTTP body payload or data injected into a SQL database). All these cases have to be handled, otherwise the tools would end up generating test cases that do not compile.

A.17 restcountries

```
@POST @Timed @UnitOfWork
@Consumes(MediaType.APPLICATION_JSON)
public MediaFile create(
        @Auth @ApiParam(hidden = true) AuthResult authResult,
        @Context HttpServletResponse response,
        MediaFile mediaFile) {
  doAuth(authResult, response, Permission.mediaitem_create);
  try {
  URI uri = new URI(mediaFile.getUri());
  if ("data".equals(uri.getScheme())) {
```

Fig. 17. Snippet of `create` function from the `MediaFileResource` class in the API *scout-api*.

On this API, high coverage is achieved (i.e., 77.1%). Like the other SUTs, there is quite a bit of dead-code due to getters/setters that are never called and catch blocks for exceptions that might not be possible to throw via test cases. The class `StripeRest` presents an interesting case, as it defines the endpoint `/contribute`, but its info is not in the schema (and so such endpoint is never called).

Figure 16 shows a code snippet which is repeated several times with small variations in few endpoints. Here, this API returns a list of countries based on different filtering criteria, like for example the country codes. However, the response with `NOT_FOUND` is never returned. The problem is that, even if the HTTP requests provide invalid inputs (e.g., a country code that does not exist), the `countries` list is wrongly populated with null values, and so the list is never empty. This is an interesting bug in the API, which then result in a crash (i.e., a returned 500 status code), as `parsedCountries` throws an exception. However, until this fault in the API is fixed, all these statements with `NOT_FOUND` are technically dead-code that cannot be reached with the tests.

Similarly to *rest-ncs*, the fuzzing of this API might be considered as a *solved* problem, at least for its current faulty version.

A.18 scout-api

On this API, white-box EvoMaster achieves slightly worse results (53.4% vs. 54.7%), although the difference is not statistically significant (based on 10 runs). Large part of its codebase (more than a third) cannot be executed. For example, *scout-api* can start in the background a thread to fetch and analyse some data, independently from the RESTful endpoints (i.e., the whole module `data-batch-jobs`). Such thread is deactivated by default, and so nearly 30% of the whole codebase is not executed. Another non-negligible part of the codebase (around 5%) is related to authentication using OAuth via Google APIs (*scout-api* provides different ways to authenticate).

There is a large part of the codebase that could be executed, but it is not due to an `if` statement at the beginning of one of the HTTP endpoints. This is shown in Figure 17. Here an incoming JSON payload is unmarshalled into a `MediaFile` object, which has a string field that is treated as a URI. Most random strings are a valid URI, as a URI can just be a name. But, it is extremely unlikely that a random string would represent a URI with a valid scheme component, and so `uri.getScheme()` returns null. However, this is a case that likely would be trivial to solve with taint analysis, e.g., by extending the techniques in [34] to support URI objects. This does not mean though that much higher coverage would be achieved, it depends on all the remaining code that is executed once that predicate is satisfied.

Even without code analyses, this type of challenge could also be addressed with black-box techniques. For example, in this API the string field representing the URI is called `uri`. An analysis of the names of the fields can be used to possibly infer their expected type. For example, any field

```
router.get('/:id', cache(300), async (ctx) => {
  const result = await Core.findById(ctx.params.id);
  if (!result) {
    ctx.throw(404);
  }
  ctx.status = 200;
  ctx.body = result;
});
```

Fig. 18. Snippet of the handle for the `GET /cores/:id` endpoint in the API *spacex-api*.

```
router.post('/', auth, authz('core:create'), async (ctx) => {
  try {
    const core = new Core(ctx.request.body);
    await core.save();
    ctx.status = 201;
  } catch (error) {
    ctx.throw(400, error.message);
  }
});
```

Fig. 19. Snippet of the handle for the `POST /cores` endpoint in the API *spacex-api*.

whose name starts or ends with `uri` could be treated as an actual URI when data is generated. Also, some types are quite common in RESTful APIs, like strings representing URLs and dates. Instead of sampling strings completely at random, it could make sense to sample with some probability strings with given types that are commonly used. This might now work in all cases, but it is something worth to investigate.

A.19 spacex-api

Similarly to *realworld-app*, in this API there is only a small difference between EvoMaster black-box and Schemathesis (c8 instrumentation: 84.7% vs. 85.4%), and between grey-box and white-box EvoMaster (EvoMaster instrumentation without bootstrap coverage: 41.5% vs. 41.5%, but with $\hat{A}_{12} = 0.58$). There are two main issues affecting the coverage results here.

First, this API requires authentication, where each API endpoint requires specific authorization roles to be able to be executed. For these experiments, a user with the right credentials was created in the database, and the fuzzers were given the authorization info to send HTTP messages on behalf of this user. The setting up of this user was done manually, in a script. However, after running the experiments and analyzing the generated tests, we realized that some authorization roles were missing (e.g., `capsule:create`). So, few endpoints were not covered due to misconfigured authentication.

