2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

RESTler: Stateful REST API Fuzzing

Vaggelis Atlidakis*
Columbia University

Patrice Godefroid
Microsoft Research

Marina Polishchuk
Microsoft Research

***Abstract*—This paper introduces *RESTler*, the first stateful REST API fuzzer. *RESTler* analyzes the API specification of a cloud service and generates sequences of requests that automatically test the service through its API. *RESTler* generates test sequences by (1) inferring *producer-consumer dependencies* among *request types* declared in the specification (*e.g.,* inferring that "a request B should be executed after request A" because B takes as an input a resource-id x produced 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 present experimental results showing that these two techniques are necessary to thoroughly exercise a service under test while pruning the large search space of possible request sequences. We used *RESTler* to test GitLab, an open-source Git service, as well as several Microsoft Azure and Office365 cloud services. *RESTler* found 28 bugs in GitLab and several bugs in each of the Azure and Office365 cloud services tested so far. These bugs have been confirmed and fixed by the service owners.

I. Introduction

Over the last decade, we have seen an explosion in cloud services for hosting software applications (Software-as-a-Service), for building distributed services and 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 (AWS) [2] and Microsoft Azure [29], are programmatically accessed through REST APIs [11] by third-party applications [1] and other services [31]. Meanwhile, Swagger (recently renamed OpenAPI) [40] 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, what responses may be received, and the response 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 [4], [34], [7], [41], [3]. Many of these tools were born as extensions of more established website testing and scanning tools (see Section VIII). Since these REST API testing tools are all recent and not yet widely used, it is still largely unknown how effective they are in finding bugs and how security-critical those bugs are.

In this paper, we introduce *RESTler*, the first automatic stateful REST API fuzzing tool. Fuzzing [39] means automatic test generation and execution with the goal of finding security vulnerabilities. Unlike other REST API testing tools, *RESTler* performs a lightweight static analysis of an entire Swagger specification, and then generates and executes tests that exercise the corresponding cloud service in a stateful manner. By *stateful*, we mean that *RESTler* attempts to explore service states that are reachable only using sequences of multiple requests. With *RESTler*, each test is defined as a sequence of requests and responses. *RESTler* generates tests 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 showing that these two techniques are necessary to thoroughly test a service, while pruning the large search space defined by all possible request sequences. *RESTler* also implements several search strategies (akin to those used in model-based testing [43]) and we compare their effectiveness while fuzzing GitLab [13], an open-source self-hosted Git service with a complex REST API.

During the course of our experiments, we found 28 new bugs in GitLab (see Section VI). We also ran experiments on four public cloud services in Microsoft Azure [29] and Office365 [30] and found several bugs in each service tested (see Section VII). This paper makes the following contributions:

- We introduce *RESTler*, the first automatic, stateful fuzzing tool for REST APIs, which analyzes a Swagger specification, automatically infers dependencies among request types, and dynamically generates tests guided by feedback from service responses.
- We present detailed experimental evidence showing that the techniques used in *RESTler* are necessary for effective automated stateful REST API fuzzing.
- We 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.
- We discuss preliminary experiences using *RESTler* to test

*The work of this author was mostly done at Microsoft Research.

DOI 10.1109/ICSE.2019.00083

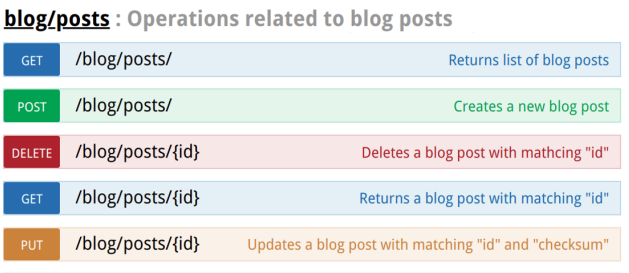

Fig. 1: **Swagger Specification of Blog Posts Service**

several Microsoft public cloud services.

The remainder of the paper is organized as follows. Section II describes how Swagger specifications are processed by *RESTler*. Sections III and IV present the main test-generation algorithm used in *RESTler* and implementation details. Section V presents an evaluation of the test-generation techniques and search strategies implemented in *RESTler*. Section VI discusses new bugs found in GitLab. Section VII presents our experiences fuzzing several public cloud services. Section VIII discusses related work, and Section IX concludes the paper.

II. Processing API Specifications

In this paper, we consider services accessible through REST APIs described with a Swagger specification. 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. A Swagger specification describes how to access a service through its REST API (*e.g.,* what requests the service can handle and what responses may be expected). 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.

A sample Swagger specification, in web-UI form, is shown in Figure 1. This specification describes the API of a simple blog posts hosting service. The API consists of five *request types*, specifying the endpoint, method, and required parameters. 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 (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 the specification describes the HTTP-syntax for this POST request as well as the format of the expected response.

