RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs

Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés

Smart Computer Systems Research and Engineering Lab (SCORE), Research Institute of Computer Engineering (I3US)
Universidad de Sevilla, Spain
{alberto.martin,sergiosegura,aruiz}@us.es

Abstract. Automated testing approaches for RESTful web APIs typically follow a black-box strategy, where test cases are derived from the API specification. These techniques show promising results, but they neglect constraints among input parameters (so-called *inter-parameter dependencies*), as these cannot be formally described in current API specification languages. As a result, black-box tools rely on brute force to generate valid test cases, i.e., those satisfying all the input constraints. This is not only extremely inefficient, but it is also unlikely to work for most real-world services, where inter-parameter dependencies are complex and pervasive. In this paper, we present RESTest, a framework for automated black-box testing of RESTful APIs. Among its key features, RESTest supports the specification and automated analysis of inter-parameter dependencies, enabling the use of constraint solvers for the automated generation of valid test cases. This allows to detect more faults, and faster, through a deeper evaluation of valid and invalid input parameters' combinations and the use of novel test oracles. Evaluation results on 6 commercial APIs show that RESTest can efficiently generate up to 99% more valid test cases than random testing techniques, 60% on average. More importantly, RESTest revealed 2K failures undetected by random testing, uncovering bugs in all the services under test.

Keywords: REST · Black-box testing · Constraint-based testing · Web services

1 Introduction

Web APIs allow systems to interact over the network, typically using web services [18]. Modern web APIs typically adhere to the REpresentational State Transfer (REST) architectural style [5], being referred to as RESTful web APIs. RESTful web APIs are comprised of one or more RESTful web services, each of which implements one or more create, read, update, or delete (CRUD) operations to access and manipulate a resource, e.g., a video in the YouTube API. RESTful APIs are commonly described using languages like the OpenAPI Specification (OAS) [15], originally created as a part of the Swagger tool suite [22]. OAS is designed to provide a structured description of a RESTful web API that allows

both humans and computers to discover and understand the capabilities of a service without requiring access to the source code or additional documentation.

Web APIs often impose dependency constraints that restrict the way in which two or more input parameters can be combined to form valid calls to the service, these are often called *inter-parameter dependencies* (or simply dependencies henceforth). For example, in the Google Maps API, when searching for places, if the `location` parameter is set, then the `radius` parameter must be set too, otherwise a 400 status code ("bad request") is returned. In a recent study, we reviewed more than 2.5K operations from 40 industrial APIs and found that dependencies are extremely common and pervasive—they appear in 4 out of every 5 APIs across all application domains and types of operations [11]. Unfortunately, current API specification languages like OAS provide no support for the formal description of this type of dependencies, despite being a highly demanded feature by practitioners[1]. Instead, users are encouraged to describe dependencies among input parameters informally, using natural language, which leads to ambiguities and makes it hardly possible to interact with services without human intervention[2]. To address this problem, in previous work we proposed a domain-specific language for the formal specification of dependencies, called Inter-parameter Dependency Language (IDL), and a tool suite for the automated analysis of IDL using constraint programming [12] (c.f. Section 2). In this paper, we show the potential of IDL and its tool suite in the context of testing RESTful APIs.

The validation of RESTful web APIs is critical as they play a key role in modern software integration. A faulty API can have a huge impact in the many applications using it. The automated detection of bugs in RESTful web APIs is an active research topic [2–4, 9, 19, 23]. Most contributions in this context follow a black-box strategy, where the specification of the API under test (described using the OAS language) is used to drive the generation of test cases [3, 4, 9, 23]. Essentially, these approaches exercise the API under test using (pseudo) random test data. Test data generation strategies include using default values [4], input data dictionaries [3], test data generators [9] and data observed in previous calls to the API [23]. Failures are detected when the observed output deviates from the specification, e.g., unexpected HTTP status codes.

Problem: Current black-box testing approaches for RESTful web APIs do not support inter-parameter dependencies since, as previously mentioned, these are not formally described in the API specification used as input. As a result, existing approaches simply ignore dependencies and resort to brute force to generate valid test cases, i.e., those satisfying all input constraints. This is not only extremely inefficient, but it is also unlikely to work for most real-world services, where inter-parameter dependencies are complex and pervasive. For example, the search operation in the YouTube API has 31 input parameters, out of which 25 are involved in at least one dependency: trying to generate valid

[1] This is reflected in an open feature request in OAS entitled *"Support interdependencies between query parameters"*, with over 290 votes and 55 comments from 33 participants. `https://github.com/OAI/OpenAPI-Specification/issues/256`

[2] `https://swagger.io/docs/specification/describing-parameters/`

test cases randomly is like hitting a wall. This was confirmed in our evaluation, where 98 out of every 100 random test cases for the YouTube search operation violated one or more inter-parameter dependencies (c.f. Section 4.3).

