REST-ler: Automatic Intelligent REST API Fuzzing

Vaggelis Atlidakis[*] *Columbia University*
Patrice Godefroid *Microsoft Research*
Marina Polishchuk *Microsoft Research*

Abstract

Cloud services have recently exploded with the advent of powerful cloud-computing platforms such as Amazon Web Services and Microsoft Azure. Today, most cloud services are accessed through REST APIs, and Swagger is arguably the most popular interface-description language for REST APIs. A Swagger specification describes how to access a cloud service through its REST API (*e.g.,* what requests the service can handle and what responses may be expected).

This paper introduces *REST-ler*, the first automatic intelligent REST API security-testing tool. *REST-ler* analyzes a Swagger specification and generates tests that exercise the corresponding cloud service through its REST API. Each test is defined as a sequence of requests and responses. *REST-ler* generates tests intelligently by (1) inferring *dependencies among request types* declared in the Swagger specification (*e.g.,* inferring that "a request B should not be executed before a request A" because B takes as an input argument a resource-id *x* returned by A) and by (2) analyzing *dynamic feedback* from responses observed during prior test executions in order to generate new tests (*e.g.,* learning that "a request C after a request sequence A;B is refused by the service" and therefore avoiding this combination in the future). We show that these two techniques are necessary to thoroughly exercise a service under test while pruning the large search space of possible request sequences. We also discuss the application of *REST-ler* to test GitLab, a large popular open-source self-hosted Git service, and the new bugs that were found.

1 Introduction

Over the last decade, we have seen an explosion in cloud services for hosting software applications (Software-as-a-Service), for data processing (Platform-as-a-Service), and for providing general computing infrastructure (Infrastructure-as-a-Service). Today, most cloud services, such as those provided by Amazon Web Services and Microsoft Azure, are programmatically accessed through REST APIs [9] by third-party applications [1] and other services [25]. Meanwhile, Swagger [36] (recently renamed OpenAPI) has arguably become the most popular interface-description language for REST APIs. A Swagger specification describes how to access a cloud service through its REST API, including what requests the service can handle and what responses may be expected in what format.

Tools for automatically testing cloud services via their REST APIs and checking whether those services are reliable and secure are still in their infancy. The most sophisticated testing tools currently available for REST APIs capture live API traffic, and then parse, fuzz and replay the traffic with the hope of finding bugs [3, 29, 6, 37, 2]. Many of these tools were born as extensions of more established web-site testing and scanning tools (see Section 6). Since these REST API testing tools are all recent and not widely used, it is currently unknown how effective they are in finding bugs and how security-critical those bugs are.

In this paper, we introduce *REST-ler*, the first automatic intelligent REST API fuzzing tool. Fuzzing [35] means automatic test generation and execution with the goal of finding security vulnerabilities. Unlike other REST API testing tools, *REST-ler* performs a lightweight static analysis of an entire Swagger specification, and then generates and executes tests that exercise the corresponding cloud service through its REST API. Each test is defined as a sequence of requests and responses. *REST-ler* generates tests *intelligently* by

1. inferring *dependencies among request types* declared in the Swagger specification (*e.g.,* inferring that a resource included in the response of a request A is necessary as input argument of another request B, and therefore that A should be executed before B), and by
2. analyzing *dynamic feedback* from responses observed during prior test executions in order to generate new tests (*e.g.,* learning that "a request C after a request sequence A;B is refused by the service" and therefore avoiding this combination in the future).

We present empirical evidence that these two techniques (partly inspired by prior work on API testing for object-oriented programs [27]) are necessary to thoroughly exercise a service under test while pruning the large search

[*]The work of this author was mostly done while visiting Microsoft Research.

space defined by all possible request sequences. *REST-ler* also implements several search strategies (some inspired by prior work on model-based testing [40]), and we compare their effectiveness while fuzzing GitLab [11], a large popular open-source self-hosted Git service with a complex REST API. During the course of our experiments, we found several new bugs in GitLab, including security-relevant ones (see Section 5).

In summary, this paper makes the following contributions:

- We introduce *REST-ler*, the first automatic intelligent fuzzing tool for REST APIs which analyzes a Swagger specification, automatically infers dependencies among request types, generates tests defined as request sequences satisfying those dependencies, and dynamically learns what request sequences are valid or invalid by analyzing the service responses to those tests.
- We present detailed experimental evidence showing that the techniques used in *REST-ler* are necessary for effective automated REST API fuzzing.
- We also present experimental results obtained with three different strategies for searching the large search space defined by all possible request sequences, and discuss their strengths and weaknesses.
- We present a detailed case study with GitLab, a large popular open-source self-hosted Git service, and discuss several new bugs found so far and their severity.

This paper is organized as follows. In the next Section, we describe Swagger specifications and how they are processed by *REST-ler*. In Section 3, we present the main test-generation algorithm used in *REST-ler*, and discuss different search strategies and other implementation details. In Section 4, we present experimental results evaluating the effectiveness of the test-generation techniques and search strategies implemented in *REST-ler*. In Section 5, we discuss several new bugs found in GitLab during the course of this work. We discuss related work in Section 6 and conclude in Section 7.

2 Processing API Specifications

In this paper, we consider cloud services accessible through REST APIs described with a Swagger specification. A Swagger specification describes how to access a cloud service through its REST API (*e.g.,* what requests the service can handle and what responses may be expected). A client program can send messages, called *requests*, to a service and receive messages back, called *responses*. Such messages are sent over the HTTP protocol. Given a Swagger specification, open-source Swagger tools can automatically generate a web UI that allows users to view the documentation and interact with the API via a web browser.

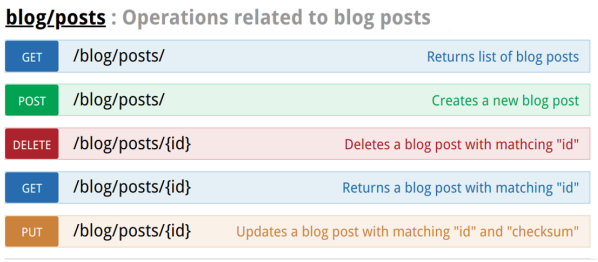

Fig. 1: **Swagger Specification of Blog Posts Service**

```
basePath: '/api'
swagger: '2.0'
definitions:
  "Blog Post":
    properties:
      body:
        type: string
      id:
        type: integer
    required:
    -body
    type: object

paths:
  "/blog/posts/"
    post:
      parameters:
      -in: body
        name: payload
        required: true
      schema:
        ref: "/definitions/Blog Post"
```

```
from restler import requests
from restler import dependencies

def parse_posts(data):
  post_id = data["id"]
  dependencies.set_var(post_id)

request = requests.Request(
  restler_static("POST"),
  restler_static("/api/blog/posts/"),
  restler_static("HTTP/1.1"),
  restler_static("{"),
  restler_static("body:"),
  restler_fuzzable("string"),
  restler_static("}"),
  'post_send': {
      'parser': parse_posts,
      'dependencies': [
          post_id.writer(),
      ]
  }
)
```

Fig. 2: **Swagger Specification and Automatically Derived *REST-ler* Grammar.** Shows a snippet of Swagger specification in YAML format (left) and the corresponding grammar generated by *REST-ler* (right).

An example of Swagger specification in web-UI form is shown in Figure 1. This specification describes five types of requests supported by a simple service for hosting blog posts. This service allows users to create, access, update and delete blog posts. In a web browser, clicking on any of these five request types expands the description of the request type.