From such a specification, *RESTler* automatically constructs the test-generation grammar shown on the right of Figure 2. This grammar is encoded in executable `python` code. It consists of code to generate an HTTP request, of type POST in this case, and code to process the expected response of this request. Each function `restler_static` simply appends the string it takes as argument without modifying

```
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 *RESTler* Grammar.** Shows a snippet of Swagger specification in YAML (left) and the corresponding grammar generated by *RESTler* (right).

it. In contrast, the function `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 response is expected to return a new *dynamic object* (a dynamically created resource id) named `id` of type `integer`. Using the `schema` shown on the left, *RESTler* automatically generates the function `parse_posts` shown on the right.

By similarly analyzing the other request types described in this Swagger specification, *RESTler* will infer automatically that `ids` 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 *producer-consumer dependencies* are extracted by *RESTler* when processing the Swagger specification and are later used for test generation, as described next.

III. Test Generation Algorithm

The main algorithm for test generation used by *RESTler* is shown in Figure 3 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 3, and of their dependencies (more on this later).

The algorithm computes a set of request sequences, as inferred from Swagger, denoted `seqSet` and initially containing an empty sequence ϵ (line 5). A request sequence is *valid* if every response in the sequence has a valid return code, defined here as any code in the `200` range. At each iteration of its main loop (line 8), starting with $n = 1$, the algorithm computes all valid request sequences `seqSet` of length n 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 9) to create a set of new sequences of length n

```
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 an empty sequence ϵ)
5  seqSet = {ϵ}
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. 3: **Main Algorithm used in *RESTler*.**

by appending each request with satisfied dependencies at the end of each sequence, as described in the EXTEND function (line 14). The function DEPENDENCIES (line 39) checks if all dependencies of the specified request are *satisfied*. This is true when every dynamic object that is a required parameter of the request, denoted by `CONSUMES(req)`, is produced by some response to 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 *concretized*, which substitutes one concrete value of that type taken out of a finite, user-configurable *dictionary* of values. For instance, for fuzzable type `integer`, *RESTler* 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 a very long fixed string. The function RENDER generates all possible such combinations (line 28). Each combination thus corresponds to 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 status code, the new request sequence is valid and retained (line 33); otherwise, it is discarded and the received error code is logged for analysis and debugging.

More precisely, the function EXECUTE executes each request in a sequence 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, *i.e.,* it becomes unusable later on, *RESTler* will detect this by receiving an invalid status code (outside the `200` range) when attempting to reuse that unusable object, and will then discard that request sequence.

By default, the function RENDER of Figure 3 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 [10] 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 3.

The function EXTEND of Figure 3 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 V, we report experiments performed also with two additional search strategies: BFS-Fast and RandomWalk.

BFS-Fast. In function EXTEND, instead of appending every request to every sequence, every request is appended to at most one sequence. This results in 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 still exercises every executable request type at each iteration of the main loop in line 8: it still provides full grammar coverage but with fewer request sequences, which allows it to go deeper faster 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.)

IV. Implementation

We have implemented *RESTler* in 3,151 lines of modular python code split into: the parser and compiler module, the core fuzzing runtime module, and the garbage collector (GC) module. The parser and compiler module is used to parse a Swagger specification and to generate the *RESTler* grammar describing how to fuzz a target service. (In the absence of a Swagger specification, the user could directly provide the *RESTler* grammar.) The core fuzzing runtime module implements the algorithm of Figure 3 and its variants. It renders API requests, processes service-side responses to retrieve values of the dynamic objects created, and analyzes service-side feedback to decide which requests should be reused in future generations while composing new request sequences. Finally, the GC runs as a separate thread that tracks the creation of the dynamic objects over time and periodically deletes aging objects that exceed some user-defined limit (see Section VII).

A. *Using* RESTler

RESTler is a command-line tool that takes as input a Swagger specification, service access parameters (e.g. IP, port, authentication), the mutations dictionary, and the search strategy to use during fuzzing. After compiling the Swagger specification, RESTler displays the number of endpoints discovered and the list of resolved and unresolved dependencies, if any. In case of unresolved dependencies, the user may provide additional annotations or resource-specific mutations (see Section VII) and re-run this step to resolve them. Alternatively, the user may choose to start fuzzing right away and *RESTler* will treat unresolved dependencies in consumer parameters as `restler_fuzzable` string primitives. During fuzzing, *RESTler* reports each *bug*, currently defined as a 500 HTTP status code (`500` "Internal Server Error") received after executing a request sequence, as soon as it is found.

B. Current Limitations

Currently, *RESTler* does not support requests for API endpoints with server-side redirects (*e.g.,* `301` "Moved Permanently", `303` "See Other", and `307` "Temporary Redirect"). Furthermore, *RESTler* currently can only find bugs defined as unexpected HTTP status codes. 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, *RESTler* has already found confirmed bugs in a production-scale open-source application and in several Microsoft Azure and Office365 services, as will be discussed in Sections VI and VII.

