

Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?

Alberto Martin-Lopez
SCORE Lab, I3US Institute
Universidad de Sevilla
Seville, Spain
alberto.martin@us.es

Andrea Arcuri
Kristiania University College
Oslo Metropolitan University
Oslo, Norway
andrea.arcuri@kristiania.no

Sergio Segura
SCORE Lab, I3US Institute
Universidad de Sevilla
Seville, Spain
sergiosegura@us.es

Antonio Ruiz-Cortés
SCORE Lab, I3US Institute
Universidad de Sevilla
Seville, Spain
aruiz@us.es

Abstract—Automated test case generation for RESTful APIs is a thriving research topic due to their critical role in software integration. Testing approaches can be divided into black-box and white-box. Black-box approaches exploit the API specification for the generation of test cases, while white-box approaches can also leverage the source code. Both strategies have shown great promise, but they have not been fully compared yet, hindering the selection of the right tool for the job. In this paper, we report on our experience comparing black-box and white-box test case generation for RESTful APIs using the state-of-the-art tools RESTest (black-box) and EvoMaster (white-box). Also, we propose integrating both approaches by using black-box test cases as the seed for white-box search-based test case generation. Evaluation results on four RESTful APIs involving over 40 million API calls show that there is no *one-size-fits-all* strategy. More importantly, the combination of black-box and white-box yielded the best results in most case studies in terms of code coverage and fault finding, paving the way for better tools integrating the best of both perspectives. As a result of our work, we provide lessons learned and open challenges for guiding the use and further development of current tool support.

Index Terms—REST, API, web service, SBST, seeding

I. INTRODUCTION

Web Application Programming Interfaces (APIs) allow heterogeneous systems to interact over the network [1], [2]. Web APIs are rapidly proliferating as a key element to foster software reusability, integration, and innovation, enabling new consumption models such as wearables and smart home apps. Companies such as Facebook, Twitter, Google, eBay or Netflix receive billions of API calls everyday from thousands of different third-party applications and devices, which constitutes more than half of their total traffic [1]. Modern web APIs typically adhere to the *REpresentational State Transfer* (REST) architectural style [3], being referred to as *RESTful* APIs. RESTful APIs provide a standard mechanism to implement create, retrieve, update, and delete (CRUD) operations over resources (e.g., a YouTube video) in a distributed way. The widespread use of RESTful APIs is reflected in the size of popular API directories such as ProgrammableWeb [4], currently indexing over 24K RESTful APIs from domains such as shopping, finances, social, or telephony.

Testing RESTful APIs is critical due to their key role in software integration. A bug in an organization’s API could have a huge impact both internally (services relying on

that API) and externally (third-party applications and end users). To address this challenge, numerous approaches for test case generation for RESTful APIs have emerged in recent years [5]–[12]. These approaches can be divided into black-box and white-box. Black-box approaches [5], [6], [8]–[12] exploit the specification of the API, typically in the OpenAPI Specification (OAS) [13] format, to drive the generation of test cases in the form of (pseudo-)random API calls. They often use custom data generators, manually implemented by the users, based on domain knowledge of the application. White-box approaches [7] leverage the API specification and its source code to generate test cases that cover as much code as possible, trying to minimize the amount of manual work from the users. Both approaches try to identify server errors (i.e., 5XX HTTP status codes) and mismatches with the API specification.

Both test case generation strategies for RESTful APIs, black-box and white-box, have been largely evaluated in isolation, showing their own merits and limitations. However, they have not been fully compared yet. Thus, many questions remain unanswered regarding the scope and performance of each approach. Besides this, both approaches have followed completely different paths so far, and thus the potential benefits of their combination remain unexplored.

In this paper, we report on our experience comparing black-box and white-box test case generation techniques for RESTful APIs. Specifically, we use the state-of-the-art tools RESTest [14] (black-box) and EvoMaster [7] (white-box). Additionally, we propose the integration of both approaches by using black-box test cases—derived from the API specification—as the seed for search-based test case generation—leveraging the source code of the API. We report on the results of an extensive empirical evaluation comparing the effectiveness and performance of these three approaches: black-box, white-box and hybrid (black-box + white-box) on four open-source RESTful APIs of different sizes and complexity. The results show that our novel hybrid approach performed best in most scenarios. In terms of fault finding, the hybrid approach was the only one capable of uncovering bugs in all systems under test (SUTs). We provide an in-depth discussion and analysis of the results obtained by each technique, and possible causes leading to such results. Our work contributes to a better understanding of the strengths and

limitations of black-box and white-box test case generation for RESTful APIs, and shows the potential of combining both strategies. However, more work and experimentation will be required in the future to complement our results including the use of different testing techniques, tools, and case studies.

After explaining the background (Section II) and discussing related work (Section III), this paper provides the following original research and engineering contributions:

- A novel *hybrid* testing approach based on the use of black-box test cases as the seed for white-box test generation, and its implementation in EvoMaster [7] (Section IV).
- A comparison of black-box, white-box, and hybrid techniques for test case generation for RESTful APIs in terms of code coverage and fault finding using two different state-of-the-art tools and four open-source APIs (Section V).
- Lessons learned from the comparison of the three techniques, including practical guidelines for the selection of the right approach depending on several factors such as the size of the SUT and the available resources (Section VI).
- A list of challenges for the three test case generation approaches under comparison derived from the results of the study (Section VII).

We address threats to validity in Section VIII and conclude the paper in Section IX.

II. BACKGROUND

A. RESTful Web APIs

REST [3] is a software architectural style for building distributed systems. Most current web services follow the principles defined by REST and are referred to as *RESTful* (or simply REST) web services. RESTful web services provide a standard approach to interact with resources over the network. A *resource* is any piece of data that can be exposed to the Web, for example, a document (Google Drive API [15]), a picture (Flickr API [16]), or even a tweet (Twitter API [17]). Resources can be accessed and manipulated by means of CRUD operations. These operations can be invoked by sending HTTP requests to specific API endpoints, each of which is identified by an HTTP method (e.g., GET) and a path (e.g., /documents). A RESTful API may be composed of one or more RESTful services.