For instance, selecting the second request, which is a POST request, reveals text similar to the left of Figure 2. This text is in YAML format and describes the exact syntax expected for that specific request and its response. In this case, the `definition` part of the specification indicates that an object named `body` of type `string` is required and that an object named `id` of type `integer` is optional (since it is not required). The `path` part of

```
Sending: POST /api/blog/posts/ HTTP/1.1
Accept: application/json
Content−Type: application/json
Host: localhost:8888
{"body":"sampleString"}

Received: HTTP/1.1 201 CREATED
Content−Type: application/json
Content−Length: 37
Server: Werkzeug/0.14.1 Python/2.7.12
Date: Sun, 01 Apr 2018 05:10:32 GMT
{"body": "sampleString", "id": 5889}
```

Fig. 3: ***REST-ler* Trace of HTTP Request and Response.** Shows a network-layer *REST-ler* trace with a POST request creating a blog post and the respective response.

the specification describes the HTTP-syntax for a POST request of this type as well as the format of the expected response.

From such a specification, *REST-ler* automatically constructs the test-generation grammar shown on the right of Figure 2. This grammar is encoded in executable `python` code. It mainly consists of code to generate an HTTP request, of type POST in this case. Each command `restler_static` simply appends the string it takes as argument without modifying it. In contrast, the command `restler_fuzzable` takes as argument a value type (like `string` in this example) and replaces it by one value of that type taken from a (small) *dictionary* of values for that type. How dictionaries are defined and how values are selected is discussed in the next section.

The grammar on the right also includes code to process the expected response of the request. In this case, the response is expected to return a new object named `id` of type `integer`. Using the `schema` specified on the left, *REST-ler* automatically generates the function `parse_posts` shown on the right. Figure 3 shows an example of a HTTP-level trace of a single POST request generated by *REST-ler* for the blog posts service and the corresponding response.

By similarly analyzing the other request types described in this Swagger specification, *REST-ler* will infer automatically that `id`s returned by such POST requests are necessary to generate well-formed requests of the last three request types shown in Figure 1 which each requires an `id`. These *dependencies* are extracted by *REST-ler* when processing the Swagger specification and are later used for test generation, as described next.

```
1  Inputs: swagger_spec, maxLength
2  # Set of requests parsed from the Swagger API spec
3  reqSet = PROCESS(swagger_spec)
4  # Set of request sequences (initially empty)
5  seqSet = {}
6  # Main loop: iterate up to a given maximum sequence length
7  n = 1
8  while (n =< maxLength):
9      seqSet = EXTEND(seqSet, reqSet)
10     seqSet = RENDER(seqSet)
11     n = n + 1
12 # Extend all sequences in seqSet by appending
13 # new requests whose dependencies are satisfied
14 def EXTEND(seqSet, reqSet):
15     newSeqSet = {}
16     for seq in seqSet:
17         for req in reqSet:
18             if DEPENDENCIES(seq, req):
19                 newSeqSet = newSeqSet + concat(seq, req)
20     return newSeqSet
21 # Concretize all newly appended requests using dictionary values,
22 # execute each new request sequence and keep the valid ones
23 def RENDER(seqSet):
24     newSeqSet = {}
25     for seq in seqSet:
26         req = last_request_in(seq)
27         V⃗ = tuple_of_fuzzable_types_in(req)
28         for v⃗ in V⃗:
29             newReq = concretize(req, v⃗)
30             newSeq = concat(seq, newReq)
31             response = EXECUTE(newSeq)
32             if response has a valid code:
33                 newSeqSet = newSeqSet + newSeq
34             else:
35                 log error
36     return newSeqSet
37 # Check that all objects referenced in a request are produced
38 # by some response in a prior request sequence
39 def DEPENDENCIES(seq, req):
40     if CONSUMES(req) ⊆ PRODUCES(seq):
41         return True
42     else:
43         return False
44 # Objects required in a request
45 def CONSUMES(req):
46     return object_types_required_in(req)
47 # Objects produced in the responses of a sequence of requests
48 def PRODUCES(seq):
49     dynamicObjects = {}
50     for req in seq:
51         newObjs = objects_produced_in_response_of(req)
52         dynamicObjects = dynamicObjects + newObjs
53     return dynamicObjects
```

Fig. 4: **Main Algorithm used in *REST-ler*.**

3 *REST-ler*

3.1 Test Generation Algorithm

The main algorithm for test generation used by *REST-ler* is shown in Figure 4 in `python`-like notation. It

starts (line 3) by processing a Swagger specification as discussed in the previous section. The result of this processing is a set of request types, denoted `reqSet` in Figure 4, and of their dependencies (more on this later).

The algorithm computes a set of request sequences, denoted `seqSet` and initially empty (line 5). At each iteration of its main loop (line 8), the algorithm computes *all valid* request sequences `seqSet` of length n, starting with $n = 1$ before moving to $n+1$ and so on until a user-specified `maxLength` is reached. Computing `seqSet` is done in two steps.

First, the set of valid request sequences of length $n-1$ is *extended* (line 8) by appending at the end of each sequence one new request whose dependencies are satisfied, for all possible requests, as described in the EXTEND function (line 14). The function DEPENDENCIES (line 39) checks that all the object types required in the last request, denoted by `CONSUMES(req)`, are produced by some response in the request sequence preceding it, denoted by `PRODUCES(seq)`. If all the dependencies are satisfied, the new sequence of length n is retained (line 19), otherwise it is discarded.

Second, each newly-extended request sequence whose dependencies are satisfied is *rendered* (line 10) one by one as described in the RENDER function (line 23). For every newly-appended request (line 26), the list of all fuzzable primitive types in the request is computed (line 27) (those are identified by `restler_fuzzable` in the code shown on the right of Figure 2). Then, each fuzzable primitive type in the request is replaced by one concrete value of that type taken out of a finite (and small) dictionary of values. The function RENDER generates all possible such combinations (line 28). Each combination thus defines a fully-defined request `newReq` (line 29) which is HTTP-syntactically correct. The function RENDER then *executes* this new request sequence (line 31), and checks its response: if the response has a valid return code (defined here as any code in the `200` range), the new request sequence is "valid" and retained (line 33), otherwise it is discarded and the received error code is logged for further analysis and debugging.

More precisely, the function EXECUTE executes each request in a sequence request one by one, each time checking that the response is valid, extracting and memoizing dynamic objects (if any), and providing those in subsequent requests in the sequence if needed, as determined by the dependency analysis; the response returned by function EXECUTE in line 31 refers to the response received for the last, newly-appended request in the sequence. Note that if a request sequence produces more than one dynamic object of a given type, the function EXECUTE will memoize all of those objects, but will provide them later when needed by subsequent requests in the exact order in which they are produced; in other words, the function EXECUTE will not try different ordering of such objects. If a dynamic object is passed as argument to a subsequent request and is "destroyed" after that request, that is, it becomes unusable later on, *REST-ler* will detect this by receiving an invalid response (`400` or `500` error code) when attempting to reuse that unusable object, and will then discard that request sequence.

For each fuzzable primitive type, the algorithm of Figure 4 uses a small set of values of that type, called *dictionary*, and picks one of these values in order to *concretize* that fuzzable type (lines 27-29). For instance, for fuzzable type `integer`, *REST-ler* might use a small dictionary with the values 0, 1, and -10, while for fuzzable type `string`, a dictionary could be defined with the values "sampleString", the empty string and one very long fixed string. The user defines those dictionaries.

By default, the function RENDER of Figure 4 generates *all* possible combinations of dictionary values for every request with several fuzzable types (see line 28). For large dictionaries, this may result in astronomical numbers of combinations. In that case, a more scalable option is to randomly sample each dictionary for one (or a few) values, or to use *combinatorial-testing* algorithms [8] for covering, say, every dictionary value, or every pair of values, but not every k-tuple. In the experiments reported later, we used small dictionaries and the default RENDER function shown in Figure 4.