V. Evaluation

We present experimental results obtained with *RESTler* 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 V-B)
- **Q2:** Are tests generated by *RESTler* exercising deeper service-side logic as sequence length increases? (Section V-C)
- **Q3:** How do the three search strategies implemented in *RESTler* compare across various APIs? (Section V-D)

We answer the first question (Q1) using a simple 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 V-E how to bucketize (*i.e.,* group together) the numerous bugs that can be reported by *RESTler* in order to facilitate their analysis.

A. 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 [12]. Its functionality is exposed over a REST API with a Swagger specification shown in Figure 1. The 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`: 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 is written in over 376K lines of ruby code using ruby-on-rails [35] and its functionality is exposed through a REST

[1] GitLab [13] 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 [20].

API [14]. For our deployment, we apply the following configuration settings: we use Nginx to proxypass the Unicorn web server and configure 15 Unicorn workers limited to up to 2GB 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 redis workers. According to GitLab's deployment recommendations, such configuration should scale up to 4,000 concurrent users [15].

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 56GB of physical memory.

B. Techniques for Effective REST API Fuzzing

In this section, we report 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 simple 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) *RESTler* 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) *RESTler* ignores service-side dynamic feedback and does not eliminate invalid sequences during the search, but still infers dependencies among request types and generates request sequences satisfying those.
3) *RESTler* uses the algorithm of Figure 3 using both dependencies among request types and dynamic feedback.

Figure 4 shows the number of tests, *i.e.,* request sequences, up to maximum length 3, generated by each of these three algorithms, from left to right. The top plots show the cumulative code coverage measured in lines of code over time, as well as when the sequence length increases. The bottom plots show the cumulative number of HTTP status codes received.

Code Coverage. First, we observe that without considering dependencies among request types (Figure 4, 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 illustrates the limitations of using a naive approach to 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. In contrast, by infering dependencies among requests and by processing service responses *RESTler* achieves an increase in code coverage up to 150 lines of code (Figure 4, top center and right).

Second, we see that without considering dynamic feedback to prune invalid request sequences in the search space (Figure 4, top center) the number of tests generated grows quickly, even for a simple API. Specifically, without considering dynamic feedback (Figure 4, top center), *RESTler* produces more than $4,600$ tests that take $1,750$ seconds and cover about 150 lines of code. In contrast, by considering dynamic feedback (Figure 4, top right), the state space is significantly reduced and *RESTler* achieves 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 4, bottom center). This indicates that 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% without using dependencies among request types (Figure 4 bottom left) and to 20% with using dependencies among request types (Figure 4, bottom right). Moreover, when using dependencies among request types (Figure 4, bottom right), we observe the highest percentage of `20X` status codes (approximately 80%), indicating that *RESTler* then exercises a larger part of the service logic – also confirmed by coverage data (Figure 4, top right).

Second, when ignoring dependencies among request types, we see that no `500` status codes are detected (Figure 4, bottom left), while *RESTler* finds a handful of `500` status codes when using dependencies among request types (see (Figure 4, 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 V-A). When ignoring dependencies among request types, *RESTler* misses this bug (Figure 4, bottom left). In contrast, when 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`, *RESTler* does trigger the bug. Furthermore, when additionally using dynamic feedback, the search space is pruned while preserving this bug, which is then found with the least number of tests (Figure 4, bottom right).

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

C. Deeper Service Exploration

In this section, we use GitLab to determine whether tests generated by *RESTler* exercise deeper service-side logic as sequence length increases (Q2). We perform individual experiments on six groups of GitLab APIs related to usual operations with commits, branches, issues and notes, repositories and repository files, groups and group membership, and projects.

Table I shows the total number of requests in each of the six target API groups and presents experimental results obtained