RESTful web APIs are usually described with interface description languages like the OpenAPI Specification (OAS) [13]. OAS is heavily used nowadays for automating certain tasks in the API lifecycle such as code generation [18], monitoring [19] and testing [14]. An OAS document (also called *schema*) describes a RESTful API in terms of the allowed inputs (HTTP requests) and the expected outputs (HTTP responses). Figure 1 depicts an extract of an OAS schema, taken from LanguageTool, a proofreading API [20]. As illustrated, the POST /check endpoint allows to detect mistakes in a text (line 4). The specification details the

```

1 paths:
2   /check:
3     post:
4       summary: Check a text
5       consumes:
6         - application/x-www-form-urlencoded
7       parameters:
8         - name: text
9           description: The text to be checked. This or 'data' is required.
10          in: formData
11          type: string
12          required: false
13         - name: data
14           description: The text to be checked, given as a JSON document.
15          in: formData
16          type: string
17          required: false
18         - name: language
19           description: A language code like 'en-US', 'de-DE', 'fr', or 'auto'
20           to guess the language automatically.
21          in: formData
22          type: string
23          required: true
24         - name: motherTongue
25           description: A language code of the user's native language.
26          in: formData
27          type: string
28          required: false
29         - name: preferredVariants
30           description: Comma-separated list of preferred language variants.
31          in: formData
32          type: string
33          required: false
34         - name: enabledRules
35           description: IDs of rules to be enabled, comma-separated.
36          in: formData
37          type: string
38          required: false
39         - name: disabledRules
40           description: IDs of rules to be disabled, comma-separated.
41          in: formData
42          type: string
43          required: false
44         - name: enabledCategories
45           description: IDs of categories to be enabled, comma-separated.
46          in: formData
47          type: string
48          required: false
49         - name: disabledCategories
50           description: IDs of categories to be disabled, comma-separated.
51          in: formData
52          type: string
53          required: false
54         - name: enabledOnly
55           description: Enable only specified rules and categories.
56          in: formData
57          type: boolean
58          required: false
59          default: false
60       x-dependencies:
61         - OnlyOne(text, data);
62         - IF preferredVariants THEN language='auto';
63         - IF enabledOnly==true THEN NOT (disabledRules OR disabledCategories);
64         - IF enabledOnly==true THEN (enabledRules OR enabledCategories);
65       responses:
66         '200':
67           description: The result of checking the text
68           schema:
69             $ref: '#/definitions/LanguageCheck'

```

Fig. 1. OAS excerpt from the LanguageTool API.

information about the operation parameters (lines 7-59), such as their data type (line 57), whether they are required or optional (line 58) and default values (line 59). For every operation, the set of expected responses is also described (lines 65-69), including their HTTP status code (line 66) and their format (line 69). Optionally, when the operation contains inter-parameter dependencies (e.g., mutually exclusive parameters), these can be specified with the IDL4OAS extension [21] (lines 60-64). For example, the input text fragment can be provided as a plain string using parameter `text` (line 8) or as a JSON document using parameter `data` (line 13), but only one of them must be set (line 61). According to a recent study by Martin-Lopez et al. [22], 4 out of every 5 industrial APIs contain these inter-parameter dependencies.

B. RESTful API Test Case Generation

Testing a RESTful API at the system level involves generating HTTP requests and asserting their responses. A test case comprises one or more requests.

Black-box testing approaches leverage the specification of the API under test (e.g., an OAS document) to automatically generate test cases. However, the specification may not suffice to generate *realistic* test inputs, or it may simply be wrong. For example, the `language` parameter in Figure 1 is defined as a `string`, but only properly formed language codes will be accepted. Manual work is typically needed to add this missing information. Another drawback of black-box testing is that, due to the lack of control over the SUT, test cases are more costly: stateful interactions (e.g., updating and deleting a resource) can only be achieved if the SUT is in the proper state (e.g., a resource exists). This means that every test case must not only exercise the SUT in a certain way, but also set it up with previous HTTP calls. As its key strength, black-box testing does not require access to the source code, and can therefore be potentially applied to any API regardless of how it is implemented (e.g., the programming language) or where it is deployed, locally or remotely.

White-box testing approaches for RESTful APIs exploit the source code of the SUT to generate test cases. Advanced heuristics such as taint checking and testability transformations [23] can be applied to maximize objectives such as code coverage and fault finding. In contrast to black-box testing, white-box testing can only be applied when the code of the API is available, and it is therefore implementation-dependent, e.g., the heuristics to measure the branch distance must be adapted to every programming language [23]. However, this limitation of white-box testing is also its best asset, since the source code of the system may contain valuable information for the generation of test cases. Moreover, since there is full control of the SUT, test cases are less costly. For example, the SUT can be set to a specific state without the need of using extra HTTP requests (as required in black-box testing) [24].

C. RESTest

RESTest [14] is a black-box testing framework for RESTful web APIs. It follows a model-based testing approach [25]: based on the OAS specification of the API under test (so-called *system model*), it automatically generates a *test model*, which can be manually augmented to tailor the testing process. Both models are subsequently used to drive the automated generation of test cases.

RESTest relies on *test data generators* to generate input data. These are automatically configured for every API parameter (e.g., a generator of English words for `string` parameters), nevertheless, it is possible to manually configure more realistic generators according to the parameter’s domain (e.g., a generator of valid language codes for the `language` parameter in Figure 1). Regarding test case generation, RESTest leverages constraint programming solvers to automatically generate requests satisfying the inter-parameter dependencies of the API. Dependencies must be defined as a part of the OAS specification using the IDL4OAS extension [21].

We chose RESTest, and in particular its constraint-based test case generator, as a good representative of black-box testing tools due to its support for data generators and inter-parameter

dependencies. Both features have shown to be effective in finding real-world bugs in industrial APIs such as YouTube and Yelp [8].

D. EvoMaster

EvoMaster [7] is a white-box testing tool for RESTful web APIs. It integrates a search-based technique for the automated generation of system-level test cases. The default search algorithm used in EvoMaster is the Many Independent Objective (MIO) algorithm [26]. In MIO, test cases are evolved and evaluated independently, and only at the end of the search, a test suite is constructed by choosing the combination of test cases that cover more targets (e.g., source code statements and HTTP status codes). For the representation of the problem, EvoMaster considers a *solution* as a system-level test suite for the RESTful API, which is composed of one or more test cases, i.e., *individuals*. Each test case is a sequence of one or more HTTP calls. Throughout the search, EvoMaster mutates the test cases either by modifying their structure (i.e., adding or removing HTTP calls) or the data of one specific HTTP call.

EvoMaster comprises two main components: the *core*, which handles the generation and evolution of test cases; and a *controller* library, used for the manual setup of the SUT, e.g., it is responsible for instrumenting the SUT so that metrics such as the branch distance can be collected.

EvoMaster also provides some basic support for black-box testing, essentially random generation with no support for data generators or inter-parameter dependencies. However, the black-box configuration in EvoMaster performs significantly worse than the white-box strategy implemented in the own tool [5]. To the best of our knowledge, EvoMaster is the only tool supporting white-box system-level testing of RESTful web APIs.

III. RELATED WORK

RESTful API testing is an active field of research nowadays. Testing approaches can be divided into black-box and white-box, the former being more common than the latter. Black-box approaches mainly differ in three aspects: (1) how they generate *stateful interactions* in the SUT; (2) how they generate the *input data* to feed into the API; and (3) what *test oracles* they use.

Regarding the generation of stateful interactions, some approaches do not explicitly support them since they test the API operations in isolation [12]. Other tools, like RESTest, reuse data from previous API operations’ responses (e.g., a resource identifier) for subsequent requests to different operations, thus eventually achieving stateful interactions [14]. Lastly, a number of approaches generate structured sequences of requests, either by using predefined templates (e.g., POST-PUT-GET) [11], [27], or by dynamically deciding the next request to execute based on the result of the last one (e.g., if a POST was successful, execute a GET, otherwise execute a POST again) [9], [10].

Regarding the generation of input data, several strategies can be followed. Ed-douibi et al. [12] proposed extracting default and example values from the OAS specification of the API under test. Atlidakis et al. [11] introduced RESTler, a fuzzing tool for RESTful APIs. RESTler uses fuzzing dictionaries for each data type (e.g., 0 and 1 for `integer` parameters). Other tools like QuickREST [9] or RESTest [14] support the use of customizable test data generators (e.g., a generator of real coordinates for a mapping API). Lastly, Viglianisi et al. [10] proposed extracting values from API responses and using them as inputs in subsequent API requests. This last approach is only applicable for APIs containing *producer-consumer* relationships among their operations (i.e., an operation returns some data that another operation needs as input), which is not the case for some of the APIs used as case studies in this paper, for instance.