The function EXTEND of Figure 4 generates *all* request sequences of length $n+1$ whose dependencies are satisfied. Since n is incremented at each iteration of the main loop of line 8, the overall algorithm performs a *breadth-first search* (BFS) in the search space defined by all possible request sequences. In Section 4, we report experiments performed also with two additional search strategies: BFS-Fast and RandomWalk.

BFS-Fast. In function EXTEND, the loops of line 17 and line 18 are swapped in such a way that every request `req` in `reqSet` is appended only once to some request sequence in `seqSet` in line 19, resulting in a smaller set `newSeqSet` which covers (*i.e.,* includes at least once) every request but does not generate all valid request sequences. Like BFS, BFS-Fast thus still provides full grammar-coverage at each iteration of the main loop in line 8, but it generates fewer request sequences, which allows it to go deeper more quickly than BFS.

RandomWalk. In function EXTEND, the two loops of line 17 and line 18 are eliminated; instead, the function now returns a single new request sequence whose dependencies are satisfied, and generated by *randomly* selecting one request sequence `seq` in `seqSet` and one request in `reqSet`. (The function randomly chooses such a pair until all the dependencies of that pair are satisfied.) This search strategy will therefore explore the

search space of possible request sequences deeper more quickly than BFS or BFS-Fast. When RandomWalk can no longer extend the current request sequence, it restarts from scratch from an empty request sequence. (Since it does not memoize past request sequences between restarts, it might regenerate the same request sequence again in the future.)

3.2 Implementation Details

We have implemented *REST-ler* with 2,230 lines of python code. Its functionality is split into four modules: the main application entry point, the parser and compiler module, the core fuzzing engine, and the logging module.

The main application entry-point is responsible for starting a fuzzing session according to a set of configuration parameters controlling: the Swagger specification which should be used to derive an input grammar and perform fuzzing; the desired search strategy, including BFS, BFS-Fast, and RandomWalk; the maximum sequence length and the maximum fuzzing time; the HTTP status codes indicating errors (*e.g.,* `500`); the dictionary of fuzzing mutations to be used when rendering fuzzable primitive types; and the port and IP-address of the fuzzing target along with any additional authorization tokens.

The parser and compiler module is responsible for parsing a Swagger specification and generating a *REST-ler* grammar for fuzzing the target service. In the absence of a Swagger specification, the user can manually provide a *REST-ler* grammar to be used for fuzzing.

The core engine module implements the algorithm of Figure 4, and is responsible for rendering API requests and for composing sequences of requests using any of the supported search strategies. The rendered request sequences are sent to the target service using `send` on `python` sockets. Similarly, the corresponding response is received using `recv` on `python` sockets.

Finally, the logging module monitors client/service interactions and traces all messages exchanged via `python` sockets. Sequences of requests sent to the target service, along with the corresponding HTTP status codes received from the service, are persistently stored and inspected for errors and bug detection.

3.3 Current Limitations

Currently *REST-ler* only supports token-based authorization, such as OAUTH [26], and there in no support for operations that involve web-UI interactions. Moreover, *REST-ler* does not support requests for API endpoints that depend on server-side redirects (*e.g.,* `301` "Moved Permanently", `303` "See Other", and `307` "Temporary Redirect"). Finally, our current *REST-ler* prototype can only find bugs defined as unexpected HTTP status codes, namely `500` "Internal Server Error". Such a simple test oracle cannot detect vulnerabilities that are not visible though HTTP status codes (*e.g.,* "Information Exposure" and others). Despite these limitations, *REST-ler* is already useful in finding security-relevant bugs, as will be discussed in Section 5.

4 Evaluation

We present experimental results obtained with *REST-ler* that answer the following questions:

Q1: Are both inferring dependencies among request types and analyzing dynamic feedback necessary for effective automated REST API fuzzing? (Section 4.2)

Q2: Are tests generated by *REST-ler* exercising deeper service-side logic as sequence length increases? (Section 4.3)

Q3: What search strategy should be used in *REST-ler*? (Section 4.4)

We answer the first question (Q1) using a simple Model-View-Controller (MVC) blog posts service with a REST API. We answer (Q2), and (Q3) using GitLab, an open-source, production-scale [1] web service for self-hosted Git. We conclude the evaluation by discussing in Section 4.5 how to bucketize (*i.e.,* group together) the numerous bugs that can be reported by *REST-ler* in order to facilitate their analysis. Afterwards, we move on to Section 5 where we discuss new bugs found in GitLab.

4.1 Experimental Setup

Blog Posts Service. We answer (Q1) using a simple blog posts service, written in 189 lines of python code using the Flask web framework [10] and following the MVC web development paradigm. Every blog post is persistently stored in a SQLite [33] database and has a user-assigned body, an automatically-assigned post id (primary key), and an automatically-derived checksum (SHA-1) of the blog post's body. The service's functionality is exposed over a REST API with a Swagger specification shown in Figure 1. This API contains five request types: (i) GET on `/posts` : returns all blog posts currently registered; (ii) POST on `/posts` : creates a new blog post (body: the text of the blog post); (iii) DELETE `/posts/id` : deletes a blog post; (iv) GET `posts/id` : returns the body and the checksum of an individual blog post; and (v) PUT `/posts/id` :

[1] GitLab [11] is used by more than 100,000 organizations, has millions of users, and has currently a 2/3 market share of the self-hosted Git market [14].

updates the contents of a blog post (body: the new text of the blog post and the checksum of the older version of the blog post's text). To model an imaginary subtle bug, at every update of a blog post (PUT request with body text and checksum) the service checks if the checksum provided in the request matches the recorded checksum for the current blog post, and if it does, an uncaught exception is raised. Thus, this bug will be triggered and detected only if dependencies on dynamic objects shared across requests are taken into account during test generation.

Gitlab. We answer (Q2) and (Q3) using GitLab, an open-source web service for self-hosted Git. GitLab's back-end functionality is written in over 376K lines of ruby code using ruby-on-rails [31]. It follows the MVC web development paradigm and exposes its functionality over a REST API. GitLab's API is extensively documented and has hundreds of individual API requests spread across 63 groups [12]. There is a publicly available GitLab Swagger specification which we used for our experiments [15]. A typical GitLab deployment consists of the following components: (i) a low-level HTTP server used to proxy-pass the Rails Unicorn web server, (ii) a Sidekiq job queue which, in turn, uses (iii) redis as a non-persistent database, and (iv) a structured database for persistent storage. We follow this deployment and, unless otherwise specified, apply the following configuration settings: we use Nginx to proxypass the Unicorn web server and configure 10 Unicorn workers limited to up to 900MB of physical memory; we use postgreSQL for persistent storage configured with a pool of 10 workers; we use GitLab's default configuration for sidekiq queues and reddis workers; finally, we mount the repositories' destination folder in physical memory. According to GitLab's deployment recommendations, our configuration should scale up to 4,000 concurrent users [13].

Fuzzing Dictionaries. For the experiments in this section, we use the following dictionaries for fuzzable primitives types: *string* has possible values "sampleString" and "" (empty string); *integer* has possible values "0" and "1"; *boolean* has possible values "*true*" and "*false*".

All experiments were run on Ubuntu 16.04 Microsoft Azure VMs configured with eight Intel(R) Xeon(R) E5-2673 v3 @ 2.40GHz CPU cores and 28GB of physical memory, unless otherwise specified.

4.2 Techniques for Effective REST API Fuzzing

In this section, we report experimental results with our blog posts service to determine whether both (1) inferring dependencies among request types and (2) analyzing dynamic feedback are necessary for effective automated REST API fuzzing (Q1). We choose a controled experiment with a known simple blog-posts service in order to clearly measure and interpret the testing capabilities of the two core techniques being evaluated. Those capabilities are evaluated by measuring service code coverage and client-visible HTTP status codes.