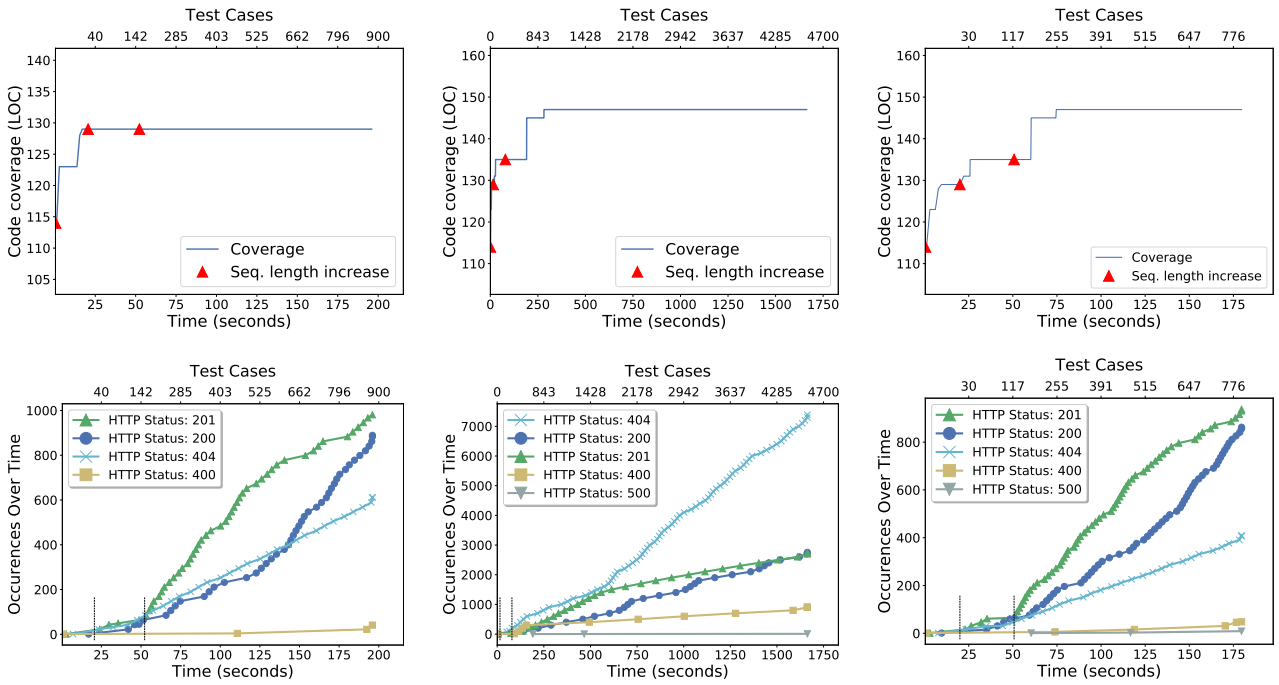

Fig. 4: **Blog Posts Service Code Coverage and HTTP Status Codes Over Time.** Shows the increase in code coverage over time (top) and the cumulative number of HTTP status codes received over time (bottom), for the simple blog posts service. *Left: RESTler* ignores dependencies among request types. *Center: RESTler* ignores dynamic feedback. *Right: RESTler* utilizes both dependencies among request types and dynamic feedback. When leveraging both techniques, *RESTler* achieves the best code coverage and finds the planted `500` "Internal Server Error" bug with the least number of tests.

API	Total Requests	Seq. Len.	Coverage Increase	Tests	seqSet Size	Dynamic Objects
Commits	11	1	598	1	1	1
		2	1108	7	5	10
		3	1196	250	46	521
		4	1760	2220	1341	6577
		5	1760	3667	20679	12518
Branches	7	1	598	1	1	1
		2	1089	8	6	11
		3	1172	58	44	107
		4	1182	576	387	1279
		5	1185	3644	5528	9336
Issues	22	1	816	37	37	37
		2	1163	2444	1839	4245
		3	1163	4156	15658	8870
Repos	10	1	598	1	1	1
		2	1117	97	65	206
		3	1181	5153	2194	15472
Groups	50	1	887	39	39	38
		2	1177	3508	3360	5204
		3	1177	4817	79518	8946
Projects	48	1	934	42	41	38
		2	1192	1870	1781	3343
		3	1203	3226	18173	7374

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

with the test-generation algorithm of Figure 3 using BFS. For each experiment we run *RESTler* with a 5-hours timeout and limit the number of fuzzable primitive-type 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 group, as time goes by, Table I shows the increase (going down) in the sequence length, code coverage, tests executed, `seqSet` size, and the number of dynamic objects 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 I shows the cumulative code coverage achieved after executing all the request sequences generated by *RESTler* for each sequence length, or until the 5-hours timeout expires. The results are incremental on top of 16,836 lines of code executed during service boot.

From Table I, we can see that longer sequence lengths consistently lead to increased service-side code coverage. This is the desired behaviour, 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 dependency 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 598 to $1,108$ to $1,196$ lines of code for sequence lengths of one,

API	Total Requests	Time (hrs)	BFS			BFS-Fast			RandomWalk		
			Len.	Coverage	seqSet	Len.	Coverage	seqSet	Len. (restarts)	Coverage	seqSet
Commits	11 (*11)	1	4	1202		7	1697		13 (16)	1285	
		3	5	1760		9	1731		13 (35)	1295	
		5	5	1760	20679	12	1731	33	13 (56)	1303	1
Branches	7 (*2)	1	5	1182		21	1154		15 (24)	1182	
		3	5	1185		37	1178		19 (92)	1187	
		5	5	1185	5528	47	1178	11	22 (158)	1208	1
Issues	22 (*82)	1	2	1150		2	1086		10 (1)	770	
		3	3	1163		4	1551		10 (1)	770	
		5	3	1163	15658	5	1570	26	16 (2)	847	1
Repos	10 (*24)	1	3	1127		5	1141		10 (29)	1195	
		3	3	1127		7	1141		13 (88)	1231	
		5	3	1181	2194	8	1161	64	13 (142)	1231	1
Groups	50 (*2)	1	2	961		6	1275		19 (41)	1167	
		3	3	1177		11	1275		19 (120)	1250	
		5	3	1177	79518	14	1275	130	22 (186)	1283	1
Projects	48 (*4)	1	2	1006		5	1318		4 (3)	889	
		3	2	1053		11	1319		22 (31)	1024	
		5	3	1203	18173	15	1319	171	22 (45)	1273	1