Lastly, regarding the test oracles used, most testing approaches rely on the presence of 5XX status codes (i.e., server errors) and the conformance with the OAS schema [9]–[12]. Other authors have proposed more thorough oracles such as checking the status code [8], metamorphic relations [6] and security properties [28].

White-box testing approaches for RESTful APIs are less common in the literature. Arcuri [7] advocates for a search-based approach, implemented in the open-source tool EvoMaster (discussed in the previous section). EvoMaster integrates diverse advanced heuristics such as testability transformations [23] and the handling of SQL databases [24]. Such heuristics drive the generation of input data that is continuously evolved, aimed at generating test cases that cover as much code and find as many faults (5XX status codes) as possible. Besides system-level testing, other tools such as EvoSuite [29] and Randoop [30] allow to automatically generate unit tests for Java programs, but they are not specifically tailored for RESTful web APIs.

Regarding the comparison of black-box and white-box testing techniques (one of the main contributions of our work), only one paper has previously addressed this matter: Arcuri [5] compared black-box and white-box testing in eight case studies with EvoMaster, showing that the latter always outperformed the former both in terms of code coverage and fault finding. However, the black-box approach under comparison was rather naive, i.e., basic random generation. Compared to previous work, we evaluate two different state-of-the-art tools for black-box and white-box test case generation of RESTful APIs, RESTest and EvoMaster, which provides a fairer comparison. On the one hand, RESTest is specifically designed to exploit the information in the API specification using constraint-programming techniques. On the other hand, EvoMaster leverages the source code for the generation of effective test cases using search-based algorithms. Additionally, we propose a hybrid approach that combines both strategies, and show its potential to outperform black-box and white-box testing in isolation.

The hybrid approach presented in this paper is based on existing open-source testing tools for RESTful APIs, namely,

RESTest [14] and EvoMaster [7]. We extended both tools to enable their inter-operability, and we introduce *system-level test case seeding* as a way to enhance search-based testing of RESTful web APIs. Previous authors have proposed multiple seeding strategies for enhancing unit test generation [31], such as reusing previous solutions or seeding values observed at runtime. These strategies could be complementary to our approach, and they could be used to further improve system-level test case generation.

IV. BLACK-BOX + WHITE-BOX TEST CASE GENERATION

We present a novel *system-level test case seeding* approach, where the test cases generated by the black-box approach are seeded into the search algorithm used by the white-box approach. Therefore, instead of starting “from scratch”, the search starts with a (potentially) thorough test suite, which is subsequently evolved aiming to maximize code coverage and faults found.

A. Motivation

Let us consider the operation for proofreading a text in the LanguageTool API, depicted in Figure 1. This operation accepts 10 input parameters and imposes numerous constraints on them, including both inter-parameter dependencies (e.g., if parameter `preferredVariants` is used, then `language` must be set to ‘`auto`’) and constraints on single parameters (e.g., parameter `motherTongue` must be a properly formed language code). In order for an API call to be *valid* (i.e., to obtain a 2XX HTTP status code), it must satisfy all these constraints. These constraints are reflected in the source code of the system as multiple branch conditions. If any of these conditions is not satisfied, processing terminates and a 400 status code is immediately returned.

Automatically generating a valid API call is not trivial. A valid call must include the specific combination of API parameters (e.g., `motherTongue`) set to the specific values (e.g., ‘`en-US`’) such that all the input constraints present in the API are satisfied. This involves generating test inputs that satisfy the numerous branch conditions found in the source code before exercising the actual functionality of the API, i.e., proofreading the text. Even when using a search algorithm to leverage the source code, a large number of iterations may be required.

Listing 1 depicts a test case¹ generated by EvoMaster for the operation in Figure 1 at the beginning of the search. This test case was rejected by the API, because it did not use a valid value for parameter `language`. Listing 2, on the other hand, shows a test case generated by RESTest. This test case did obtain a successful response, since it passed all the input validation logic implemented in the system. RESTest can generate this type of requests automatically because: (1) it uses realistic data generators for each parameter (e.g., a language code generator for parameter `language`); and (2) it analyzes the dependencies expressed with IDL4OAS (lines

¹Due to space constraints, we do not show the assertions on the returned responses in this and the subsequent listings.

```

1 @Test
2 public void test_17() {
3     RestAssured.given()
4         .contentType("application/x-www-form-urlencoded")
5         .formParam("enabledOnly", "false")
6         .formParam("text", "gh53wgh2")
7         .formParam("enabledRules", "bnyt34yhav")
8         .formParam("language", "b9plsv")
9     .when()
10        .post("/check");
11 }

```

Listing 1. Random test case generated by EvoMaster at the start of the search.

```

1 @Test
2 public void test_t4idj2asd3s6_check() {
3     RestAssured.given()
4         .contentType("application/x-www-form-urlencoded")
5         .formParam("preferredVariants", "en-GB,ja-JP,sk-SK")
6         .formParam("motherTongue", "sk-SK")
7         .formParam("language", "en")
8         .formParam("text", "stick around nose")
9     .when()
10        .post("/check");
11 }

```

Listing 2. Realistic test case generated by RESTest.

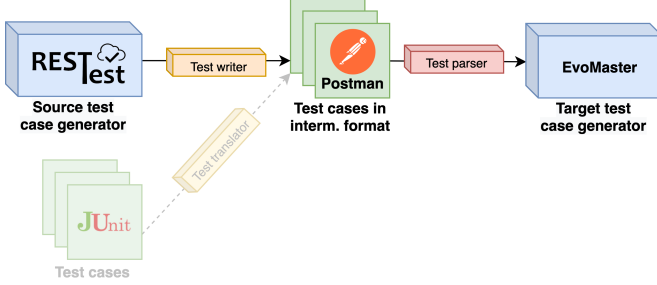


Fig. 2. Test-to-test translation approach.

60-64 from Figure 1) and uses constraint solvers to generate valid combinations of parameters and values [8].

B. Implementation

We implemented our approach as an extension of the open-source tools RESTest [14] and EvoMaster [7]. RESTest generates the initial population of test cases (seed), and EvoMaster subsequently evolves them. This entails a technical challenge: how to transform RESTest’s test cases into a format that EvoMaster can handle? Figure 2 depicts the proposed approach for ensuring inter-operability between different testing approaches and test case formats. Rather than a direct translation of the source test cases into the target format, we advocate for translating the seed into an intermediary format which can be subsequently parsed. This has several benefits: (1) already existing test suites (e.g., those manually created by the SUT developers) can be used as a seed by implementing specific *test translators*; (2) seeds can be generated by existing *source test case generators* (e.g., RESTest [14]) by implementing *test writers* to the intermediary format; (3) only one *test parser* is needed for feeding the seed into the search algorithm, thereby allowing the *target test case generator* (e.g., EvoMaster [7]) to leverage the seed.