Specifically, we compare results obtained when exhaustively generating all possible request sequences of length up to three, with three different test-generation algorithms:

1. *REST-ler* ignores dependencies among request types and treats dynamic objects – such as post `id` and `checksum` – as fuzzable primitive type `string` objects, while still analyzing dynamic feedback.
2. *REST-ler* ignores service-side dynamic feedback and does not eliminate invalid request sequences during its search, but still infers dependencies among request types and generates request sequences satisfying such dependencies.
3. *REST-ler* follows the algorithm of Figure 4 and uses both dependencies among request types and dynamic feedback.

Figure 5 shows the number of tests, *i.e.,* request sequences, generated up to maximum length 3 by each of these three algorithms, from left to right. At the top, the figure shows cumulative code coverage measured in lines of python code (using python's `settrace` system utility) and increasing over time, as well as when the sequence length increases (from 1 to 3). At the bottom, the figure shows the cumulative number of HTTP status code received as responses so far.

Code Coverage. First, we observe that without considering dependencies among request types (Figure 5, top left), code coverage is limited to up to 130 lines and there is no increase over time, despite increasing the length of request sequences. This is expected and illustrates the limitations of a naive approach to effectively test a service where values of dynamic objects like `id` and `checksum` cannot be randomly guessed or picked among values in a small predefined dictionary. On the other hand, by infering dependencies among requests manipulating such objects *REST-ler* achieves an increase in code coverage up to 150 lines of code (Figure 5, top both center and right).

Second, we can see that without considering dynamic feedback to prune invlid request sequences in the search space (Figure 5, top center), the number of tests generated grows quickly, even for a simple API. Specifically, without considering dynamic feedback (Figure 5, top center), *REST-ler* produces more that $4,600$ tests that take $1,750$ seconds and cover about 150 lines of code; in contrast, by considering dynamic feedback (Figure 5, top right), the state space is significantly reduced and *REST-*

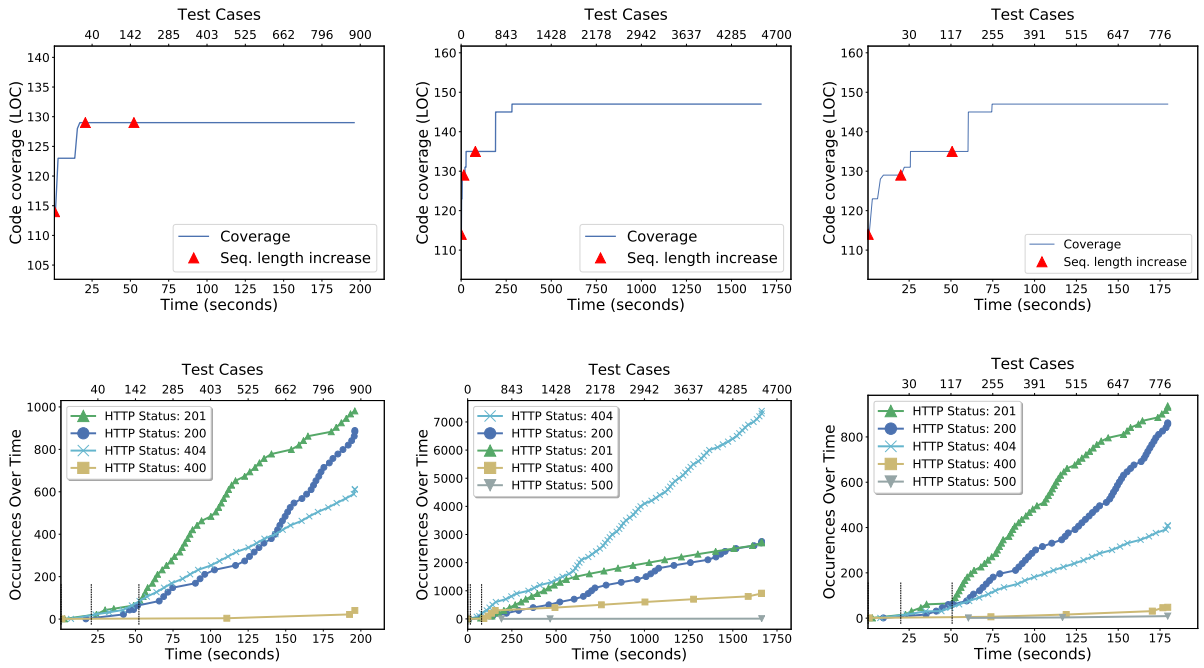

Fig. 5: **Code Coverage and HTTP Status Codes Over Time.** Shows the increase in code coverage over time (top) and the cumulative number of HTTP status codes received over time (bottom), for the simple blog posts service. *Left: REST-ler* ignores dependencies among request types. *Center: REST-ler* ignores dynamic feedback. *Right: REST-ler* utilizes both dependencies among request types and dynamic feedback. When leveraging both techniques, *REST-ler* achieves the best code coverage and finds the planted `500` "Internal Server Error" bug with the least number of tests.

ler can achieve the same code coverage with less than 800 test cases and only 179 seconds.

HTTP Status Codes. We make two observations. First, focusing on `40X` status codes, we notice a high number of `40X` responses when ignoring dynamic feedback (Figure 5, bottom center). This is expected since without considering service-side dynamic feedback, the number of possible invalid request sequences grows quickly. In contrast, considering dynamic feedback dramatically decreases the percentage of `40X` status codes from 60% to 26% (Figure 5, bottom left) and 20% (Figure 5, bottom right), respectively, without or with using dependencies among request types. Moreover, when also using dependencies among request types (Figure 5, bottom right), we observe the highest percentage of `20X` status codes (approximately 80%), indicating that *REST-ler* then exercises a larger part of the service logic, as also confirmed by coverage data (Figure 5, top right).

Second, when ignoring dependencies among request types, we see that no `500` status code are detected (Figure 5, bottom left), while *REST-ler* finds a handful of `500` status codes when using dependencies among request types (see (Figure 5, bottom left and bottom right). These `500` responses are triggered by the unhandled exception we planted in our blog posts service after a PUT blog update request with a checksum matching the previous blog post's body (see Section 4.1). As expected, ignoring dependencies among request types does not find this bug (Figure 5, bottom left). In contrast, analyzing dependencies across request types and using the checksum returned by a previous GET `/posts/id` request in a subsequent PUT `/posts/id` update request with the same `id` does trigger the bug. Morever, when also using dynamic feedback, the search space is pruned while preserving this bug, which is then found with the least number of tests (Figure 5, bottom right).

Overall, these experiments illustrate the complementarity between using dependencies among request types and using dynamic feedback, and show that both are needed for effective REST API fuzzing.

4.3 Deeper Service Exploration

In this section, we use GitLab to determine whether tests generated by *REST-ler* exercise more service-side logic and code as sequence length increases (Q2). In total, GitLab has 63 API groups and *REST-ler* identifies 358 API request types after analyzing GitLab's Swagger specification [12, 15]. Here, we constrain our investigation to five of GitLab's API groups, related to usual operations

API	Requests Tested	Seq. Len.	Coverage Increase	Tests	seqSet Size	Dynamic Objects
Commits	4 / 10	1	498	1	1	1
		2	959	66	5	66
		3	1453	483	93	900
		4	1486	7300	2153	18663
Branches	5 / 7	1	498	1	1	1
		2	926	5	3	5
		3	969	23	14	37
		4	986	139	82	289
		5	988	911	516	2119
		6	1026	4700	5652	12371
Issues	14 / 25	1	498	1	1	1
		2	879	514	257	514
		3	1158	8400	6964	16718
Repos	3 / 11	1	498	1	1	1
		2	871	18	5	18
		3	937	231	25	640
		4	954	1424	125	4611
		5	1030	5200	909	20713
Groups	10 / 20	1	687	25	25	1
		2	718	1250	1225	1226
		3	798	7900	13421	12466

Table 1: **Testing Common GitLab APIs with *REST-ler*.** Shows the increase in sequence length, code coverage, tests executed, `seqSet` size, and the number of dynamic objects being created, until a 5-hours timeout is reached. Longer request sequences gradually increase service-side code coverage.

with commits, branches, issues and issue notes, repositories and repository files, and groups and group membership. We target 36 out of 73 request types defined for the API groups under investigation, and also include a POST `/projects` request, which is a common root dependency for all five API groups. Indeed, we focus on request types with POST, PUT, and DELETE HTTP methods, which may modify the service's internal state by creating, updating and deleting resources, but we omit request types with GET methods which are here simple accessors and cause an unnecessary state space explosion.