Contribution: In this paper, we present RESTest, an open-source and black-box automated testing framework for RESTful web APIs. RESTest follows a model-based approach enabling its integration with different test case generators and testing frameworks. As its most distinctive feature, RESTest supports the specification and automated analysis of inter-parameter dependencies using the IDL tool suite. This allows to exploit constraint solving as a part of the test generation process, a testing technique generally known as *constraint-based testing* [8]. Constraint-based testing enables a better coverage of the program under test through the systematic generation of valid and invalid input combinations, as well as the use of novel output assertions, i.e., test oracles. For the evaluation of RESTest, we tested 9 operations from 6 commercial APIs, including Tumblr, GitHub and YouTube. Specifically, we compared random testing—state-of-the-art technique for black-box testing of RESTful APIs—and constraint-based testing. As expected, random testing struggled to generate valid test cases: 60% of the generated test cases violated inter-parameter dependencies (about 99% in the APIs of Stripe and YouTube). In contrast, constraint-based testing generated 100% valid test cases for all the services under test, keeping the test case generation time in milliseconds. More importantly, constraint-based testing detected more failures than random testing (4K vs 3K), in more services (9 vs 5), showing the potential of RESTest in practice.

This work includes the following original contributions in the context of automated testing of RESTful web APIs:

1. An open-source and model-based framework for automated black-box test case generation and execution.
2. A new constraint-based approach for improving test case generation techniques, including two novel automated test oracles.
3. Experimental evidence on the limits of using random testing in real-world services with inter-parameter dependencies.
4. A comparison of random testing and constraint-based testing on 6 commercial APIs, showing the potential of both techniques, and especially constraint-based testing, to uncover real bugs.

The remainder of the paper is organised as follows: Section 2 introduces the IDL tool suite, used for the automated analysis of inter-parameter dependencies in RESTful web APIs. Section 3 presents RESTest, our testing framework for RESTful APIs. Section 4 explains the evaluation performed and the results obtained. Section 5 outlines threats to validity. Section 6 describes related work. Finally, Section 7 draws conclusions and discusses future lines of research.

2 IDL Tool Suite

RESTest relies on the IDL tool suite for the automated management of inter-parameter dependencies in RESTful APIs [12]. *Inter-parameter Dependency Lan-*

guage (IDL) [21] is a domain-specific language for the specification of inter-parameter dependencies in web APIs. It is based on a thorough study of more than 2.5K operations in 40 real-world APIs [11]. Specifically, it provides support for eight different types of dependencies among input parameters consistently found in practice. Listing 1 shows an example of each type of dependency taken from commercial APIs. The syntax is self-explanatory. For example, the *Requires* dependency in line 1, observed in the API of YouTube, states that, when using the parameter `videoDefinition`, the parameter `type` must be set to `'video'`. IDL specifications can be integrated into OAS documents using the IDL4OAS extension [12]. This allows to enrich API specifications with an accurate, not ambiguous and machine-readable description of the dependencies among input parameters. We refer the reader to the supplementary material of the paper for examples of API specifications using the OAS language and the IDL4OAS extension [21].

```
1 IF videoDefinition THEN type=='video';                          // Requires
2 Or(query, type);                                                // Or
3 ZeroOrOne(radius, rankby=='distance');                          // ZeroOrOne
4 AllOrNone(location, radius);                                    // AllOrNone
5 OnlyOne(amount_off, percent_off);                               // OnlyOne
6 publishedAfter >= publishedBefore;                              // Relational
7 limit + offset <= 1000;                                         // Arithmetic
8 IF intent=='browse' THEN OnlyOne(ll AND radius, sw AND ne);     // Complex
```

Listing 1: Examples of IDL dependencies from real-world APIs.

IDLReasoner [21] is an open-source Java library for the automated analysis of IDL specifications [12]. Given an OAS specification and a set of IDL dependencies (e.g., using IDL4OAS), the tool translates them into a constraint satisfaction problem (CSP) expressed in MiniZinc [14], a constraint solving language designed for modelling optimisation problems in a high-level, solver-independent way. Then, several analysis operations can be invoked on the resulting CSP, for instance, to know whether a given API request satisfies all the inter-parameter dependencies. Section 3.2 describes the analysis operations used to support test case generation in RESTest.

3 RESTest

In this section, we present RESTest, our framework for automated black-box testing of RESTful web APIs. RESTest follows a model-based approach, where test cases are automatically derived from the specification of the API under test. No access to the source code is required, which makes it possible to test APIs written in any programming language, running in local or remote servers.