The intermediary format selected for our approach is Postman. *Postman* [32] is an industry-standard platform for web API development. It provides support for several tasks throughout the API lifecycle, such as the creation of API clients and documentation, or even basic automated testing. Postman allows to create test suites for RESTful APIs and export them to so-called *Postman collections*, in JSON format. This makes our approach readily applicable in practice in those cases where a previous Postman test suite already exists. When the test suite is in a different format (e.g., JUnit [33] using the REST Assured library [34]), only a test translator

TABLE I
CASE STUDIES USED IN THE EVALUATION.

SUT	Classes	LOCs	Endpoints
RESTcountries	20	1,450	22
YouTubeMock	30	3,371	1
Ohsome	75	9,813	122
LanguageTool	1,021	162,341	1
Total	1,146	176,975	147

needs to be created (grayed-out area of Figure 2). For the implementation of our approach, we extended both RESTest and EvoMaster with a Postman test writer and a Postman test parser, respectively.

V. EMPIRICAL STUDY

The aim of this practical experience report is to provide empirical evidence on the applicability and the limitations of black-box, white-box and hybrid testing techniques for RESTful APIs. To this end, we pose the following research questions:

- **RQ1:** How do black-box constraint-based testing and white-box search-based testing compare in terms of code coverage and fault finding?
- **RQ2:** How does our novel hybrid approach compare with the black-box and white-box techniques in terms of code coverage and fault finding?

A. Case Studies

Table I summarizes the main features of the case studies selected, including their name and size in terms of Java classes, lines of code (LOCs) and HTTP endpoints. RESTcountries² is an API for querying information about countries based on several filters such as the code, the currency or the continent. YouTubeMock³ is an open-source implementation of the search operation of the YouTube API [35]. Ohsome⁴ allows to consume OpenStreetMap data [36] of the whole world and to get aggregated statistics about it. Lastly, LanguageTool⁵ is a proofreading API with support for more than 20 languages.

We selected RESTful APIs belonging to varied application domains and with very different sizes. These APIs are non-trivial, since they deal with input constraints with varied

²<https://github.com/apilayer/restcountries>

³<https://github.com/opensourcingapis/YouTubeMock>

⁴<https://github.com/GIScience/ohsome-api>

⁵<https://github.com/language-tool-org/language-tool>

TABLE II
BUDGETS AND SEEDS USED FOR EACH SUT.

SUT	Budgets		Seeds		
	Low	High	Min	Med	Max
RESTcountries	5K	100K	22	220	2.2K
YouTubeMock	10K	200K	10	100	1K
Ohsome	30K	600K	122	1.22K	12.2K
LanguageTool	100K	1M	100	1K	10K

degrees of complexity, such as the inter-parameter dependencies present in Figure 1 (lines 60-64). In order to make a fair comparison between black-box and white-box testing, we selected only *stateless* APIs, since the API state cannot be reset between test cases in black-box testing (see Section II-B for details). For each API under test, we studied its documentation and extended its OAS schema to explicitly define inter-parameter dependencies using IDL4OAS [21].

B. Experimental Setup

For each case study, we evaluated three techniques, namely:

- **BB**: The black-box approach, that is, the constraint-based testing technique implemented in RESTest.
- **WB**: The white-box approach, namely, the search-based technique implemented in EvoMaster.
- **HYBRID**: Our novel hybrid approach, implemented as an extension of both tools, i.e., RESTest’s test cases are seeded into EvoMaster.

Table II shows the budgets with which we evaluated each testing technique. For *HYBRID*, the size of the seeds used is also shown. Both metrics are measured in terms of HTTP calls, and they were adjusted according to the SUT size (in terms of LOCs and number of endpoints) since it strongly determines the time required for white-box heuristics to evolve toward good solutions. For instance, YouTubeMock was assigned twice the budget of RESTcountries because it has twice as many LOCs, and Ohsome was assigned a minimum seed of 122 HTTP calls because it contains 122 endpoints. For *BB*, we considered only the minimum of the two budgets used for *WB* and *HYBRID*, to make the experiments affordable.⁶ This is in line with how black-box testing is performed in practice, where resources are often constrained due to restrictive quota limitations imposed by commercial APIs [8], [37].

For each SUT, we performed a total of 9 experiments: one for *BB* (the minimum budget), two for *WB* (both budgets), and six for *HYBRID* (two budgets \times three seed sizes). In order to account for the randomness of the algorithms, we repeated each experiment 10 times, following the guidelines in [38]. As an exception, some configurations were run only once, either because they took too much time to complete (e.g., 24 hours) or due to technical limitations of the tools used (discussed in Sections VI and VII).⁷ Given four SUTs, this led to $4 \times 9 \times$

⁶RESTest does not shrink test suites, and so it is time-consuming to measure the coverage of millions of test cases, which must be done manually and dozens of times, as explained further on.

⁷Configurations run only once: for Ohsome, *BB*; for LanguageTool, *WB* with the high budget, *HYBRID* with the high budget and all seed sizes, and *HYBRID* with the low budget and largest seed.

TABLE III
RQ1: COMPARISON OF *BB* AND *WB*.

SUT	Coverage (%)				5XX			
	<i>BB</i>	<i>WB</i>	\hat{A}_{12}	<i>p</i> -value	<i>BB</i>	<i>WB</i>	\hat{A}_{12}	<i>p</i> -value
RESTcountries	74.9	73.6	0.68	0.136	0.0	0.5	0.25	0.014
YouTubeMock	81.9	26.0	1.00	< 0.001	0.0	0.0	0.50	NaN
Ohsome	74.0	20.4	1.00	0.111	122.0	0.0	1.00	0.004
LanguageTool	46.6	35.3	1.00	< 0.001	1.0	1.0	0.50	NaN

$10 - (9 \times 6) = 306$ experiment runs, for a total of more than 40 million HTTP calls.

In order to make a fair comparison between the three techniques, we manually executed the (hundreds of) generated test suites and measured the code coverage with IntelliJ IDEA,⁸ a Java IDE. For the faults found, we relied on the number of distinct API endpoints returning at least one 5XX status code, as done in [5], [7].

Tooling Setup: We performed the experiments with two open-source tools, RESTest and EvoMaster. Both tools require some manual setup. In the case of RESTest, we modified the test models of each SUT to include realistic test data generators for domain-specific parameters (e.g., YouTube video IDs for the YouTubeMock API). In the case of EvoMaster, we wrote the required Java classes used by the core component to start, stop and reset the SUT. In both cases, we used the default configuration of the tools.

C. Results

To evaluate the effectiveness of each testing technique, we computed the percentage of code coverage achieved (in terms of statement coverage of the whole application) and faults found (in terms of the number of endpoints returning 5XX status codes), out of 10 runs. In order to answer the research questions, we performed pair comparisons between the techniques evaluated, using the Vargha-Delaney effect size \hat{A}_{12} and the Mann-Whitney-Wilcoxon U-test (at $\alpha = 0.05$ significance level) [38].

1) *RQ1: Black-Box vs White-Box*: Table III shows the results achieved by black-box and white-box testing for the lower budget of HTTP calls (i.e., the only budget used for *BB*). “NaN” values obtained for the *p*-values mean that both techniques performed exactly in the same way in all cases (e.g., both *BB* and *WB* found one faulty endpoint in LanguageTool in the 10 experiment runs).