TABLE II: **Comparison of *BFS*, *BFS-Fast* and *RandomWalk* over Time**. Shows the maximum sequence length, the increase in lines of code covered (excluding service-boot coverage), and the `seqSet` size with each search strategy after 1, 3, and 5 hours. The second column shows the total number of requests in each API along with the average feasible request renderings (*). Although *BFS* covers slightly more lines of code, *BFS-Fast* and *RandomWalk* reach deeper request sequences and maintain a much smaller `seqSet` size.

two, and three, respectively. Most notably, for the Branches API, service-side code coverage keeps gradually increasing for sequences of length up to five, and reaches $1,185$ lines when the 5-hours limit expires.

Tests, Sequence Sets, and Dynamic Objects. In addition to code coverage, Table I also shows the increase in the number of tests executed, the size of `seqSet` after the RENDER function returns (line 10 of Figure 3), and the number of dynamic objects created by *RESTler*. All those numbers are quickly growing since the search space also grows quickly due to the exhaustive nature of the BFS search strategy.

Nevertheless, we emphasize that without the two key techniques evaluated in Section V-B this growth would be much worse. For instance, for the Commit API, the `SeqSet` size is $20,679$ and there are $12,518$ dynamic objects created by *RESTler* for sequences of length up to five. By comparison, since the Commits API has 11 request types with an average of 4 rendering combinations, the number of all possible rendered request sequences of up to length four is already more than 164 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.

D. Search Strategies

We now present results of experiments comparing the BFS, BFS-Fast, and RandomWalk search strategies defined in Section III (Q3). For each search strategy, Table II shows the maximum sequence length, the increase in lines of code covered (excluding service-boot coverage) after 1, 3, and 5 hours, and the size of the `seqSet` when the 5-hours timeout is reached. For the RandomWalk search strategy the total number of restarts is also shown in parenthesis.

First, we compare BFS with BFS-fast. We observe that after five hours, BFS achieves better coverage than BFS-Fast in Commits, Branches, and Repos. These groups of APIs have relatively fewer requests and BFS delivers better coverage by exercising all feasible request sequences. However, BFS does not scale well in APIs with relatively more requests, such as Issues, Groups, and Projects. As shown in Table II after 5 hours for Issues, Groups, and Projects, BFS is still exploring sequences of length 3 while BFS-Fast is exploring sequences of length 5, 14, and 15, respectively. BFS-Fast scales better in APIs with many request because, unlike BFS, it does not explore all feasible request sequences but instead appends each request to at most one sequence in each generation. BFS-Fast maintains a smaller `seqSet`, and explores deeper sequences and grows coverage faster in Issues, Groups, and Projects.

We now compare BFS with RandomWalk. By construction, RandomWalk does not guarantee full grammar coverage since it appends each request to one random sequence in each generation. As shown in Table II, RandomWalk maintains a small `seqSet` at any time, by construction. Furthermore, after 5 hours RandomWalk explores considerably deeper request sequences compared to BFS and, in most cases, compared to BFS-Fast. RandomWalk also delivers the best coverage after 5 hours in Branches, Repos, and Groups.

On the other hand, in Issues, we observe that after 5 hours RandomWalk explores sequences of length 16 and the coverage increase is 847 lines. In the same time-frame, BFS explores sequences of length 3 but the coverage increase is $1,163$ and BFS-Fast explores sequences length 5 with a

API	BFS	BFS-Fast	Random-Walk	Intersection	Union
Commits	5	1	5	1	5
Branches	7	7	7	5	8
Issues	0	1	1	0	1
Repos	2	3	3	2	3
Groups	0	0	2	0	2
Projects	2	1	3	1	3
Total	16	13	21	9	22

TABLE III: **Bug Buckets found by BFS, BFS-Fast, and RandomWalk after Five Hours.** Shows the sets of bugs found by each search strategy in each API. In total: *RESTler* found 22 new bugs.

coverage increase of $1,570$. Compared to all other APIs shown in Table II, Issues have 82 feasible request renderings on average. This is relatively large and in such a case, with many feasible request renderings, the breadth of the search achieved by RandomWalk is small (*e.g.,* after 5 hours there are only 2 restarts). Consequently, the search remains focused on a very restricted subspace which reflects poorly on coverage.