Figure 1 shows how RESTest works. It takes as input the OAS specification of the API under test, considered the *system model*. The specification can optionally describe inter-parameter dependencies using the IDL4OAS extension. Then, a so-called *test model* is automatically generated from the system model including test-specific configuration data. The default test model can be manually enriched with fine-grained configuration details such as test data generation

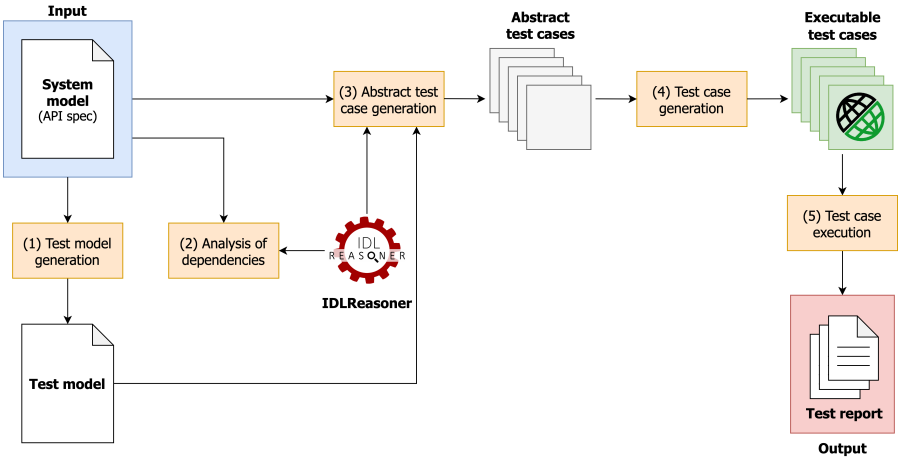

Fig. 1: Test case generation and execution in RESTest.

settings. Then, both the system and the test models are leveraged for the generation of abstract test cases following user-defined test case generation strategies such as random testing. In parallel, inter-parameter dependencies, if any, are fed into the tool IDLReasoner, providing support for their automated analysis during test case generation, for instance, to check whether an API call satisfies all the inter-parameter dependencies defined in the specification. Finally, abstract test cases are transformed into platform-specific executable test cases and they are executed. In the following sections, we detail the main steps of the process.

3.1 Default Test Model Generation

RESTest takes as input the specification of the API under test, i.e., *system model*. Specifications described using the OAS language—arguably considered the industry-standard and used in related approaches [2–4, 9, 23]—are supported, but other API specification languages could be integrated into the framework using available converters, e.g., RAML to OAS [15]. As a distinctive feature, RESTest supports the specification of inter-parameter dependencies within the OAS document using the IDL4OAS extension [12] (c.f. Section 2).

Test case generation in RESTest is driven by the system and the test models. The *test model* includes all test-related configuration settings for the API under test. A default test model, formatted in YAML (i.e., same language used in OAS), is automatically generated from the input API specification (system model). Such test model might be enough to generate effective test cases in some APIs. However, in practice, some manual tuning is often necessary, for example, to generate input values hardly inferred from the specification such as identifiers or codes. In particular, RESTest supports the following configuration settings:

– *Operations under test.* It is possible to specify the subset of the API operations to be tested. Specific test configuration settings can be defined for each operation under test.

- *Authentication data*. This includes the API keys or tokens required to call secured APIs.
- *Test data generators*. This allows to customise the data values used for each input parameter. Test data generators in RESTest include random value generators, regular expression generators, boundary-value generators, and data dictionaries, among others. Default generators are configured according to the type of the input parameters, e.g., English words for string parameters.
- *Weights*. Testers might be interested in testing some parameters more thoroughly than others, for example, those more used in practice. Weights allow to do so. A weight is a real number in the range [0,1]. The higher the weight of a parameter, the more frequently it will be used in test cases. By default, all parameters have a weight of 0.5.

3.2 Automated Analysis of Inter-Parameter Dependencies

API specifications including inter-parameter dependencies in IDL are provided as input to IDLReasoner [12]. This tool transforms the specification into a CSP and automatically checks for inconsistencies in the specification, informing the user about any errors, e.g., parameters that cannot be selected. Once the specification is validated, IDLReasoner provides test case generators with a catalogue of helpful analysis operations. Among these, three analysis operations stand out as particularly helpful during test case generation, namely:

- *isValidRequest*. This operation takes as input an API specification (including inter-parameter dependencies) and a service request (i.e., a list of parameters and their values), and returns a Boolean indicating whether the request is valid or not. A service request is valid if it satisfies all the inter-parameter dependencies defined in the specification.
- *getRandomValidRequest*. This operation receives the API specification of an API operation, and returns a random valid request for the operation, that is, a random assignment of values to input parameters satisfying all the dependencies of the specification.
- *getRandomInvalidRequest*. Contrary to the previous operation, this operation returns a random request violating one or more dependencies.

The use of the IDL tool suite allows to decouple the automated management of dependencies from the specific test case generation approach used. This makes RESTest highly generic and easy to maintain. Furthermore, it eases the use of different CSP solvers and the development of new analysis operations.

3.3 Abstract Test Case Generation

Test cases can be derived from the system and test models using one or more test case generation techniques. These test cases are *abstract* or platform-independent, meaning that they can be later transformed into executable test cases for specific testing frameworks and programming languages. RESTest currently supports random and constraint-based test case generation, but other techniques

(e.g., search-based generation) could be easily integrated extending the right interfaces. Abstract test cases comprise test inputs, expected outputs (test oracles), and the required information to build the API request, e.g., the endpoint. RESTest currently supports testing at the operation-level, that is, each test case performs a single API request.