In terms of code coverage, *BB* outperformed *WB* with strong statistical evidence in 2 out of 4 SUTs. However, note that the high *p*-value for Ohsome is due to running the *BB* technique only once. The largest differences were observed in YouTubeMock and Ohsome (>50% code coverage), because these SUTs present very complex input constraints on and among their parameters. These constraints need to be satisfied in order to pass the input validation logic implemented in the system and exercise their actual functionality (e.g., computing the area of a city in Ohsome). In fact, none of the requests generated by EvoMaster for these SUTs obtained a successful response.

⁸<https://www.jetbrains.com/idea/>

TABLE IV
CONFIGURATIONS OF *HYBRID*.

SUT	Budget	Coverage (%)			5XX		
		Min	Med	Max	Min	Med	Max
REStcountries	5K	79.5	79.7	79.4	0.9	0.9	0.8
	100K	80.2	80.2	80.1	1.0	1.0	1.0
YouTubeMock	10K	68.9	81.4	86.3	0	0	0
	200K	74.2	84.5	87.7	0	0.2	0.5
Ohsome	30K	68.7	74.6	75.1	45.3	116.1	122.0
	600K	70.3	75.8	75.8	46.0	116.0	122.0
LanguageTool	100K	41.3	42.7	45.0	1.0	1.0	1.0
	1M	43.0	43.0	45.0	1.0	1.0	1.0

In the case of REStcountries and LanguageTool, these SUTs do not expose as complex constraints, and therefore a search-based approach can eventually infer those from the source code and generate high-coverage test cases.

In terms of fault finding, *BB* and *WB* performed more similarly. Both of them found one fault⁹ in LanguageTool and none in YouTubeMock, and each technique found unique bugs in one SUT (*WB* uncovered a faulty endpoint in REStcountries and *BB* found faults in all 122 endpoints of Ohsome). An endpoint may return a 500 status code due to multiple faults in the SUT, or vice versa, multiple endpoints might fail due to the same bug. Without a manual in-depth analysis, out of the scope of this paper, it is not possible to tell the exact number of faults found by each of the (hundreds of) test suites generated. However, an exploratory analysis in the Ohsome API seems to indicate that most of its failures are due to the same faults. All operations have nearly the same parameters, therefore all endpoints implement mostly the same validation logic, where the faults are located.

RQ1: With a low budget of HTTP calls, black-box constraint-based testing achieved between 1.3% and 55.9% higher code coverage than white-box testing. In terms of the faults found, both techniques performed similarly.

2) RQ2: Hybrid vs Black-Box & White-Box Approaches:

As previously explained, we evaluated *HYBRID* with three different seed sizes. Table IV shows the results obtained for each SUT, budget and the three seed sizes (min, med, and

⁹Recall that, by “fault” or “bug”, we actually refer to “an endpoint returning at least one 5XX status code”, as done by previous authors [5], [7].

max) described in Table II. The highest values on each row are highlighted in boldface. The best configuration was obtained with the largest seed, except for REStcountries, where the medium seed yielded the best results. This may be explained by the fact that the search algorithm may be overfitted when seeding many test cases, and instead it is necessary to explore the search space with more varied test cases.

We compared the best performing configuration of *HYBRID* against *BB* and *WB*, in pairs. Table V illustrates such comparison. Compared both to *BB* and *WB*, the improvements achieved in the code coverage by our novel hybrid approach are statistically significant with strong effect sizes, in all possible configurations (all SUTs and budgets) except for LanguageTool, where *BB* performed best. This difference may be caused by the peculiarities of this SUT. LanguageTool is composed of hundreds of Java classes consisting of “language rules”, whose code is only executed when the rule is triggered. A rule is triggered when a text provided as input contains a mistake matching the rule pattern. Since *BB* uses a random text generator, it may be more effective than using a search-based approach to evolve strings (which is not trivial [23], [39]) in ways that exercise new code. Overall, and in raw terms, *HYBRID* covered 2.2% more code than *BB* and 28.8% more code than *WB*.

Figure 3 shows the coverage achieved by *BB* (3a), *WB* (3b, 3d) and *HYBRID* (3c, 3e) at 5% intervals of the overall budget used. For *WB* and *HYBRID*, two charts are shown, one for the low budget (3b, 3c) and another for the high budget (3d, 3e). This figure reveals a significant difference in how the three techniques perform as the budget is progressively consumed. While *BB* needs a small portion of the budget to reach a coverage close to the maximum achieved, the coverage of *WB* and *HYBRID* evolves more steadily, because the search algorithm keeps generating test cases that cover new targets (e.g., source code statements and branches).

Regarding the faults found, we can say with high confidence (i.e., p -value < 0.05) that *HYBRID* performed better than *BB* in one scenario—REStcountries, low budget—and better than *WB* in three scenarios, namely, YouTubeMock with the high budget and Ohsome with both budgets. This result in the Ohsome SUT was expected, since the seeded test suite already contained fault-revealing test cases, which *WB* was not able to generate. It is noteworthy, however, that *HYBRID* uncovered a bug in YouTubeMock not uncovered by either

TABLE V
RQ2: COMPARISON OF *HYBRID* WITH *BB* AND *WB*.

SUT	Budget	Coverage (%)							5XX						
		HYBRID (h)	BB (b)	WB (w)	\bar{A}_{hb}	p-value	\bar{A}_{hw}	p-value	HYBRID (h)	BB (b)	WB (w)	\bar{A}_{hb}	p-value	\bar{A}_{hw}	p-value
REStcountries	5K	79.7	74.9	73.6	1.00	< 0.001	0.98	< 0.001	0.9	0.0	0.5	0.95	< 0.001	0.70	0.064
	100K	80.2	-	79.0	-	-	0.88	0.002	1.0	-	1.0	-	-	0.50	NaN
YouTubeMock	10K	86.3	81.9	26.0	1.00	< 0.001	1.00	< 0.001	0.0	0.0	0.0	0.50	NaN	0.50	NaN
	200K	87.7	-	47.3	-	-	1.00	< 0.001	0.5	-	0.0	-	-	0.75	0.014
Ohsome	30K	75.1	74.0	20.4	1.00	0.035	1.00	< 0.001	122.0	122.0	0.0	0.50	NaN	1.00	< 0.001
	600K	75.8	-	21.2	-	-	1.00	< 0.001	122.0	-	0.2	-	-	1.00	< 0.001
LanguageTool	100K	45.0	46.6	35.3	0.00	0.111	1.00	0.195	1.0	1.0	1.0	0.50	NaN	0.50	NaN
	1M	45.0	-	36.0	-	-	1.00	1.000	1.0	-	1.0	-	-	0.50	NaN