In practice, both controlling the size of `seqSet`, when facing broader search spaces due to large APIs with many requests or when reaching greater depths, and maintaining some breadth when extending request sequences seem key to delivering better code coverage. Nevertheless, the ultimate goal is to find bugs, and maximizing code coverage is just a heuristic to try to reach that goal.

E. Bug Bucketization

Before discussing real bugs found with *RESTler*, we introduce a bucketization scheme to cluster similar `500` "Internal Server Errors". When fuzzing, different instances of a same bug are often found repeatedly. 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 *RESTler*:

> Whenever a new bug is found, we compute all non-empty suffixes of its non-rendered request sequence[2] (starting with the smallest one) and check whether some suffix is a previously-recorded sequence leading to a bug found earlier. 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.

When using BFS or BFS-Fast, this bucketization scheme will identify bugs by the shortest sequence necessary to find it.

Table III shows the sets of bug buckets found by each search strategy, after five hours, in each GitLab API group. To demonstrate the overlap between the bugs reported by each method, the last two columns show the intersection and the union of the bug buckets. In the context of these experiments, *RESTler* found 22 new *unique* bugs, after running each search strategy for 5 hours on each API group.

RandomWalk stands out in Table III by finding the most bugs: 21 compared to 16 and 13 for BFS and BFS-Fast respectively. It is particularly intriguing that RandomWalk finds as many bugs as BFS and BFS-Fast combined in Commits and in Issues APIs because in these APIs RandomWalk delivers relatively little coverage. After 5 hours, in commits, RandomWalk finds as many bugs as BFS and more than BFS-Fast. At the same time, RandomWalk delivers less code coverage than each of BFS and BFS-Fast in Commits (see Table II). Similarly, RandomWalk finds 1 bug in Issues, while BFS finds none and BFS-Fast also finds one. Yet, again, RandomWalk achieves less code coverage than each of BFS and BFS-Fast in Issues. The differences between BFS and BFS-Fast are less striking. BFS finds more bugs in Commits, while BFS-Fast finds more bugs in Issues and Repos.

Overall, within the 5-hours time-frame of our experiments, RandomWalk finds more bugs than BFS or BFS-Fast despite the fact that it does not always deliver the best coverage. It is unclear how this generalizes to longer fuzzing sessions or to other APIs. Yet, it becomes apparent that coverage increase should not always dictate the selection of a search strategy because different search strategies may be complementary within a large search space. Next, we discuss details of the bugs founds with *RESTler* in GitLab and the total number of bugs found when running longer fuzzing experiments.

VI. New Bugs Found in GitLab

During all our fuzzing experiments with *RESTler* on our local GitLab deployment, we found a total of 28 new unique bugs. All bugs were easily reproducible, disclosed to GitLab developers, confirmed and fixed. Due to space limitation, we describe only 2 of these bugs, to give the reader a flavor of what those bugs look like and how they were found. (See [16], [17], [18], [19] for other examples of bugs found.)

Example 1: Bug in Commits API. One of the bugs found by *RESTler* in the Commits API is triggered when a user tries to cherry-pick a commit to a branch with an empty name. Due to incomplete input validation, an invalid branch name can be passed between two different layers of abstraction as follows: The ruby code that checks if a target branch exists, invokes 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 native function, an exception is raised. This exception is unhandled by the higher-level ruby code, and therefore it causes a `500` "Internal Server Error". The bug can be reproduced by (1) creating a project, (2) creating a new branch (in addition to master branch which is created by default), (3) posting a valid commit with action "create" in the branch created in (2), and (4) cherry-picking the commit to a branch whose name is set to the empty string.

Example 2: Bug in Branches API. Another bug, found by *RESTler* in the Branches API, is triggered when a user

[2]A request sequence of length n has n suffixes of length $1, 2, \ldots, n$.

tries to edit a branch of a recently deleted project. The bug is due to invalid serialization of operations which results in an database entry update using an invalid foreign key of a deleted project. Since the project-id (foreign key) is not present in the respective "projects" table, a *PG::ForeignKeyViolation* exception causes a `500` "Internal Server Error". The bug can be reproduced by (1) creating a project, (2) creating a branch, (3) deleting the project created in (1), and (4) quickly editing the branch of the deleted project.

From the above bug descriptions, we see a two-fold pattern. First, *RESTler* produces a sequence that exercises the target service deep enough so that it reaches a particular valid "state". Second, while the service is in such a state, *RESTler* produces an additional request with an unexpected fuzzed value (*e.g.,* an empty string) or an unexpected action (*e.g.,* edit a branch of a recently deleted project). Most bugs found by *RESTler* require a combination of these two features in order to be found.

VII. Experiences with Public Cloud Services

In this section, we describe our preliminary experiences running *RESTler* on three Azure [29] services and one Microsoft Office365 [30] service. The services we fuzzed primarily perform resource management and real-time data aggregation. Swagger specifications for these services are publicly available and published by Microsoft on GitHub.