RESTest generates both nominal and faulty test cases. *Nominal test cases* aim to test the API with valid inputs, i.e., those conforming to the API specification. In practice, it is not always possible to guarantee that a nominal test case represents a valid call to the API since it could violate some inter-parameter dependency, for example. Therefore, nominal test cases can be regarded as *potentially valid* test cases aimed at obtaining successful responses from the API (i.e., 2XX status codes). *Faulty test cases* check the ability of the API to handle invalid inputs, and therefore they expect a client error as a response (i.e., 4XX status codes). Faulty test cases are generated by creating faulty variants (i.e., mutants) of nominal test cases. For example, RESTest supports the automated generation of faulty test cases by excluding mandatory parameters, using out-of-range values (e.g., assigning a string to an integer parameter), and violating the JSON schema of the request body, among others. Additionally, as a novel feature of RESTest, the framework supports the automated generation of invalid requests violating inter-parameter dependencies using IDLReasoner.

Test case generation techniques mostly focus on generating test inputs, however, half of the challenge in testing lies on test oracles, that is, how to distinguish correct outputs from incorrect ones. RESTest supports the five test oracles described below, where the last two are novel as they rely on the automated analysis of inter-parameter dependencies.

- *5XX*. The status code must be lower than 500 (server error).
- *OAS*. The response must conform to the OAS schema.
- $2XX_P$. If the request violates the specification of individual parameters (e.g., a mandatory parameter is missing), the status code must not be 2XX (successful response).
- $2XX_D$. If the request violates one or more inter-parameter dependencies, the status code must not be 2XX.
- *4XX*. If the request is valid according to the API specification, the status code must not be 4XX (client error response).

Oracles $2XX_D$ and *4XX* are novel contributions of our work. Both of them reveal failures undetectable by current state-of-the-art test oracles. It is worth noting that oracle *4XX* is particularly helpful as it allows to detect critical bugs: those making the API return a client error response (4XX status code) with a valid API call. Detecting this kind of failures is only possible thanks to the automated analysis of inter-parameter dependencies, which allows to automatically determine whether a request is valid before calling the actual API (assuming that the specification is correct and that the right test data generators are used).

3.4 Test Case Generation and Execution

The last step is concerned with test execution. Abstract test cases are *instantiated* into executable test cases using specific testing frameworks and libraries. RESTest currently supports the generation of executable test cases using REST Assured [17], a Java library for testing RESTful services, developed as a JUnit extension. However, other frameworks and programming languages could be easily supported by implementing specific test writers.

Test execution can be done either offline or online. In *offline testing*, test case generation and execution are independent tasks. This has certain benefits. For example, test cases can be generated once, and then be executed many times as a part of regression testing. Also, test generation and test execution can be performed on different machines and at different times. In *online testing*, test case generation and execution are interleaved. This enables, for example, fully autonomous testing of RESTful web APIs, e.g., generating and executing test cases 24/7 as a part of a Continuous Integration (CI) setup. RESTest supports both offline and online testing. However, more sophisticated techniques for online testing remain to be implemented. For example, the test generation algorithms can react to the actual outputs of the API under test, e.g., to guide search-based test case generation algorithms based on the coverage achieved so far [13].

4 Evaluation

In this section, we assess the ability of RESTest to generate valid test cases (i.e., those satisfying all the input constraints) and to reveal failures in real-world APIs with inter-parameter dependencies. To this end, we compare random testing (RT)—state-of-the-art technique for black-box testing of RESTful APIs—and constraint-based testing (CBT). We address the following research questions:

- **RQ1**: *What is the effectiveness of CBT in generating valid test cases for real-world APIs with inter-parameter dependencies?*
- **RQ2**: *What is the fault-finding capability of CBT in real-world APIs with inter-parameter dependencies?*

4.1 Services Under Test

We tested 9 services from 6 commercial RESTful APIs with millions of users. We selected both read and write operations from those services. In order to assess the potential of CBT, we selected operations containing the eight types of dependencies identified in our study [11], with more than 50% of their parameters involved in at least one dependency. Table 1 provides a summary of the services under test (SUTs). For each SUT, the table shows an identifier (used to refer to it within the rest of the paper), API name, description of the operation tested, number of input parameters (P), number of IDL dependencies (D), and number (and percentage) of different parameters involved in at least one dependency

ID	API	Operation	P	D	PD (%)
Foursquare	Foursquare	Search venues	17	8	10 (59%)
GitHub	GitHub	Get user repositories	5	2	3 (60%)
Stripe-CC	Stripe	Create coupon	9	3	5 (56%)
Stripe-CP	Stripe	Create product	18	6	11 (61%)
Tumblr	Tumblr	Get blog likes	5	1	3 (60%)
Yelp	Yelp	Search businesses	14	3	7 (50%)
YouTube-GCT	YouTube	Get comment threads	11	5	8 (73%)
YouTube-GV	YouTube	Get videos	12	5	7 (58%)
YouTube-S	YouTube	Search	31	16	25 (81%)
Mean			13.6	5.4	8.8 (62%)

Table 1: RESTful API operations used in the evaluation. P = Number of parameters, D = Number of IDL dependencies, PD = Number and percentage of parameters involved in at least one dependency.