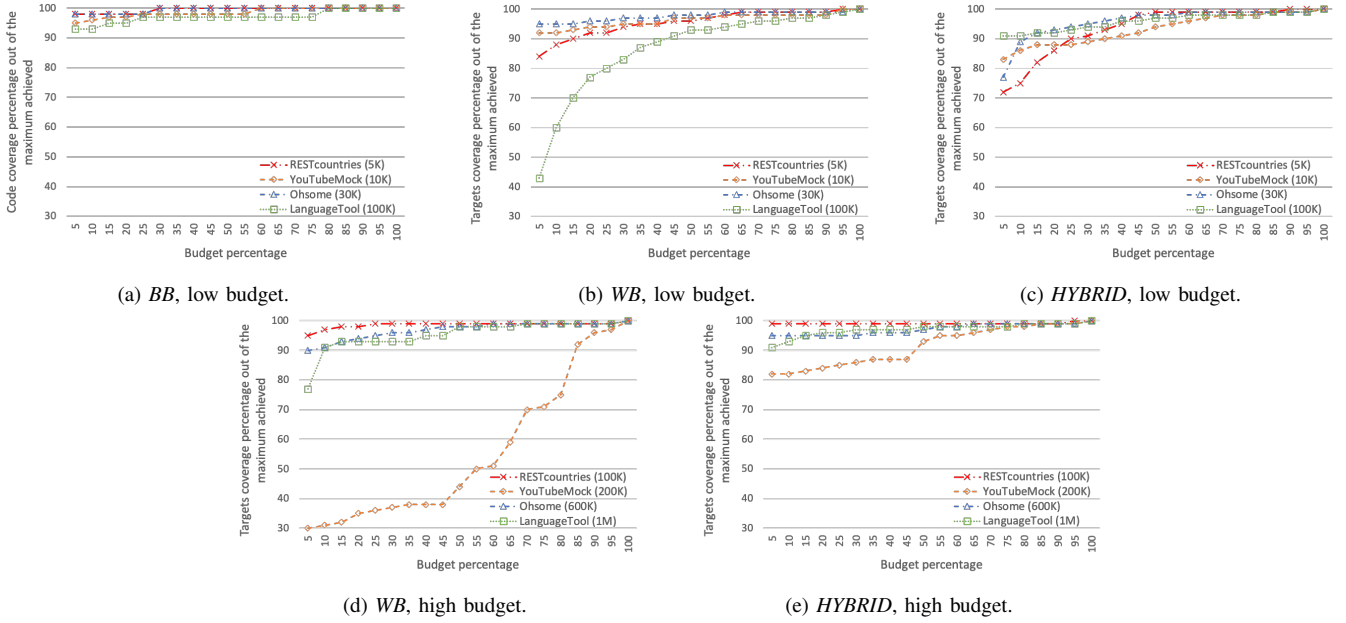


Fig. 3. Trend of coverage evolution for the *BB*, *WB* and *HYBRID* approaches. For *BB*, only the median test suite (i.e., the fifth best performing in terms of code coverage) is considered, since this had to be done manually. For *WB* and *HYBRID*, the average coverage of all test suites is computed, since EvoMaster can output these stats automatically.

BB nor *WB*. This highlights the potential of our proposal: seeding an evolutionary algorithm with a thorough test suite can provide a bootstrap for the search, and eventually make it find new faults. Even in the cases where no new bugs were uncovered, more code was covered in less time. For example, in RESTcountries, 5K HTTP calls sufficed to cover 79.7% code with *HYBRID*, while *WB* needed 100K HTTP calls to reach 79.0% coverage.

RQ2: Our novel hybrid approach covered between 1.2% and 60.3% more code than white-box testing in all SUTs, and between 1.1% and 4.8% more than black-box testing in 3 out of 4 SUTs. In terms of fault finding, the hybrid approach was the only one uncovering faults in all SUTs.

VI. LESSONS LEARNED

In this section, we provide lessons learned, derived from the empirical study. We may remark that these lessons are restricted to the scope of our practical experience and future research will be needed for generalizing them further.

Lesson 1: Black-box testing is significantly faster than white-box testing. Figure 3 shows that, for *BB*, ~5% of the budget suffices to reach ~95% of the code coverage obtained with the whole budget. In contrast, *WB* and *HYBRID* need more budget, but better results could be obtained. Moreover, since *WB* and *HYBRID* apply several heuristics for collecting metrics such as the statement coverage or the branch distance [40], this also translates into a time overhead. This was extreme in the case of LanguageTool, where a single *HYBRID*-1M-budget experiment run took over 10 hours to complete in a standard

PC.¹⁰ *BB* took less than 3 hours to generate a similar test suite. There might be some scenarios where time can be limited. For example, from a developer’s viewpoint, when developing a new feature for an enterprise application, this must be tested thoroughly before going into production. Taking into account the previous statement, that is, ~5% of the budget is enough to reach ~95% of the code coverage in black-box testing, it is worth considering the following question: what would developers prefer? Obtaining a reasonably thorough test suite in 10 minutes, or a more exhaustive test suite in 10 hours? Which of the two test suites would be *enough* to verify that a new feature did not break anything? Both approaches could be complementary, e.g., using *BB* as a quick check and integrating *HYBRID* as a part of a continuous integration (CI) setup.

Lesson 2: Hybrid testing is more effective than black-box and white-box testing in isolation. This is one of the main conclusions derived from the evaluation. In terms of code coverage, *HYBRID* achieved the best results in 3 out of 4 SUTs, increasing coverage by up to 4.8% and 60.3% compared to *BB* and *WB*, respectively. In terms of fault finding, *HYBRID* was the only technique uncovering bugs in all SUTs. Note that *HYBRID* could also be applied with already existing test suites, e.g., one created by the SUT developers. Even small test suites can mean a significant bootstrap for the search. For instance, test suites comprising just one test case per API endpoint in Ohsome (122 in total) and 10 test cases in YouTubeMock achieved +50% code coverage than when no seed was provided (Table V). Crafting such kind of test suites is a trivial task, especially thanks to current tool support, which

¹⁰Technical specifications: Intel i5 CPU, 16GB RAM, 256GB SSD, Windows 10, Java 8.

allows to do it manually (e.g., with the Swagger tool suite [41] or Postman [32]) or automatically (e.g., with RESTest [14]). This makes our novel hybrid approach readily applicable in practice when a test suite already exists, and easily applicable when it does not.

Lesson 3: Black-box constraint-based testing is generally preferred for large and complex systems. Based on our empirical study, *WB* is more effective than *BB* for simpler or smaller SUTs when given high budgets (e.g., RESTcountries). However, when the SUT contains intricate input constraints, the black-box approach generally outperforms the white-box, because the given custom domain knowledge in the data generators and the handling of inter-parameter dependencies [22] significantly helps in dealing with those constraints. A search algorithm may struggle to learn how to generate high-coverage inputs in short time, due to complex branch conditions implemented in the SUT code. This same limitation applies to *HYBRID* as well (since it also leverages the source code), and it became clear in LanguageTool. LanguageTool is the biggest SUT, with 10 times more LOCs than the other SUTs together. It is a CPU-bound application (e.g., no CRUD operations on a database), doing complex computations on string text inputs. In this case, some experiment runs of *WB* and *HYBRID* even failed to finish. The EvoMaster Java process ran out of memory due to the presence of very long regular expressions in the code. EvoMaster could not handle those (e.g., with more than 200K characters) and crashed. In other cases, EvoMaster simply timed out due to very long execution times (e.g., more than 24 hours). All these results emphasize the following: (1) white-box testing is subject to technical limitations, depending on the SUT; and (2) black-box testing is applicable regardless of how the SUT is implemented, its size or its complexity.