While still in an early stage of development, *RESTler* found new bugs in all of these services. These bugs range from mis-handled invalid inputs (e.g., using a wrong ID or enum value), executing operations in invalid states (e.g., updating a resource that no longer exists), and inconsistent parameter validations (e.g., using a valid request body with incorrect metadata). Although we cannot disclose detailed descriptions of these bugs, we emphasize that all bugs found by *RESTler* so far have been confirmed and fixed by Microsoft service developers. Indeed, "500 Internal Server Errors" are server state corruptions that may severely damage service health and security: it is safer to fix these rather than risk a live incident with unknown consequences.

During this effort, we faced a number of challenges unique to public cloud services, including resource quota limitations, short-lived access tokens, and complex API dependencies beyond the canonical REST API structure with application-specific resource values and naming schemes. We describe the extensions made to *RESTler* to address these challenges.

Resource Quotas. Production services that run in public clouds are deployed with default resource quotas. Once quotas are reached, *RESTler*'s core algorithm will continue to try request sequences containing requests that can no longer succeed due to exceeded quotas (since these requests were valid in prior tests and generated lots of new resources), which impedes progress. This challenge is unique to public cloud deployments, contrary to self-contained deployments where one can easily control and reconfigure resource quotas. To address this problem, we implemented a garbage collector (GC) in *RESTler*. The GC runs as a separate thread that monitors the creation of dynamic resources over time and periodically deletes dynamic objects that are no longer used in order to avoid exceeding resource quotas. This allows *RESTler* to continuously test new sequences for hours or days without hitting resource-quota-related errors.

Short-lived Access Tokens. Unlike in self-contained deployments where an admin can pre-populate static or long-lived authentication tokens, public cloud services use short-lived, refreshable authentication tokens. Usually, a public endpoint, accessible with some type of static credentials (e.g., a username-password pair or a master token) and service-specific logic, generates fresh, short-lived access tokens. The latter are added in the header of HTTPS requests. Since different services may require custom logic to access their public authentication endpoints, *RESTler* provides an authentication hook which periodically executes a user-provided piece of code (e.g., a script) and propagates fresh values in the pool of refreshable authentication tokens.

Application-specific Naming Schemes. As discussed in Section II, *RESTler* performs a light-weight static analysis of a Swagger specification to infer dependencies among requests of the target REST API. However, part of a target API may not be fully REST compliant, or the specification may be incomplete, and consequently the inferred dependencies will also be incomplete. To address this challenge, *RESTler* supports annotations, which can be added directly to the specification (as Swagger extensions), in order to explicitly declare dependencies, as well as resource-specific mutations, which can be used for the creation of resources that require some custom format (e.g., an IP address). These two features have proven useful in practice because Azure services use PUT requests to create resources whose user-provided names are passed as URL parameters and, after successful creation, are also returned in the response. For this scenario, one can use resource-name-specific mutations to indicate that a PUT request should create a resource named in a custom format, and then use that name to identify the corresponding dynamic object in subsequent requests.

VIII. Related Work

HTTP Fuzzers. Since REST API requests and responses are transmitted over the HTTP protocol, HTTP-fuzzers can be used to fuzz REST APIs. Fuzzers like Burp [8], Sulley [38], BooFuzz [7], or the commercial AppSpider [4] and Qualys's WAS [34], 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 [4], [34] or user-defined rules [38], [7]. 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 [4], [34], [41], [3]. Compared to those tools, the main originality of *RESTler* is the lightweight static analysis of Swagger specifications in order to infer dependencies among request types, which in turn allows *RESTler* to automatically generate sequences of requests that exercise the business logic exposed by the API in a *stateful manner* and *without pre-recorded HTTP traffic*.

Feedback-directed Test Generation. The dynamic feedback *RESTler* uses to prune invalid requests from the search space (line 32 in Figure 3) is similar to the feedback used in Randoop [32]. However, the Randoop search strategy (in particular, search pruning and ordering) is different from the three search strategies considered in our work, and the Randoop optimizations related to object equality and filtering are not relevant in our context. *RESTler*'s dependency analysis is also related to the analysis of type dependencies performed by the Randoop algorithm [32] for typed object-oriented programs. However, unlike in the Randoop work, dynamic objects in Swagger specifications are implicitly declared and untyped (*e.g.,* authentication tokens or service-specific resources). When a Swagger specification is not complete or *RESTler* cannot infer object types correctly, *RESTler* supports annotations (see Section VII) that the user can use to fix and control *RESTler*'s behavior. In the future, it would be interesting to allow richer user annotations in order to easily specify complex service-specific types as well as their properties, in the spirit of code contracts [28], [5].