The second main issue is related to the database. Figure 18 shows an example in which a record is search by id. No test generated by EvoMaster was able to make a call that returned a 200 HTTP status code. For white-box testing, EvoMaster fails to do this because it is not able to analyze queries on MongoDB databases. Even with black-box testing, it could be possible to create a new record with a `POST`, and use its id for a following `GET`. Fuzzers can exploit this kind of information to create sequences of HTTP calls where ids are linked. But, this all depends on how the link is defined. Figure 19 shows the implementation of the `POST` endpoint to create such record. EvoMaster has no issue to fully cover the code of such endpoint, and create valid records. Also, EvoMaster uses different strategies to *link* endpoints that manipulate the same resources [30, 76]. But those are based on the recorded interactions with the database (not done here for NodeJS and MongoDB), and on best practices in RESTful API design. For example, it is a recommended practice that `POST`s on collections of resources (e.g., `/cores`) create a new resource, whose `id` (chosen on the server, to guarantee it is unique, unless it is a UUID) is present in the `location` HTTP header of the response (e.g., `location: /cores/42`). EvoMaster can use these location headers to create `POST` requests followed with the appropriate linked `GET`. However, on this API, the `POST` implementation in Figure 19 does not seem to follow best practices in REST API design. For example, it does not setup the `location` header in the response, and neither it returns the generated `id` (e.g., in the body payload). However, it could still be possible to test this kind of API by doing the following: create a new resource with `POST /cores`, followed by a `GET` of the whole collection (i.e., `GET /cores`), and then extract the `id` fields from this response to call `GET /cores/:id` with a valid `id`. Note: this is assuming that the collection is empty. If it is not, then the first `POST` is not even necessary. However, this also assumes that the name of the field representing the id is the same (or at least very similar) in both the body payload and endpoint path parameter (so they can be matched).

REFERENCES

[1] [n.d.]. Amazon Gateway API. https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-rest-api.html. Online, Accessed May 20, 2022.
[2] [n.d.]. APIFuzzer — HTTP API Testing Framework. https://github.com/KissPeter/APIFuzzer.
[3] [n.d.]. ApiTester. https://github.com/opendata-for-all/api-tester. Online, Accessed May 20, 2022.
[4] [n.d.]. bBOXRT. https://git.dei.uc.pt/cnl/bBOXRT.
[5] [n.d.]. C8. https://github.com/bcoe/c8. Online, Accessed May 20, 2022.
[6] [n.d.]. EvoMaster. https://github.com/EMResearch/EvoMaster.
[7] [n.d.]. EvoMaster Benchmark (EMB). https://github.com/EMResearch/EMB. Online, Accessed May 20, 2022.
[8] [n.d.]. Google Drive API. https://developers.google.com/drive/api/v3/about-sdk. Online, Accessed May 20, 2022.
[9] [n.d.]. Hibernate. http://hibernate.org. Online, Accessed May 20, 2022.
[10] [n.d.]. JaCoCo. https://www.jacoco.org/. Online, Accessed May 20, 2022.
[11] [n.d.]. Language-agnostic HTTP API Testing Tool. https://github.com/apiaryio/dredd.
[12] [n.d.]. LinkedIn API. https://docs.microsoft.com/en-us/linkedin/. Online, Accessed May 20, 2022.
[13] [n.d.]. OpenAPI/Swagger. https://swagger.io/.
[14] [n.d.]. Programmable Web. https://www.programmableweb.com/. Online, Accessed May 20, 2022.
[15] [n.d.]. Reddit API. https://www.reddit.com/dev/api. Online, Accessed May 20, 2022.
[16] [n.d.]. RestCT. https://github.com/GIST-NJU/RestCT.
[17] [n.d.]. RESTest. https://github.com/isa-group/RESTest.
[18] [n.d.]. RESTler. https://github.com/microsoft/restler-fuzzer.
[19] [n.d.]. RestTestGen. https://github.com/SeUniVr/RestTestGen.
[20] [n.d.]. Schemathesi - Command Line Interface. https://schemathesis.readthedocs.io/en/stable/cli.html.
[21] [n.d.]. Schemathesis: Property-based testing for API schemas. https://schemathesis.readthedocs.io/.
[22] [n.d.]. Tcases for OpenAPI: From REST-ful to Test-ful. https://github.com/Cornutum/tcases/tree/master/tcases-openapi.
[23] [n.d.]. Twitter API. https://developer.twitter.com/en/docs/twitter-api. Online, Accessed May 20, 2022.
[24] [n.d.]. WireMock. https://wiremock.org/. Online, Accessed May 20, 2022.
[25] Nasser Albunian, Gordon Fraser, and Dirk Sudholt. 2020. Causes and effects of fitness landscapes in unit test generation. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. 1204--1212.
[26] A. Arcuri. 2009. Full Theoretical Runtime Analysis of Alternating Variable Method on the Triangle Classification Problem. In *International Symposium on Search Based Software Engineering (SSBSE)*. 113--121.
[27] A. Arcuri. 2009. Insight Knowledge in Search Based Software Testing. In *Genetic and Evolutionary Computation Conference (GECCO)*. 1649--1656.
[28] Andrea Arcuri. 2017. RESTful API Automated Test Case Generation. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 9--20.
[29] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[30] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.