(PD). On average, the operations have 14 parameters, 5 dependencies and 9 parameters involved in dependencies.

The OAS specification of each API under test, used as input in RESTest, was taken from the API website or from the APIs.guru repository [1]. When the specification was not available (Foursquare, Tumblr and Yelp), we created it manually based on the online API documentation. Then, we looked for inter-parameter dependencies described in the documentation and included them as a part of the specification using the IDL4OAS extension. The links to the APIs under test and their OAS specifications are available as part of the supplementary material of the paper [21].

4.2 Test Case Generation Techniques

Next, we describe the test case generation techniques used in the evaluation.

Random testing (RT). This is the state-of-the-art approach used as baseline in our work [4, 9, 23]. Nominal test cases are generated by randomly selecting a subset of the operation parameters and assigning random values to them within their domain. All parameters are selected with the same probability (i.e., weight = 0.5) except mandatory ones, which are always included in the API request. Notice that this approach neglects inter-parameter dependencies and so the generated test cases may not be valid, i.e., they may generate responses with 4XX status codes (client error). Faulty test cases are generated by mutating nominal test cases as described in Section 3.3, e.g., excluding a mandatory parameter from the API call.

Constraint-based testing (CBT). Nominal test cases are generated in two steps. First, the domain of each input parameter is discretised and reduced to a fixed number of random values, within their domain, using RESTest test data generators. Then, the analysis operation *getRandomValidRequest* is invoked on IDLReasoner to generate a request that satisfies all inter-parameter dependencies. Analogously to RT, faulty test cases can be generated by mutating nominal

test cases. Additionally, faulty test cases can also be generated by invoking the *getRandomInvalidRequest* operation on IDLReasoner to generate an API call violating one or more inter-parameter dependencies.

4.3 Experiment 1: Generation of Valid Test Cases

In this experiment, we aim to answer RQ1 by evaluating the effectiveness of RT and CBT in generating valid test cases, i.e., those satisfying all inter-parameter dependencies. The automated generation of valid test cases has two main benefits. First, these are very helpful during regression testing as a part of Continuous Integration. Second, and more importantly, valid test cases help identify critical bugs: those returning an error (i.e., 4XX or 5XX status code) with an input that should be successfully handled by the service (i.e., 2XX status code). In what follows, we describe the setup and the results of the experiment.

Setup. For each SUT and test generation technique (RT and CBT), we generated 1,000 nominal test cases. Recall that a nominal test case is a potentially valid test case intended to test the API under valid inputs. Then, we ran the test cases on the services under test and counted the number of actual valid test cases based on the 2XX responses obtained. Interestingly, we found that some of the services tested had dependencies not described in the API documentation. This was observed when obtaining 4XX status codes (client errors) with some input combinations that should be valid according to the documentation. For example, when using the `channelType` parameter in the YouTube API, the `type` parameter must be set to `'channel'`, although this dependency is not documented. In order to assess the effect of the missing dependencies, we defined them in the specification and included them in the evaluation as variants of the original SUTs, denoted with * after their name in Table 2. Overall, we added 4 new dependencies and updated 9 dependencies in 4 out of the 9 services under test. The experiments were performed in a standard PC with an Intel i5 processor, 16GB of RAM and an SSD, running on Windows 10 and Java JDK 8.

Results. Table 2 shows the results of the experiment. For each SUT and test generation technique, the table shows the percentage of valid test cases generated (column *Valid*) and the time required to generate the 1,000 test cases in seconds (column *Time*). Note that test case execution times are not included since those are independent of the test case generation approach.

As expected, RT struggled to generate valid test cases in most of the APIs, with the percentage of valid test cases ranging from 1.3% (Stripe) to 89% (Foursquare), 40.1% on average. In contrast, CBT successfully managed to generate 100% valid test cases in the API operations of GitHub, Tumblr and YouTube-GV. Similarly, CBT generated 100% valid test cases in the operations of Foursquare, Stripe and YouTube-S once the missing dependencies were included in the specification (rows denoted with *). Some of the test cases generated by CBT did not obtain successful responses in the SUTs of Yelp and YouTube-GCT. Interestingly, we found this was due to actual faults in those services, as

SUT	RT		CBT	
	Valid (%)	Time (s)	Valid (%)	Time (s)
Foursquare	89.0	1.4	93.6	107.0
Foursquare*	-	-	100	105.4
GitHub	62.1	3.6	100	96.8
Stripe-CC	17.1	0.4	82.0	102.9
Stripe-CC*	-	-	100	105.8
Stripe-CP	1.3	1.7	46.4	109.7
Stripe-CP*	-	-	100	108.8
Tumblr	65.5	0.2	100	100.3
Yelp	54.6	1.7	97.1	102.8
YouTube-GCT	20.5	0.6	99.9	95.9
YouTube-GV	49.2	7.5	100	114.2
YouTube-S	1.6	1.0	49.2	104.3
YouTube-S*	-	-	100	104.0
Mean	40.1	2.0	85.4 (99.7)	104.5

Table 2: Percentage of valid test cases and test case generation times.

discussed in the next section. CBT generated an average of 85.4% valid test cases in the services under test using the dependencies described in their documentation, and 100% (99.7% counting fault-revealing test cases) when considering all the dependencies, including those missing in the API documentation. Overall, out of 1,000 test cases, CBT generated between 11% (FourSquare) and 99% (Stripe-CP) more valid test cases than RT, 59.9% on average.