For each API group, Table 1 gives the number of request types included in that API as well as the number of request types tested by *REST-ler* within the API (second column). The Table presents results of experiments performed with all five GitLab sub-APIs by the main test-generation algorithm of Figure 4 (thus using BFS) with a 5-hours timeout and while limiting the number of fuzzable primitive-types combinations to maximum $1,000$ combinations per request. Between experiments, we reboot the entire GitLab service to restart from the same initial state. For each API, as time goes by, the Table shows the increase (going down) in the sequence length, code coverage, tests executed, `seqSet` size, and the number of dynamic objects being created, until the 5-hours timeout is reached.

Code Coverage. We collect code coverage data by configuring Ruby's `Class: TracePoint` hook to trace GitLab's `service/lib` folder. Table 1 shows the cumulative code coverage achieved after executed all the request sequences generated by *REST-ler* for each sequence length, or until the 5-hours timeout expires. The results are incremental on top of 14,468 lines of code executed during service boot.

From Table 1, we can see a clear pattern across all five experiments: longer sequence lengths consistently lead to increased service-side code coverage. This is not surprising, especially for small sequence lengths, as some of the service functionality can only be exercised after at least a few requests are executed.

As an example, consider the GitLab functionality of "selecting a commit". According to GitLab's specification, selecting a commit requires two dynamic objects, a *project-id* and a *commit-id*, and the following dependencies of requests is implicit: (1) a user needs to create a project, (2) use the respective *project-id* to post a new commit, and then (3) select the commit using its *commit-id* and the respective *project-id*. Clearly, this operation can only be performed by sequences of three requests or more. For the Commit APIs, note the gradual increase in coverage from 498 to 959 to $1,453$ lines of code for sequence lengths of one, two, and three, respectively.

As is also expected, for API groups with fewer requests (*e.g.,* Commits, Repos, and Branches), *REST-ler*'s BFS reaches deeper sequences within the 5-hour time budget. Most notably, for the Branches API, service-side code coverage keeps gradually increasing for sequences of length up to five, and then reaches $1,026$ lines when the 5-hours limit expires. In contrast, for API groups with more requests (*e.g.,* Issues and Groups), *REST-ler* generates a higher number of shallow sequences before reaching the 5-hours timeout.

Tests, Sequence Sets, and Dynamic Objects. In addition to code coverage, Table 1 also shows the increase in the number of tests executed, the size of `seqSet` after the RENDER function returns (in line 10 of Figure 4), and the number of dynamic objects created by *REST-ler* in order to get that increased code coverage. We can clearly see that all those numbers are quickly growing, because the search space is rapidly growing as its depth increases and because of the BFS search strategy used here.

Nevertheless, we emphasize that, without the two key techniques evaluated in Section 4.2, this growth would be much worse. For instance, for the Commit API, the `SeqSet` size is $2,153$ and there are $18,663$ dynamic objects created by *REST-ler* for sequences of length up to four. By comparison, since the Commit API has four request types with an average of 17 rendering combinations, the number of all possible rendered request se-

API	Time (hours)	BFS		RandomWalk		Intersection
		Cov.	Len.	Cov.	Len. (restarts)	Cov.
Commits	1	65	4	0	10	1420
	3	45	4	0	12	1441
	5	44	4	0	14 (116)	1442
Branches	1	0	6	0	21	988
	3	0	6	0	21	988
	5	38	6	0	24 (462)	988
Issues	1	0	3	80	9	1020
	3	36	3	3	15	1119
	5	39	3	3	15 (60)	1119
Repos	1	90	5	0	12	940
	3	90	5	0	14	940
	5	90	5	0	14 (138)	940
Groups	1	0	3	31	23	754
	3	0	3	25	31	760
	5	28	3	16	31 (197)	770

Table 2: **Search Strategies and Code Coverage.** Compares test results for GitLab APIs using the *BFS* and *RandomWalk* search strategies, after 1, 3, and 5 hours. The table shows the maximum length of request sequences and the unique lines of code covered by each search strategy, as well as their intersection (excluding service-boot coverage). For *RandomWalk*, the total number of restarts after 5 hours is also shown in parenthesis. Although both strategies mostly cover overlapping lines of code, some are covered only by one or the other. Overall, after 5 hours, *BFS* often generates the best coverage.

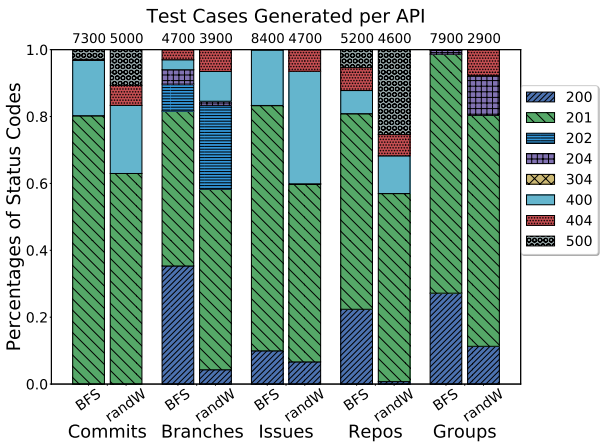

Fig. 6: **Distribution of HTTP Status Codes for GitLab APIs.** Shows the HTTP status codes collected during testing with *BFS* and *RandomWalk*. Different search strategies exercise the service under test differently.

quences of up to length four is already more than 21 millions, and a naive brute-force enumeration of those would already be untractable.

Still, even with the two core techniques used in *RESTler*, the search space explodes quickly, and we evaluate other search strategies next.

4.4 Search Strategies

We now present results of experiments with the search strategies BFS, RandomWalk, and BFS-Fast defined in Section 3. We start by comparing BFS with RandomWalk.

Code Coverage. Table 2 shows the unique lines of code covered by BFS and RandomWalk, as well as their intersection (excluding service-boot coverage), after 1, 3, and 5 hours of search. The Table also shows the maximum length of request sequences generated so far and, for RandomWalk, the total number of restarts after 5 hours in parenthesis.

We observe that both search strategies quickly converge to an overlapping set of lines of code. After 5 hours of search, BFS often generates the best coverage. However, in the Issues and Groups APIs, there are still some lines of code covered only by BFS or by RandomWalk.

By construction, BFS provides full-grammar coverage whenever it increases its sequence length, and this feature seems to pay off when measuring service-side coverage. In contrast, RandomWalk goes deeper faster but without providing full-grammar coverage. The effects on coverage of these two different search strategies is more visible on broader APIs and search spaces with more request types, like the Issues and Groups APIs (see Table 1). After 1 hour of search for these two APIs, RandomWalk has a head-start of, respectively, 80 and 31 lines of code in coverage compared to BFS. However, after 5 hours of search, BFS coverage has caught up with RandomWalk and is now overall better.