Lesson 4: A manual setup is required (and recommended) in all the studied approaches. In RESTful API testing, as in many other disciplines, there is a trade-off between automation and test thoroughness. Approaches achieving the highest automation degree apply fuzz or random testing [5], [11] or leverage techniques that are simply not applicable in all scenarios [10], [12]. The three techniques evaluated in this work require some manual setup. Without a proper empirical investigation, it is hard to tell which one requires less work, but based on our experience, EvoMaster is quicker to set up than RESTest. In fact, EvoMaster’s configuration requires writing one Java class, while RESTest requires modifying a YAML file whose size depends on the number of API endpoints and parameters. However, as suggested by Table V, the manual setup done for RESTest may play a key role in the final results obtained, especially for complex APIs dealing with very domain-specific parameters. Are developers willing to do some manual work to improve the testing process? Or do they prefer 100% automation? This is something to be empirically investigated in future work.

Lesson 5: White-box test suites are deterministic, black-

box test suites are usually not. One of the main differences between white-box and black-box testing is that, in the former, the state of the SUT can be reset between test cases (e.g., emptying a database). Therefore, it is possible to create a completely deterministic test suite, where the output of every test case does not depend at all on the test cases previously executed. This is not possible in black-box testing for stateful APIs, where, for instance, the response to the 100th API request depends entirely on the previous 99 requests executed. This complicates debugging, since a fault-revealing test case may not reveal the same fault again, if the state of the SUT has changed. White-box and hybrid testing can be used to generate regression test suites, whereas black-box testing is more typically applied for fuzzing web services for given periods of time [11].

VII. OPEN CHALLENGES

In the next paragraphs, we discuss open challenges to be tackled by future research in RESTful API testing. These are aligned with two primary goals: increasing test automation and thoroughness.

Challenge 1: Higher degree of automation. As previously mentioned, the techniques evaluated in this paper require some manual setup. This is something that may prevent practitioners from adopting them. For example, EvoMaster requires the tester to write a driver class defining how to start, stop and reset the SUT, among others. Automatic code generation techniques could be applied to ease this task. On the other hand, black-box testing techniques in general may require the addition of missing information in the API specification, such as inter-parameter dependencies. Advanced techniques that can automatically extract this information are required, e.g., by analyzing request-response patterns and inferring invariants present in the API, as hinted in the work by Mirabella et al. [42].

Challenge 2: Inference of realistic test inputs. Very related to the previous challenge, testing RESTful APIs thoroughly depends to a great extent on the inputs (HTTP requests) used. Creating and maintaining data dictionaries for each parameter is very costly [9], [14]. Based on the API specification, which usually describes the available endpoints and parameters, there is need for techniques that exploit it to infer realistic values for each API parameter, e.g., with natural language processing (NLP) capabilities. The work of previous authors in GUI testing [43], [44] may inspire future research to address this challenge.

Challenge 3: Advanced white-box heuristics. The effectiveness of white-box testing, or even its applicability, highly depends on the heuristics used to leverage the source code. If these are not correctly implemented, they may make white-box testing unusable (e.g., like the crashes that we obtained in EvoMaster with LanguageTool). If the heuristics are too simple, white-box testing may not be effective. In EvoMaster, taint checking and testability transformations, among others,

are applied to generate higher coverage inputs [23]. Nonetheless, more techniques could be applied to improve the search process, such as dynamic symbolic execution [45] and the seeding strategies discussed in [31]. Furthermore, these heuristics are language-dependent, therefore more engineering effort is required to extend them to other programming languages such as JavaScript and C#, heavily used in industry for building RESTful APIs [46].

Challenge 4: Test oracles. In automated software testing, generating test inputs is only half of the challenge. Asserting test outputs is equally important. In RESTful API testing, most typical oracles include checking the presence of 5XX status codes, and mismatches with the API specification [7]–[12]. Previous authors have proposed more specific oracles related to checking security [28] or functional properties [6], [8] of the SUT. More work is needed to catch more and more varied types of bugs.

VIII. THREATS TO VALIDITY

Next we discuss the validity threats that apply to our work.

Internal validity. Threats to the internal validity relate to those factors that may affect the results of our evaluation. The experiments were performed with two tools, RESTest and EvoMaster, and we extended both of them to support a third testing approach. Faults in such tools might compromise the validity of our conclusions. Even though both tools are thoroughly tested, it cannot be guaranteed that they are free of bugs. To mitigate this threat, and to enable replicability of our results, we intentionally used open-source tools, whose code is freely available on GitHub [47], [48]. Furthermore, the source code of the extensions of both tools (developed by the authors), the case studies and the results obtained (Java test classes, test reports, etc.) are provided as a part of the supplementary material of this paper [49].

Given that the testing techniques evaluated in this paper are based on randomized algorithms, such randomness might affect the results. To mitigate this problem, each experiment was repeated 10 times with different random seeds, and appropriate statistical tests were used to analyze the results.

External validity. Threats to the external validity might affect the generalizability of our findings. One of the main threats in this regard comes from the fact that only four RESTful web APIs were evaluated in the empirical study. This is because running experiments on system-level test case generation is very time-consuming. For example, EvoMaster’s experiments required more than 1,500 days of computational resources in a cluster of computers. Then, we had to manually run all test suites generated to measure the code coverage in IntelliJ, in order to make a fair comparison between the three approaches. Our results might not completely generalize to other web APIs (e.g., stateful APIs), however, to mitigate this threat, we selected non-trivial APIs of different sizes (between 1 and 122 endpoints, and between 1.5K and 162K LOCs), from different application domains and with varied degrees of complexity.

Besides the three testing techniques compared in this paper, others could have been evaluated as well. Regarding black-box testing, we are only aware of one more open-source tool, RESTler [11]. For white-box testing, EvoMaster seems to be the only open-source tool supporting system-level testing of RESTful APIs, although other tools exist for unit testing (e.g., EvoSuite [29]). The comparison with these other alternatives would have yielded interesting insights, but it would have required a much larger evaluation, which is out of the scope of this paper. Furthermore, we specifically selected RESTest and EvoMaster as the tools to compare and integrate because they are good representatives of black-box and white-box testing, respectively. RESTest is a sophisticated constraint-based testing tool that can exploit additional information provided by the tester (while RESTler cannot), and EvoMaster generates test cases in a fully automated fashion by leveraging evolutionary algorithms.

IX. CONCLUSION

In this practical experience report, we presented the first comparison of different black-box and white-box testing techniques for RESTful web APIs, and we proposed a novel hybrid testing approach that combines both strategies. We performed an extensive evaluation involving four case studies with very different characteristics. The results obtained bring to light two main conclusions: (1) our novel hybrid technique performed best in most scenarios, achieving the highest code coverage in 3 out of 4 SUTs and being the only technique capable of uncovering bugs in all of them; and (2) there exists no *one-size-fits-all* strategy, as they all have their own strengths and limitations. After an in-depth analysis of the results, we provide lessons learned that can hopefully provide a better understanding of the applicability of black-box, white-box and hybrid testing, according to multiple factors such as the SUT complexity or the available budget.

In future work, we plan to perform a more comprehensive evaluation including more case studies of different variety, as well as other testing techniques such as fuzzing [11], which could also be used to generate the seed in our hybrid approach. We also intend to develop more advanced heuristics to make fully-automated white-box search-based testing as effective as black-box constraint-based testing with custom data generators. In theory, given enough budget, the former should outperform the latter. However, based on the results from Table III, we are not there yet, and more work is needed.