Model-based Testing. Our BFS-Fast search strategy is inspired by test generation algorithms used in model-based testing [42], 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 [43]) in order to generate a test suite to check conformance of a (blackbox) implementation with respect to that model. BFS-Fast is also related to algorithms for generating tests from an input grammar while covering all its production rules [26]. 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.

Grammar-based Fuzzing. General-purpose grammar-based fuzzers like Peach [33] and SPIKE [37], among others [39], are not Swagger-specific but can also be used to fuzz REST APIs. With these tools, however, the user has to *manually* construct an API-specific input grammar, often encoded directly by code specifying what and how to fuzz, similar to the code shown on the right of Figure 2. By contrast, *RESTler automatically* generates an input grammar from a Swagger specification, and its fuzzing rules are determined separately and automatically by the algorithm of Figure 3.

Automatically learning input grammars from input samples is another complementary research area [25], [6], [24], [36]. *RESTler* 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. 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 REST services without a Swagger specification, it would be worth investigating how to automatically infer it by using machine learning on runtime traffic logs or static analysis on the code implementing the API.

Whitebox Fuzzing. Grammar-based fuzzing can also be combined [27], [21] with whitebox fuzzing [23], which uses dynamic symbolic execution [22], [9], constraint generation and solving in order to generate new tests exercising new code paths. In contrast, *RESTler* is currently purely blackbox: the inner workings of the service under test are invisible to *RESTler* 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, *RESTler* 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.

Penetration Testing. In practice, the main technique used today to ensure the security of cloud services is the so-called "penetration testing", or *pen testing*, 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 *RESTler* can partly automate the discovery of specific classes of security vulnerabilities, and are complementary to pen testing.

IX. Conclusion

RESTler is the first automatic tool for stateful fuzzing of cloud services through their REST APIs. While still in early stages of development, *RESTler* was able to find 28 bugs in GitLab and several bugs in each of the four Azure and Office365 cloud services tested so far. Although still preliminary, our results are encouraging. How general are these results? To find out, we need to fuzz more services through their REST APIs and check more properties to detect different kinds of bugs and security vulnerabilities. Indeed, unlike buffer overflows in binary-format parsers, use-after-free bugs in web browsers, or cross-site-scripting attacks in web-pages, it is still 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 anecdotal. New automated tools, like *RESTler*, 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.

X. Acknowledgements

We thank William Blum, Dave Tamasi and David Molnar for their helpful comments, and the whole Microsoft Security Risk Detection team for their support. We also thank Albert Greenberg, Mark Russinovich and John Walton, from Microsoft Azure, for encouraging us to pursue this line of work. Finally, we thank the GitLab and Microsoft developers we interacted with, for graciously acknowledging, discussing and fixing the bugs found during this work.

REFERENCES

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

[2] Amazon. AWS. https://aws.amazon.com/.

[3] APIFuzzer. https://github.com/KissPeter/APIFuzzer.

[4] AppSpider. https://www.rapid7.com/products/appspider.

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

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

[7] BooFuzz. https://github.com/jtpereyda/boofuzz.

[8] Burp Suite. https://portswigger.net/burp.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of CCS'2006*, 2006.

[10] 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.

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

[12] Flask. Web development, one drop at a time. http://flask.pocoo.org/.

[13] GitLab. GitLab. https://about.gitlab.com.

[14] GitLab. GitLab API. https://docs.gitlab.com/ee/api/.

[15] GitLab. Hardware requirements. https://docs.gitlab.com/ce/install/requirements.html.

[16] GitLab. Sample Bug1. https://gitlab.com/gitlab-org/gitlab-ce/issues/50955.

[17] GitLab. Sample Bug2. https://gitlab.com/gitlab-org/gitlab-ce/issues/50265.

[18] GitLab. Sample Bug3. https://gitlab.com/gitlab-org/gitlab-ce/issues/50270.

[19] GitLab. Sample Bug4. https://gitlab.com/gitlab-org/gitlab-ce/issues/50949.

[20] GitLab. Statistics. https://about.gitlab.com/is-it-any-good/.

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

[22] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005*, pages 213–223, 2005.

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

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

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

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

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

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

[29] Microsoft. Azure. https://azure.microsoft.com/en-us/.

[30] Microsoft. Office. https://www.office.com/.

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

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

[33] Peach Fuzzer. http://www.peachfuzzer.com/.

[34] Qualys Web Application Scanning (WAS). https://www.qualys.com/apps/web-app-scanning/.

[35] Ruby on Rails. Rails. http://rubyonrails.org.

[36] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. Neuzz: Efficient fuzzing with neural program learning. *CoRR*, abs/1807.05620, 2018.

[37] SPIKE Fuzzer. http://resources.infosecinstitute.com/fuzzer-automation-with-spike/.

[38] Sulley. https://github.com/OpenRCE/sulley.

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

[40] Swagger. https://swagger.io/.

[41] TnT-Fuzzer. https://github.com/Teebytes/TnT-Fuzzer.

[42] 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.

[43] 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.