HTTP Status Codes. Figure 6 shows the distribution of HTTP status codes collected during 5 hours of testing of each of the GitLab APIs with BFS and RandomWalk. Focusing on each API group separately, we can see that the distributions obtained with BFS and RandomWalk are different. Both search strategies seem to exercise the service-under test differently. For instance, in the Commits, Issues and Groups APIs, only RandomWalk triggers `404` status codes. Yet in the Commits case, BFS's coverage is strictly better (see Table 2). Another interesting difference is that, in the Commits and Repos APIs, RandomWalk triggers at least three times more `500` "Internal Server Error" status codes than BFS. We use those `500` status codes to detect bugs, and we will discuss those more in Sections 4.5 and 5.

Overall, the differences in coverage and HTTP status-code distribution profiles obtained with BFS and RandomWalk motivate us to explore trade-offs between these two extreme search strategies. Specifically, we now evaluate such a trade-off, namely BFS-Fast.

Comparison with BFS-Fast. Like RandomWalk, BFS-Fast goes deeper faster than BFS. But like BFS, BFS-Fast still provides full-grammar coverage when increasing sequence length.

API	Time (hours)	BFS-Fast			BFS		
		Len.	Cov.	seqSet	Len.	Cov.	seqSet
Commits	1	10	1482	-	4	1485	-
	3	14	1482	-	4	1486	-
	5	17	1482	6	4	1486	2153
Branches	1	22	974	-	6	988	-
	3	42	974	-	6	988	-
	5	42	974	6	6	1026	5652
Issues	1	4	1100	-	3	1020	-
	3	5	1150	-	3	1155	-
	5	5	1150	272	3	1158	6964
Repos	1	12	935	-	5	1030	-
	3	18	972	-	5	1030	-
	5	23	986	13	5	1030	909
Groups	1	13	772	-	3	754	-
	3	19	772	-	3	760	-
	5	21	772	31	3	798	13421

Table 3: **Comparison of *BFS-Fast* and *BFS* over Time**. Shows the maximum sequence length and the increase in lines of code covered (excluding service-boot coverage) obtained with each search strategy after 1, 3, and 5 hours. The `seqSet` size is also shown after 5 hours. Although *BFS* covers slightly more lines of code, *BFS-Fast* reaches deeper request sequences and maintains a much smaller `seqSet` size.

Table 3 presents results of experiments comparing BFS-Fast with BFS. The Table shows the maximum sequence length and the increase in lines of code covered (excluding service-boot coverage) by each search strategy after 1, 3, and 5 hours. The `seqSet` size is also shown after 5 hours.

We can see that, although BFS covers slightly more lines of code, BFS-Fast reaches deeper request sequences and maintains a much smaller `seqSet` size. With BFS-Fast, the set `seqSet` remains consistently small: at each iteration when the sequence length increases, the function EXTEND only creates one new sequence per request type; then this set is in turn expanded by the function RENDER, but then shrinks again as many fuzzing combinations lead to invalid responses. This explains why `seqSet` tends to oscillate and stabilize around small sizes as the search goes deeper.

In practice, controling the size of `seqSet` is key to *scalability* when reaching greater depths during longer searches, or when facing broader search spaces due to larger APIs with more request types. This is why we adopt BFS-Fast as the default search strategy used in *REST-ler*.

Although code coverage with BFS-Fast is slightly less than with BFS after 5 hours of search, BFS-Fast actually detects all the `500` HTTP status codes found by BFS within 5 hours, as well as those found by RandomWalk, as discussed in the next section.

4.5 Bug Bucketization

Before discussing real errors found with *REST-ler*, we introduce a bucketization scheme to cluster the numerous `500` "Internal Server Errors" that can sometimes be reported. Indeed, as usual when fuzzing, different instances of a same bug can be repeatedly found over and over again. Since all the bugs found have to be inspected by the user, it is therefore important in practice to facilitate this analysis by identifying likely-redundant instances of a same unique bug.

In our context, we define a *bug* as a `500` HTTP status code being received after executing a request sequence. Thus, every bug found is associated with the request sequence that was executed to find it. Given this property, we use the following bucketization procedure for the bugs found by *REST-ler*:

> Whenever a new bug is found during the search, we compute all non-empty suffixes of its request sequence (starting with the smallest one), and we check whether some suffix is a previously-recorded request sequence leading to a bug found earlier during the search. If there is a match, the new bug is added to the bucket of that previous bug. Otherwise, a new bucket is created with the new bug and its request sequence.

Note that, in the above procedure, requests in request sequences are identified by their type, not by how they are rendered – fuzzable pritimitive types are not taken into account, and requests rendered differently are always considered equivalent. For a request sequence of length n, there are n suffixes. When using BFS or BFS-Fast, this bucketization scheme will identify bugs by the shortest request sequence needed to find it.

For the 5-hours experiments with GitLab reported earlier in this section, *REST-ler* found bugs (`500` HTTP status codes) in two out of the five GitLab API groups (see Figure 6). After bucketization, there are two buckets found for the Commits API, two buckets found for the Repos API, and none for the Branches, Issues, and Groups APIs. Note that all four buckets are found within 5 hours by BFS, RandomWalk, and BFS-Fast.

Figure 7 depicts the (unrendered) request sequence of length 2 identifying one of the bug buckets found for the Commits API. This request sequence triggers a `500` "Internal Server Error" when a user creates a project and then posts a commit with an empty branch name – that is, when the second request is rendered with its branch field being the empty string. This specific error can be reached by numerous longer sequences (*e.g.,* creating two projects and posting a commit with an empty branch name) and multiple value renderings (*e.g.,* any

```
1/2: POST /api/v4/projects HTTP/1.1
Accept: application/json
Content−Type: application/json
Host: 127.0.0.1
PRIVATE−TOKEN: [FILTERED]
'{"name":restler_fuzzable("string")}'

2/2: POST /api/v4/projects/projectid/repository/commits HTTP/1.1
Accept: application/json
Content−Type: application/json
Host: 127.0.0.1
PRIVATE−TOKEN: [FILTERED]
'{
    "branch": restler_fuzzable("string"),
    "commit_message": restler_fuzzable("string"),
    "actions":
    [
        {
            "action": ["create", "delete", "move", "update"],
            "file_path": restler_fuzzable("string"),
            "content": restler_fuzzable("string"),
        }
    ]
}'
```

Fig. 7: **Sample *REST-ler* Bug Bucket.** This request sequence identifies a unique bug found in GitLab's Commits API. The sequence consists of two requests: the first POST request creates a project, and the second POST request uses the id of that new project and posts a commit with some action and some file path.

feasible "content" rendering along with an empty branch name). Nevertheless, the above bucketization procedure will group together all those variants into the same bucket.

We discuss the details causing this error as well as other unique bugs in the following section. Indeed, *REST-ler* also found unique bugs in the Groups, Branches, and Issues GitLab APIs when running longer fuzzing experiments.

5 New Bugs Found in GitLab

In this section we discuss "unique bugs" (*i.e.,* after bucketization) found so far by *REST-ler* during experiments with the GitLab APIs. Note that *REST-ler* does not report false alarms and that all unique bugs were so far rather easily reproducible, unless otherwise specified. All the bugs reported in this section were found by *REST-ler* while running for at most 24 hours.

Early in our experiments, *REST-ler* found a first bug in the Branches API when using a fuzzing dictionary which included a (1-byte) "\0" string as one of its values. The bug seems due to a parsing issue in `ruby-grape`, a `ruby` middleware library for creating REST APIs [30]. Since other GitLab APIs also depend on the `ruby-grape` library, the bug is easy to trigger when using a dictionary with a "\0" string. In other words, many bugs in different buckets as defined in the previous section point to that same common root cause. In order to eliminate this "noise" in our experiments, we subsequently removed that dictionary value for the experiments of Section 4 and for those that led to discover the bugs discussed below. We now describe additional bugs found per API group.