ACKNOWLEDGMENT

This work has been partially supported by FEDER/Ministerio de Ciencia e Innovación – Agencia Estatal de Investigación under project HORATIO (RTI2018101204-B-C21), by FEDER/Junta de Andalucía under projects APOLO (US-1264651) and EKIPMENT-PLUS (P18-FR-2895), and by the FPU scholarship program, granted by the Spanish Ministry of Education and Vocational Training (FPU17/04077). This work is supported by the Research Council of Norway (project on Evolutionary Enterprise Testing, grant agreement No 274385).

REFERENCES

- [1] D. Jacobson, G. Brail, and D. Woods, *APIs: A Strategy Guide*. O'Reilly Media, Inc., 2011.
- [2] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*. O'Reilly Media, Inc., 2013.
- [3] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, 2000.
- [4] "ProgrammableWeb API Directory," accessed January 2021. [Online]. Available: <http://www.programmableweb.com/>
- [5] A. Arcuri, "Automated Blackbox and Whitebox Testing of RESTful APIs With EvoMaster," *IEEE Software*, 2021.
- [6] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic Testing of RESTful Web APIs," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2018.
- [7] A. Arcuri, "RESTful API Automated Test Case Generation with EvoMaster," *ACM TOSEM*, vol. 28, no. 1, pp. 1–37, 2019.
- [8] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "REStest: Black-Box Constraint-Based Testing of RESTful Web APIs," in *International Conference on Service-Oriented Computing*, 2020, pp. 459–475.
- [9] S. Karlsson, A. Causevic, and D. Sundmark, "QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs," in *International Conference on Software Testing, Verification and Validation*, 2020.
- [10] E. Vigliani, M. Dallago, and M. Ceccato, "RestTestGen: Automated Black-Box Testing of RESTful APIs," in *International Conference on Software Testing, Verification and Validation*, 2020.
- [11] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *International Conference on Software Engineering*, 2019, pp. 748–758.
- [12] H. Ed-douibi, J. L. C. Izquierdo, and J. Cabot, "Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach," in *International Enterprise Distributed Object Computing Conference*, 2018, pp. 181–190.
- [13] "OpenAPI Specification," accessed January 2021. [Online]. Available: <https://www.openapis.org>
- [14] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "REStest: Automated Black-Box Testing of RESTful Web APIs," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '21, 2021.
- [15] "Google Drive API," accessed February 2021. [Online]. Available: <https://developers.google.com/drive/api/v3/reference/>
- [16] "Flickr API," accessed February 2021. [Online]. Available: <https://www.flickr.com/services/api/>
- [17] "Twitter API," accessed January 2021. [Online]. Available: <https://developer.twitter.com/en/docs>
- [18] "Swagger Codegen," accessed October 2020. [Online]. Available: <https://swagger.io/tools/swagger-codegen/>
- [19] S. Bucaille, J. L. Cánovas Izquierdo, H. Ed-Douibi, and J. Cabot, "An openapi-based testing framework to monitor non-functional properties of rest apis," in *International Conference on Web Engineering*, 2020, pp. 533–537.
- [20] "LanguageTool API," accessed October 2020. [Online]. Available: <https://languagetool.org/proofreading-api>
- [21] A. Martin-Lopez, S. Segura, C. Müller, and A. Ruiz-Cortés, "Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs," *IEEE Transactions on Services Computing*, 2021, article in press.
- [22] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "A Catalogue of Inter-Parameter Dependencies in RESTful Web APIs," in *International Conference on Service-Oriented Computing*, 2019, pp. 399–414.
- [23] A. Arcuri and J. P. Galeotti, "Testability Transformations For Existing APIs," in *International Conference on Software Testing, Verification and Validation*, 2020, pp. 153–163.
- [24] —, "Handling SQL Databases in Automated System Test Generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–31, 2020.
- [25] I. Schieferdecker, "Model-Based Testing," *IEEE software*, vol. 29, no. 1, pp. 14–18, 2012.
- [26] A. Arcuri, "Test Suite Generation with the Many Independent Objective (MIO) Algorithm," *Information and Software Technology*, vol. 104, pp. 195–206, 2018.
- [27] P. Godefroid, B.-Y. Huang, and M. Polishchuk, "Intelligent REST API Data Fuzzing," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, p. 725–736.
- [28] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Checking Security Properties of Cloud Services REST APIs," in *International Conference on Software Testing, Verification and Validation*, 2020.
- [29] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.
- [30] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, 2007, pp. 815–816.
- [31] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding Strategies in Search-Based Unit Test Generation," *Software Testing, Verification and Reliability*, vol. 26, no. 5, pp. 366–401, 2016.
- [32] "Postman," accessed January 2021. [Online]. Available: <https://www.getpostman.com>
- [33] "JUnit 5," accessed January 2021. [Online]. Available: <http://junit.org/junit5/>
- [34] "REST Assured," accessed January 2021. [Online]. Available: <http://rest-assured.io>
- [35] "YouTube Data API," accessed February 2021. [Online]. Available: <https://developers.google.com/youtube/v3/>
- [36] "OpenStreetMap," accessed February 2021. [Online]. Available: <https://www.openstreetmap.org/>
- [37] A. Gamez-Diaz, P. Fernandez, A. Ruiz-Cortés, P. J. Molina, N. Kolekar, P. Bhogill, M. Mohaan, and F. Méndez, "The Role of Limitations and SLAs in the API Industry," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1006–1014.
- [38] A. Arcuri and L. Briand, "A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.
- [39] P. McMin, M. Shahbaz, and M. Stevenson, "Search-Based Test Input Generation for String Data Types Using the Results of Web Queries," in *International Conference on Software Testing, Verification and Validation*, 2012, pp. 141–150.
- [40] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.
- [41] "OpenAPI Design & Documentation Tools — Swagger," accessed February 2021. [Online]. Available: <https://swagger.io/tools/>
- [42] A. G. Mirabella, A. Martin-Lopez, S. Segura, L. Valencia-Cabrera, and A. Ruiz-Cortés, "Deep Learning-Based Prediction of Test Input Validity for RESTful APIs," in *International Workshop on Testing for Deep Learning and Deep Learning for Testing*, 2021.
- [43] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Link: Exploiting the Web of Data to Generate Test Inputs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 373–384.
- [44] T. Wanwarang, N. P. Borges, L. Bettscheider, and A. Zeller, "Testing Apps With Real-World Inputs," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, 2020, p. 1–10.
- [45] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
- [46] A. Arcuri, "An Experience Report on Applying Software Testing Academic Results in Industry: We Need Usable Automated Test Generation," *Empirical Software Engineering*, vol. 23, no. 4, pp. 1959–1981, 2018.
- [47] "REStest: Automated Black-Box Testing of RESTful Web APIs," accessed January 2021. [Online]. Available: <https://github.com/isa-group/REStest>
- [48] "EvoMaster: A Tool for Automatically Generating System-Level Test Cases," accessed January 2021. [Online]. Available: <https://github.com/EMResearch/EvoMaster>
- [49] A. Martin-Lopez, A. Arcuri, S. Segura, and A. Ruiz-Cortés, "Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies? - Supplementary package [Data set]." [Online]. Available: <https://doi.org/10.5281/zenodo.4563132>