RT took 2 seconds on average to generate 1,000 test cases, whereas CBT took 104.5 seconds (less than 2 minutes) due to the overhead introduced by the constraint solver. However, the increment in the execution time of CBT is negligible compared to its potential to generate valid test cases and to detect failures. To investigate this further, we measured the time required by both techniques, RT and CBT, to generate and run test cases in the service of Stripe-CP until having 1,000 successful responses. RT took more than 10 hours and 73K total generated test cases. CBT took 11 minutes and 1K test cases.

Based on the results obtained, we can answer RQ1 as follows:

CBT can generate 100% valid test cases for RESTful web services, provided that dependencies are correctly specified. This means an average increment of 60% over RT (99% in highly constrained APIs) at a low price in terms of generation time.

4.4 Experiment 2: Detection of Failures

In this experiment, we aim to answer RQ2 by evaluating the effectiveness of RT and CBT in detecting failures in real-world APIs with inter-parameter dependencies. Next, we explain the experimental setup and the main findings.

Setup. For each SUT and test case generation technique, we generated 1,000 nominal test cases and 1,000 faulty test cases, 2,000 test cases in total. Faulty test

SUT	RT				CBT					
	5XX	OAS	$2XX_P$	Total	5XX	OAS	$2XX_P$	$2XX_D$	4XX	Total
Foursquare	0	1,042	127	1,169	1	910	65	424	64	1,464
GitHub	0	0	487	487	0	0	236	0	0	236
Stripe-CC	0	0	0	0	0	0	0	0	180	180
Stripe-CP	0	0	0	0	0	0	0	0	535	535
Tumblr	0	389	806	1,195	0	492	411	160	0	1,063
Yelp	48	19	0	67	50	42	0	68	1	161
YouTube-GCT	0	0	0	0	0	1	0	8	1	10
YouTube-GV	0	0	2	2	0	0	2	114	0	116
YouTube-S	0	0	0	0	0	5	0	0	508	513
Total	48	1,450	1,422	2,920	51	1,450	714	774	1,289	4,278

Table 3: Failures found by RT and CBT.

cases in RT were generated by violating the specification of individual parameters, e.g., omitting a mandatory parameter in the API request. In CBT, however, faulty test cases were divided into two groups: 500 test cases following the same approach as in RT, and 500 test cases violating one or more inter-parameter dependencies. To identify wrong outputs, we used the five test oracles explained in Section 3.3, i.e., server errors (*5XX*), conformance to the OAS specification (*OAS*), faulty requests obtaining successful responses ($2XX_P$ and $2XX_D$) and valid requests obtaining client error responses (*4XX*). Recall that oracles $2XX_D$ and *4XX* are only applicable in CBT as they rely on checking whether inter-parameter dependencies hold.

Results. Table 3 shows the number of failures detected in the services by each test case generation technique and test oracle. Both techniques succeeded in finding failures, but CBT proved more effective, as it uncovered 4,278 failures in all the services under test, whereas RT revealed 2,920 failures in 5 out of 9.

Regarding test oracles, both techniques uncovered a similar number of failures with oracles *5XX* and *OAS*. RT found twice the failures with test oracle $2XX_P$, but this was expected since it was checked 1,000 times in RT against 500 times in CBT. The true potential of CBT is leveraged with our two novel oracles $2XX_D$ and *4XX*. In fact, they sufficed to reveal 2,063 failures in 8 out of 9 services on their own. It is noteworthy that these failures are undetectable by current state-of-the-art techniques.

Oracle *4XX* uncovered a total of 1,289 failures in 6 services. These failures are specially critical: client errors should not be obtained when requests are well formed. Since we are using a black-box approach, it is difficult to know the exact number of distinct faults causing these failures. However, we analysed the error messages returned by the services and managed to classify the failures in multiple *potential* bugs. Due to space limitations, we describe three of the bugs uncovered with this oracle below, and refer the reader to the supplementary material for a comprehensive list [21]:

– In the Yelp service, when setting the `location` to `'Egypt'` and the `locale` to `'fi_FI'` (Finnish), the error `LOCATION_NOT_FOUND` (400 status code) is re-

turned. However, we noticed that changing the locale to Italian, for instance, makes the error disappear and actual results are returned.
- In the YouTube-GCT service, a valid request obtained an error with the following message: *"Check the structure of the* `commentThread` *resource in the request body to ensure that it is valid"*. However, no body was included in the request (actually the operation does not allow it), and so this failure becomes hard to debug.
- In the YouTube-S service, there exist two undocumented dependencies: (1) when using the `channelType` parameter, `type` must be set to `'channel'`; and (2) when using the `location` parameter, `type` must be set to `'video'`. These two unspecified dependencies caused 1 of every 2 requests to be invalid.