Commits. In the Commits API, *REST-ler* found three bugs. The first one is triggered by the request sequence shown in Figure 7 when the branch name in the second request is set to an empty string. According to GitLab's documentation [12], users can post commits with multiple files and actions, including action "create", which creates a file with a given content on a selected target branch. For performance benefits, GitLab uses `rugged` bindings to native `libgit2`, a pure C implementation of the Git core methods [22]. Due to incomplete input validation, an invalid branch name, like an empty string, can be passed between the two different layers of abstraction as follows. The ruby code checks if the target branch exists by invoking a native C function whose return value is expected to be either NULL or an existing entry. However, if an unmatched entry type (*e.g.,* an empty string) is passed to the C function, an exception is raised. Yet, this exception is unexpected/unhandled by the higher-level ruby code, and thus causes a `500` "Internal Server Error". The bug can easily be reproduced by creating a project and posting a commit with action "create" to a branch whose name is set to the empty string.

A second bug found by *REST-ler* seems similar: (1) create a project, (2) post a valid commit with action "create", and then (3) cherry-pick the commit to a branch whose name is set to the empty string. Both bugs seem due to similar improper validation of branch names.

A third bug found by *REST-ler* is slightly different: (1) create a project, and then (2) create a commit with action "move" and omit the "previous_path" field from the action's parameters. In this case, the bug is due to an incomplete validation of the "previous_path" field. We do not know how severe this specific bug is, but there have been past security vulnerabilities in GitLab due to improper file-paths validation, which could have been exploited by an attacker to perform unauthorized move operations and leak private data [16].

Repos. In the Repos API, *REST-ler* found two bugs. The first one is triggered when a user attempts to (1) create a project, and then (2) create a file using a dictionary of parameters in which either the author's email or the author's name (optional parameters) is set to the empty string. The bug seems due to incomplete input validation and error propagation between different layers of abstraction. Specifically, ruby code invokes native code from `libgit2` which attempts to parse commit options

and create a commit signature, using the author's name and email. In case an empty string is encountered in the author's name of email field, an unexpected/unhandled exception is raised to the ruby code causing a `500` "Internal Server Error".

The second bug is similar, but requires a deeper sequence of actions in order to be triggered: (1) create a project, (2) create a file, and (3) edit the file content with an empty author email or author name. Note that both bugs seem to be due to the same piece of code.

Groups. For the Groups API, *REST-ler* found a bug which can be triggered by creating a new group using an invalid parent group id. According to GitLab's documentation, users can create new groups and optionally define group nesting by providing a parent group id through a parameter named "parent_id". If group nesting is defined, the ACLs of the new group must be at least as restricted as those of the parent. However, the ruby code responsible for enforcing such a relationship between ACLs uses "parent_id" to access the respective ACLs without checking whether "parent_id" lies within a range of valid integers for `activerecord`. Therefore, using a large integer value in the field of the "parent_id" param causes a `RangeError` exception. Furthermore, after manual inspection, we realized that assigning to "parent_id" any integer value which is currently not assigned to an existing group will also cause a `NilClass` excetion. This is because there is always an attempt to access the ACLs of an object without first checking that the target object actually exists.

Branches and Issues. Finally, for both the Branches and the Issues APIs *REST-ler* found `500` "Internal Server Errors" when running in parallel fuzzing mode, where multiple request sequences are being tested concurrently. For instance, one such bug can be reproduced as follows: (1) create a project, (2) create a valid branch and *at the same time* (*e.g.,* using a second process) create another branch on the same project. To successfully reproduce this bug, one needs to make the two requests simultaneously in order for both of them to be within a small time window in which some internal book-keeping is being performed. These bugs seem due to concurrency issues and may be related to the attacks on database-backed web applications presented by Warszawski et al. [39].

Discussion. From the description of the bugs found so far with *REST-ler* we see an emerging two-fold pattern. First, *REST-ler* produces a sequence of requests that exercise a service deep enough so that it reaches a particular interesting and valid "state". Second, while the service under test is in such a state, *REST-ler* produces an additional request with an unexpected fuzzed primitive type – like a string containing "\0" or an empty string. Therefore, triggering a bug requires a combination of these two features. We believe that *REST-ler* already does a good job at exercising a service deep enough, thanks to its dependency analysis. However, more fuzzing values and more properties to check in responses could boost *REST-ler*'s bug-finding capabilities – there is clearly room for improvement here.

We emphasize that the findings reported in this section are still preliminary. The bugs found so far are input validation bugs whose severity could range from mere crashes (which might only cause Denial-Of-Service attacks in the worst case) to more subtle access-control bugs (which might cause service state corruption or private-information disclosure). At the time of this writing, we are in communication with the GitLab developers, and we will update the next version of this paper with their feedback.

6 Related Work

The lightweight static analysis of Swagger specifications done by *REST-ler* (see Section 2) in order to infer dependencies among request types is similar to the analysis of type dependencies performed by the Randoop algorithm [27] for typed object-oriented programs. However, unlike in the Randoop work, dynamic objects in Swagger specifications are untyped, and *REST-ler* has to infer those somehow, as best as it can. Sometimes, user help may be required to unblock *REST-ler* when a Swagger specification is not complete, for instance, when a type of resource is not described in the Swagger specification itself, such as authentication and authorization tokens for allowing a test client to access a service. In the future, it would be interesting to allow users to annotate Swagger specifications and to declare service-specific types as well as their properties, in the spirit of code contracts [24, 4].

The dynamic feedback *REST-ler* uses to prune invalid responses from the search space (see line 32 in Figure 4) is also similar to the feedback used in Randoop [27]. However, the Randoop search strategy (in particular, search pruning and ordering) is different from the three simple strategies considered in our work, namely BFS, BFS-Fast and RandomWalk (the latter being the closest to Randoop). Moreover, some of the optimizations of the Randoop search algorithm (related to object equality and filtering) are not directly applicable and relevant in our context. Of course, other search strategies could be used, and it would be worth exploring those in our context in future work.

Our BFS-Fast search strategy is inspired by test generation algorithms used in model-based testing [38], whose goal is to generate a minimum number of tests covering, say, every state and transition of a finite-state machine model (*e.g.,* see [40]) in order to generate a test suite to check conformance of a (blackbox) implementation with

respect to that model. It is also related to algorithms for generating tests from an input grammar while covering all its production rules [21]. Indeed, in our context, BFS-Fast provides, by construction, a full grammar coverage up to the given current sequence length. The number of request sequences it generates is not necessarily minimal, but that number was always small, hence manageable, in our experiments so far.

Since REST API requests and responses are transmitted over HTTP, HTTP-fuzzers can be used to fuzz REST APIs. Such fuzzers, like Burp [7], Sulley [34], Boo-Fuzz [6], or the commercial AppSpider [3] or Qualys's WAS [29], can capture/replay HTTP traffic, parse HTTP requests/responses and their contents (like embedded JSON data), and then fuzz those, using either pre-defined heuristics [3, 29] or user-defined rules [34, 6]. Tools to capture, parse, fuzz, and replay HTTP traffic have recently been extended to leverage Swagger specifications in order to parse HTTP requests and guide their fuzzing [3, 29, 37, 2]. Compared to those tools, the main originality of *REST-ler* is its global dependency analysis of Swagger specifications and its ability to intelligently generate sequences of requests without pre-recorded HTTP traffic.

General-purpose (*i.e.,* non-Swagger specific) grammar-based fuzzers, like Peach [28] and SPIKE [32], among others [35], can also be used to fuzz REST APIs. With these tools, the user directly specifies an input grammar, often encoded directly by code specifying what and how to fuzz, similar to the code shown on the right of Figure 2. Compared to those tools, *REST-ler* generates automatically an input grammar from a Swagger specification, and its fuzzing rules are determined separatety and automatically by the algorithm of Figure 4.