[31] Andrea Arcuri. 2020. Automated Black-and White-Box Testing of RESTful APIs With EvoMaster. *IEEE Software* 38, 3 (2020), 72--78.

[32] A. Arcuri and L. Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)* 24, 3 (2014), 219--250.

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

[34] Andrea Arcuri and Juan P Galeotti. 2021. Enhancing Search-based Testing with Testability Transformations for Existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1--34.

[35] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. 2021. EvoMaster: A Search-Based System Test Generation Tool. *Journal of Open Source Software* 6, 57 (2021), 2153.

[36] Andrea Arcuri, ZhangMan, asmab89, Bogdan, Amid Golmohammadi, Juan Pablo Galeotti, Seran, Alberto Martín López, Agustina Aldasoro, Annibale Panichella, and Kyle Niemeyer. 2022. *EMResearch/EvoMaster:*. https://doi.org/10.5281/zenodo.6651631

[37] Andrea Arcuri, ZhangMan, Amid Golmohammadi, and asmab89. 2022. *EMResearch/EMB:*. https://doi.org/10.5281/zenodo.6106830

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

[39] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020. Checking security properties of cloud service rest apis. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 387--397.

[40] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. Evolutionary-based Automated Testing for GraphQL APIs. In *Genetic and Evolutionary Computation Conference (GECCO)*.

[41] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. White-Box and Black-Box Fuzzing for GraphQL APIs. https://doi.org/10.48550/ARXIV.2209.05833

[42] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *44th IEEE/ACM International Conference on Software Engineering, ser. ICSE*, Vol. 22.

[43] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology (IST)* 104 (2018), 207--235.

[44] 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, 226--236.

[45] 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.

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

[47] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API Data Fuzzing. In *ACM Symposium on the Foundations of Software Engineering (FSE) (ESEC/FSE 2020)*. ACM, 725–736.

[48] 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*. 312--323.

[49] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2022. Testing RESTful APIs: A Survey. *arXiv preprint arXiv:2212.14604* (2022).

[50] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2022. Testing RESTful APIs: A Survey. https://doi.org/10.48550/ARXIV.2212.14604

[51] M. Harman and P. McMinn. 2010. A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search. *IEEE Transactions on Software Engineering (TSE)* 36, 2 (2010), 226--247.

[52] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving Semantics-Aware Fuzzers from Web API Schemas. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 345--346.

[53] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering (TSE)* 37, 5 (2011), 649--678.

[54] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[55] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. https://doi.org/10.48550/ARXIV.2204.08348

[56] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A black box tool for robustness testing of REST services. *IEEE Access* 9 (2021), 24738--24754.

[57] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: Model-based RESTful API Testing with Execution Feedback. In *ACM/IEEE International Conference on Software Engineering (ICSE)*.

[58] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891.

[59] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. 2022. On the Faults Found in REST APIs by Automated Test Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1--43.

[60] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. 2021. Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 231--241.

[61] Alberto Martin-Lopez, Sergio Segura, Carlos Muller, and Antonio Ruiz-Cortés. 2021. Specification and automated analysis of inter-parameter dependencies in web APIs. *IEEE Transactions on Services Computing* (2021).

[62] 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*.

[63] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: Automated Black-Box Testing of RESTful Web APIs. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 682--685.

[64] A Giuliano Mirabella, Alberto Martin-Lopez, Sergio Segura, Luis Valencia-Cabrera, and Antonio Ruiz-Cortés. 2021. Deep Learning-Based Prediction of Test Input Validity for RESTful APIs. In *2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*. IEEE, 9--16.

[65] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. 2018. An analysis of public REST web service APIs. *IEEE Transactions on Services Computing* (2018).

[66] Sam Newman. 2015. *Building Microservices*. " O'Reilly Media, Inc.".

[67] Juan Carlos Alonso Valenzuela, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. 2022. ARTE: Automated Generation of Realistic Test Inputs for Web APIs. *IEEE Transactions on Software Engineering* (2022).

[68] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

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

[70] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. 2013. Characteristic studies of loop problems for structural test generation via symbolic execution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 246--256.

[71] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. Precise identification of problems for structural test generation. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 611--620.

[72] Man Zhang and Andrea Arcuri. 2021. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021).

[73] Man Zhang and Andrea Arcuri. 2022. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *arXiv preprint arXiv:2205.05325* (2022).

[74] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2022. White-box Fuzzing RPC-based APIs with EvoMaster: An Industrial Case Study. https://doi.org/10.48550/ARXIV.2208.12743

[75] Man Zhang, Asma Belhadi, and Andrea Arcuri. 2022. JavaScript Instrumentation for Search-Based Software Testing: A Study with RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[76] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2021. Resource and dependency based test case generation for RESTful Web services. *Empirical Software Engineering* 26, 4 (2021), 1--61.