As for oracle $2XX_D$, most failures are related to inter-parameter dependencies being wrongly specified in the API documentation, or not correctly implemented in the API itself. For instance, the Yelp service defines the parameters `open_now` and `open_at` as mutually exclusive, nevertheless, a request including both parameters with `open_now` set to `'false'` will return a successful response.

In addition to the failures uncovered by our two novel oracles, RESTest successfully found other types of errors such as 500 status codes in Foursquare and Yelp and disconformities with the OAS specification in Tumblr and YouTube, among others. All things considered, we can answer RQ2 as follows:

> *CBT is highly effective at revealing failures having found bugs in the nine services under test. About half of the failures detected by CBT (2K out of 4K) were not detected by RT.*

5 Threats to Validity

The evaluation performed is subject to a number of validity threats.

Internal validity. *Are there factors that might affect the results of our evaluation?* A possible threat in this regard is the existence of bugs in the tools used, namely, RESTest and IDLReasoner. To mitigate this threat, both tools have been thoroughly tested using standard testing techniques such as equivalence partitioning and combinatorial testing. Furthermore, the tools with their test suites and the results of our experiments are freely available [21], thereby allowing full replication of the evaluation performed. Related to this, we had to manually write the OAS specifications of three services. To minimise bias, we created them solely based on their online documentation, and all specifications were independently revised by at least two authors. Another possible threat is related to the randomness of the testing techniques used (RT and CBT). A thorough evaluation should have included more repetitions per experiment (e.g., 10-30) and statistical analysis. However, due to the strict quota restrictions of the commercial APIs tested, it was not possible to do so (e.g., the YouTube-S service accepts only 100 requests per day). Despite this limitation, the total

number of test cases generated (40K) and failures found (7K) make us remain confident about the significance of the results obtained.

External validity. *To what extent can we generalise the findings?* We tested 9 services from 6 highly popular web APIs, nevertheless, this might not be a sufficiently representative sample. To minimise this threat, we selected API operations of different types (read and create), from different application domains (e.g., financial and social), with different numbers of parameters (from 5 to 31) and containing the eight patterns of dependencies found in our study of real-world APIs [11].

6 Related Work

RESTful API testing approaches can be classified into white-box and black-box. Arcuri [2] is the only author who advocates for white-box testing. He proposed a search-based approach, where test cases are generated aiming to maximise code coverage. Black-box testing approaches do not require access to the source code. Segura et al. [19] proposed to analyse the outputs returned by the service after similar requests. They managed to find bugs when inconsistencies among those outputs were detected, e.g., the API returns more data when using a filter than when no filter is applied. Other approaches achieve a higher degree of automation by leveraging the OAS specification of the API [3, 4, 9, 23]. Ed-douibi et al. [4] tested individual API operations using random, default and example parameter values present in the OAS document. Other authors [3, 9, 23] tested sequences of operations by inferring dependencies among them (e.g., creating a resource and retrieving it). For the generation of input test values, Karlsson et al. [9] resorted to property-based testing (PBT), while Atlidakis et al. [3] and Viglianisi et al. [23] used data dictionaries. All these approaches are limited in the oracles they can use, as they can only check the conformance to the OAS document, the absence of 5XX status codes and the correct management of invalid inputs. They cannot be certain about whether a given API call is valid or not, since it may violate some inter-parameter dependency. Neglecting this limitation would lead to false positives for those APIs containing dependencies, as what happened to Ed-douibi et al.: *"four errors were linked to the limitation of OpenAPI to define mutually exclusive required parameters"* [4]. Atlidakis et al. [3] proposed four additional oracles related to operation sequences (e.g., a resource that was deleted must no longer be accessible) but, again, these oracles have no effect if no valid calls to the service are generated in the first place. In constrast to previous approaches, RESTest supports the automated management of inter-parameter dependencies, enabling a deeper and faster evaluation of the SUT through the systematic generation of valid and invalid input combinations.

In the context of constraint-based testing for web services, the most related work is probably that of Sun et al. [20]. They proposed CxWSDL, a WSDL [24] extension to specify six different types of behaviour constraints such as the order in which operations should be invoked. Test cases were automatically derived from the specification using a constraint solver. Inconsistencies in the services

tested were found when some constraint was violated. Li et al. [10] presented a constrained combinatorial approach to generate optimal test suites avoiding forbidden combinations of parameters. Xu et al. [25] proposed testing web service robustness by violating constraints, including inter-parameter dependencies, which were extracted from the OWL-S [16] specification of the service. Compared to these papers, we support a wider range of inter-parameter dependencies, including the eight dependency patterns defined in [11], and we focus on RESTful APIs as the current de facto standard for web integration. Further, our approach is integrated into RESTest, an open-source framework that can be easily extended with other test generation strategies as well as testing frameworks and libraries.