How to learn automatically input grammars from input samples is another complementary research area [20, 5, 19]. *REST-ler* currently relies on a Swagger specification to represent a service's input space, and it learns automatically how to prune invalid request sequences by analyzing service responses at specific states. Still, a Swagger specification could be further refined given representative (unit tests) or live traffic in order to focus the search towards specific areas of the input space. For services with REST APIs but no Swagger specification, it would be worth investigating how to infer it automatically from runtime traffic logs using machine learning, or by a static analysis of the code implementing the API.

Grammar-based fuzzing can also be combined [23, 17] with whitebox fuzzing [18], which uses dynamic symbolic execution, constraint generation and solving in order to generate new tests exercising new code paths. In contrast, *REST-ler* is currently purely blackbox: the inner workings of the service under test are invisible to *REST-ler* which only sees REST API requests and responses. Since cloud services are usually complex distributed systems whose components are written in different languages, general symbolic-execution-based approaches seem problematic, but it would be worth exploring this option further. For instance, in the short term, *REST-ler* could be extended to take into account alerts (*e.g.,* assertion violations) reported in back-end logs in order to increase chances of finding interesting bugs and correlating them to specific request sequences.

The GitLab concurrency-related bugs we reported in the previous section look similar to known classes of bugs and attacks in database-backed applications [39]. These bugs are due to concurrent database operations that are not properly encapsulated in serializable transactions. When a REST API exposes concurrent database operations without properly isolating those, data corruptions and leaks may happen. It would be interesting to develop concurrency-specific fuzzing rules inspired by the attack techniques of [39] and to experiment with those in *REST-ler*.

In practice, the main technique used today to ensure the security of cloud services is so-called "penetration testing", or *pen testing* for short, which means security experts review the architecture, design and code of cloud services from a security perspective. Since pen testing is labor intensive, it is expensive and limited in scope and depth. Fuzzing tools like *REST-ler* can partly automate and improve the discovery of specific classes of security vulnerabilities, and are complementary to pen testing.

7 Conclusions

We introduced *REST-ler*, the first automatic intelligent tool for fuzzing cloud services through their REST APIs. *REST-ler* analyzes a Swagger specification of a REST API, and generates tests intelligently by inferring dependencies among request types and by learning invalid request combinations from the service's responses. We presented empirical evidence showing that these techniques are necessary to thoroughly exercise a service while pruning its large search space of possible request sequences. We also evaluated three different search strategies on GitLab, a large popular open-source self-hosted Git service. Although *REST-ler* is still an early prototype, it was already able to find several new bugs in GitLab, including security-related ones.

While still preliminary, our results are encouraging. How general are these results? To find out, we need to fuzz more services through their REST APIs, add more fuzzing rules to further confuse and trip service APIs, and check more properties to detect different kinds of bugs and security vulnerabilities. Indeed, unlike buffer overflows in binary-format parsers, or use-after-free bugs

in web browsers, or cross-site-scripting attacks in webpages, it is still largely unclear what security vulnerabilities might hide behind REST APIs. While past human-intensive pen testing efforts targeting cloud services provide evidence that such vulnerabilities do exist, this evidence is still too anecdotical, and new automated tools, like *REST-ler*, are needed for more systematic answers. How many bugs can be found by fuzzing REST APIs? How security-critical will they be? This paper provides a clear path forward to answer these questions.

References

[1] S. Allamaraju. *RESTful Web Services Cookbook*. O'Reilly, 2010.

[2] APIFuzzer. `https://github.com/KissPeter/APIFuzzer`. Accessed: 4/2/2018.

[3] AppSpider. `https://www.rapid7.com/products/appspider`. Accessed: 4/2/2018.

[4] M. Barnett, M. Fahndrich, and F. Logozzo. Embedded Contract Languages. In *Proceedings of SAC-OOPS'2010*. ACM, March 2010.

[5] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing Program Input Grammars. In *Proceedings of PLDI'2017*, pages 95–110. ACM, 2017.

[6] BooFuzz. `https://github.com/jtpereyda/boofuzz`. Accessed: 4/2/2018.

[7] Burp Suite. `https://portswigger.net/burp`. Accessed: 4/2/2018.

[8] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The Combinatorial Design Approach to Automatic Test Generation. *IEEE Software*, 13(5), 1996.

[9] R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD Thesis, UC Irvine, 2000.

[10] Flask. Web development, one drop at a time. `http://flask.pocoo.org/`. Accessed: 2/22/2018.

[11] GitLab. GitLab. `https://about.gitlab.com`. Accessed: 2/22/2018.

[12] GitLab. GitLab API. `https://docs.gitlab.com/ee/api/`. Accessed: 3/16/2018.

[13] GitLab. Hardware requirements. `https://docs.gitlab.com/ce/install/requirements.html`. Accessed: 2/22/2018.

[14] GitLab. Statistics. `https://about.gitlab.com/is-it-any-good/`. Accessed: 2/22/2018.

[15] GitLab. Swagger OpenAPI specification. `https://axil.gitlab.io/swaggerapi/`. Accessed: 2/22/2018.

[16] GitLab CVEs. Past Security Vulnerabilities in GitLab. `https://about.gitlab.com/2013/11/14/multiple-critical-vulnerabilities-in-gitlab/`. Accessed: 4/10/2018.

[17] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of PLDI'2008*, pages 206–215, 2008.

[18] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008*, pages 151–166, 2008.

[19] P. Godefroid, H. Peleg, and R. Singh. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of ASE'2017*, pages 50–59, 2017.

[20] M. Höschele and A. Zeller. Mining Input Grammars from Dynamic Taints. In *Proceedings of ASE'2016*, pages 720–725, 2016.

[21] R. Lämmel and W. Schulte. Controllable Combinatorial Coverage in Grammar-Based Testing. In *Proceedings of TestCom'2006*, 2006.

[22] libgit. libgit2. `https://github.com/libgit2`. Accessed: 4/10/2018.

[23] R. Majumdar and R. Xu. Directed Test Generation using Symbolic Grammars. In *Proceedings of ASE'2007*, 2007.

[24] B. Meyer. *Eiffel*. Prentice-Hall, 1992.

[25] S. Newman. *Building Microservices*. O'Reilly, 2015.

[26] OAuth. OAuth 2.0. `https://oauth.net/`. Accessed: 4/5/2018.

[27] C. Pacheco, S. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *Proceedings of ICSE'2007*. ACM, 2007.

[28] Peach Fuzzer. `http://www.peachfuzzer.com/`. Accessed: 4/2/2018.

[29] Qualys Web Application Scanning (WAS). `https://www.qualys.com/apps/web-app-scanning/`. Accessed: 4/2/2018.

[30] Ruby Grape. An opinionated framework for creating REST-like APIs in Ruby. `http://www.ruby-grape.org/`. Accessed: 4/14/2018.

[31] Ruby on Rails. Rails. `http://rubyonrails.org`. Accessed: 2/22/2018.

[32] SPIKE Fuzzer. `http://resources.infosecinstitute.com/fuzzer-automation-with-spike/`. Accessed: 4/2/2018.

[33] SQLite. SQLite. `https://www.sqlite.org/index.html`. Accessed: 2/22/2018.

[34] Sulley. `https://github.com/OpenRCE/sulley`. Accessed: 4/2/2018.

[35] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[36] Swagger. `https://swagger.io/`. Accessed: 4/2/2018.

[37] TnT-Fuzzer. `https://github.com/Teebytes/TnT-Fuzzer`. Accessed: 4/2/2018.

[38] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-Based Testing Approaches. *Intl. Journal on Software Testing, Verification and Reliability*, 22(5), 2012.

[39] T. Warszawski and P. Bailis. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of SIGMOD'2017*, 2017.

[40] M. Yannakakis and D. Lee. Testing Finite-State Machines. In *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing*, pages 476–485, 1991.