7 Conclusion and Future Work

This paper presents RESTest, a framework for automated black-box testing of RESTful web APIs. RESTest implements a novel constraint-based testing approach that leverages the specification of inter-parameter dependencies to automatically generate valid calls to the service, i.e., those satisfying all input constraints. We showed that current random testing techniques can be extremely inefficient in generating valid requests and therefore are unable to exercise the actual functionality of the services, e.g., 98 out of every 100 random test cases violated inter-parameter dependencies in YouTube. In contrast, RESTest can efficiently generate 100% valid test cases when providing the specification of inter-parameter dependencies. More importantly, RESTest implements two novel oracles to evaluate how the API responds to constraint-satisfying and constraint-violating test cases. This allowed us to reveal more than 4K failures uncovering bugs in all the services under test.

Several challenges remain for future work. On the one hand, we plan to implement currently missing features in RESTest, such as testing of sequences of operations and search-based online testing approaches. This will allow us to perform a more extensive evaluation of the framework. On the other hand, we intend to make RESTest SLA-aware with SLA4OAI [6], so that it can be deployed in API gateways such as Governify [7] and perform autonomous functional and non-functional testing of microservices architectures.

Acknowledgements

This work has been partially supported by the European Commission (FEDER) and Junta de Andalucia under projects APOLO (US-1264651) and EKIPMENT-PLUS (P18-FR-2895), by the Spanish Government under project HORATIO (RTI2018-101204-B-C21), and by the FPU scholarship program, granted by the Spanish Ministry of Education and Vocational Training (FPU17/04077). We would also like to thank Ramon Fernandez for his technical support during the development of RESTest.

References

1. APIs.guru, https://apis.guru, accessed April 2020
2. Arcuri, A.: RESTful API Automated Test Case Generation with EvoMaster. ACM TOSEM **28**(1), 1–37 (2019)
3. Atlidakis, V., Godefroid, P., Polishchuk, M.: Checking Security Properties of Cloud Services REST APIs. In: ICST (2020)
4. Ed-douibi, H., Izquierdo, J.L.C., Cabot, J.: Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In: EDOC. pp. 181–190 (2018)
5. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis (2000)
6. Gamez-Diaz, A., Fernandez, P., Ruiz-Cortes, A.: Automating SLA-Driven API Development with SLA4OAI. In: ICSOC. pp. 20–35 (2019)
7. Gamez-Diaz, A., Fernandez, P., Ruiz-Cortés, A.: Governify for APIs: SLA-Driven Ecosystem for API Governance. In: ESEC/FSE. pp. 1120–1123 (2019)
8. Gotlieb, A.: Constraint-Based Testing: An Emerging Trend in Software Testing. In: Advances in Computers, vol. 99, pp. 67–101. Elsevier (2015)
9. Karlsson, S., Causevic, A., Sundmark, D.: QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs. In: ICST (2020)
10. Li, Y., Sun, Z.a., Fang, J.Y.: Generating an Automated Test Suite by Variable Strength Combinatorial Testing for Web Services. CIT **24**(3), 271–282 (2016)
11. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: A Catalogue of Inter-Parameter Dependencies in RESTful Web APIs. In: ICSOC. pp. 399–414 (2019)
12. Martin-Lopez, A., Segura, S., Müller, C., Ruiz-Cortés, A.: Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. Submitted to *IEEE Transactions on Services Computing* (2020), https://bit.ly/2ECr9rc
13. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: Test Coverage Criteria for RESTful Web APIs. In: A-TEST. pp. 15–21 (2019)
14. MiniZinc: Constraint Modeling Language, https://www.minizinc.org, accessed April 2020
15. OpenAPI Specification, https://www.openapis.org, accessed April 2020
16. Semantic Markup for Web Services (OWL-S), https://www.w3.org/Submission/OWL-S, accessed May 2020
17. REST Assured, http://rest-assured.io, accessed April 2020
18. Richardson, L., Amundsen, M., Ruby, S.: RESTful Web APIs. O'Reilly Media, Inc. (2013)
19. Segura, S., Parejo, J.A., Troya, J., Ruiz-Cortés, A.: Metamorphic Testing of RESTful Web APIs. IEEE TSE **44**(11), 1083–1099 (2018)
20. Sun, C.a., Li, M., Jia, J., Han, J.: Constraint-Based Model-Driven Testing of Web Services for Behavior Conformance. In: ICSOC. pp. 543–559 (2018)
21. Supplementary material of the paper, https://github.com/isa-group/icsoc-2020-supplementary-material
22. Swagger, http://swagger.io, accessed April 2020
23. Viglianisi, E., Dallago, M., Ceccato, M.: RestTestGen: Automated Black-Box Testing of RESTful APIs. In: ICST (2020)
24. Web Services Description Language (WSDL) Version 2.0, https://www.w3.org/TR/wsdl20, accessed May 2020
25. Xu, L., Yuan, Q., Wu, J., Liu, C.: Ontology-based Web Service Robustness Test Generation. In: WSE. pp. 59–68 (2009)