Intelligent REST API Data Fuzzing

Patrice Godefroid
Microsoft Research
USA
pg@microsoft.com

Bo-Yuan Huang[*]
Princeton University
USA
byhuang@princeton.edu

Marina Polishchuk
Microsoft Research
USA
marinapo@microsoft.com

ABSTRACT

The cloud runs on REST APIs. In this paper, we study how to *intelligently* generate data payloads embedded in REST API requests in order to find data-processing bugs in cloud services. We discuss how to leverage REST API specifications, which, by definition, contain data schemas for API request bodies. We then propose and evaluate a range of data fuzzing techniques, including structural schema fuzzing rules, various rule combinations, search heuristics, extracting data values from examples included in REST API specifications, and learning data values on-the-fly from previous service responses. After evaluating these techniques, we identify the top-performing combination and use this algorithm to fuzz several Microsoft Azure cloud services. During our experiments, we found 100s of "`Internal Server Error`" service crashes, which we triaged into 17 unique bugs and reported to Azure developers. All these bugs are reproducible, confirmed, and fixed or in the process of being fixed.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Correctness*; • **Networks** → *Cloud computing*.

KEYWORDS

REST APIs, JSON data fuzzing, API data-payload testing, cloud security and reliability

ACM Reference Format:
Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API Data Fuzzing. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409719

1 INTRODUCTION

Cloud computing is exploding. Today, most cloud services, such as those provided by Amazon Web Services [11] and Microsoft Azure [26], are programmatically accessed through REST APIs [18], both by third-party applications [10] and other services [27]. REST APIs are implemented on top of the HTTP/S protocol and offer a uniform way to manage cloud resources. Cloud service developers can document their REST APIs using interface-description languages like Swagger (recently renamed OpenAPI) [35]. 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 request and response formats.

REST APIs can be very complex. For instance, REST APIs of Azure services are described in millions of lines of Swagger code publicly available on GitHub [25]. To master this complexity, new tools are needed to prevent expensive outages and SLA violations due to service bugs [24]. Tools for automatically testing cloud services are still in their infancy. Several *fuzzing*[1] tools for REST APIs fuzz and replay manually-defined or previously-captured API traffic to try finding bugs [12, 13, 16, 29, 37]. Perhaps the most advanced (and recent) tool in this space is *RESTler*, which performs *stateful REST API fuzzing* [15]. Given a Swagger specification, *RESTler* automatically generates *sequences of requests* in order to reach deeper service states and find more bugs. Without requiring pre-recorded API traffic, *RESTler* can find bugs such as unhandled exceptions (service crashes), which are detected as `Internal Server Error` responses.

The *data payloads* sent in REST API request *bodies* can be very complex as well. As an example, the Azure DNS service [8] maps domain names to IP addresses following mapping rules defined by users; these rules are specified using JSON data with variable-size arrays, strings, and numerical values that are sent in bodies of REST API requests (see Section 2). What happens when such arrays are re-ordered, or swapped, or made very large, or strings are replaced by numerical values, or parameters are dropped or duplicated? Can the DNS service handle all these cases?

In this paper, we study how to *intelligently* generate data payloads embedded in REST API requests to find data processing bugs in cloud services. By intelligently, we mean fuzzing techniques that can find bugs even with a limited testing budget. For instance, simple *blackbox* random fuzzing [19] works well for binary formats but is ineffective for structured JSON data because the probability of generating new interesting inputs is extremely low [34]. Symbolic-execution-based *whitebox fuzzing* [23] or simpler code-coverage-guided *greybox fuzzing* [38] are not applicable because the cloud service under test is a remote distributed black box. Support for fuzzing complex REST API data payloads is also very limited in existing REST API fuzzing tools. For instance, *RESTler* can only replace body values by other values of the same type selected from a user-defined dictionary of fixed values [15].

This paper aims to fill this void. Specifically, we explore how to leverage REST API specifications, which, by definition, contain

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

[1] Fuzzing means automatic test generation and execution with the goal of finding security vulnerabilities.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409719

```
1  {
2      "etag": "string",
3      "properties": {
4          "registrationVirtualNetworks": [
5              { "id": "string" }
6          ],
7          "maxNumberOfRecordSets": 0,
8          "numberOfRecordSets": 0,
9          "nameServers": [ "string" ],
10         "zoneType": {"enum": ["Public","Private"]},
11         "registrationVirtualNetworks": [
12             { "id": "string" }
13         ],
14         "resolutionVirtualNetworks": [
15             { "id": "string" }
16         ]
17     },
18     "id": "string",
19     "name": "string",
20     "type": "string",
21     "location": "string",
22     "tags": { "string": "string" }
23 }
```

Figure 1: Example of REST API JSON body schema.

data schemas for API request bodies. We then propose and systematically evaluate a wide range of data fuzzing techniques. We proceed in several stages to evaluate the benefit that each technique provides. We start with simple structural schema fuzzing rules, which modify the tree-structure or data types of JSON data (Section 3). Then we study combinations of individual fuzzing rules to identify synergies or redundancies among them (Section 4); because rule combinations generate so much fuzzed data, we also evaluate several search heuristics to deal with this combinatorial explosion. Next, we propose and evaluate two new techniques for generating specific concrete data values that typically need to be provided throughout a body schema: we discuss how to *extract* data values from *examples* included in REST API specifications, and how to *learn* data values *on-the-fly* from *previous service responses* (Section 5).

After evaluating all these techniques, we identify the best combination and use this algorithm to fuzz several Microsoft Azure cloud services (Section 6). During our experiments, we found 100s of "`Internal Server Error`" service crashes, which we triaged into 17 unique bugs and reported to Azure developers. All these bugs are reproducible, confirmed, and fixed or in the process of being fixed. We discuss related work in Section 7 and conclude in Section 8.

2 BACKGROUND AND MOTIVATION

Most cloud services are programmatically accessed through REST APIs [10, 27]. Swagger, also known as OpenAPI, is a popular specification language to define REST APIs [35]. For instance, most public Microsoft Azure services have Swagger API specifications available on GitHub [25]. A Swagger specification describes how client requests can create (`PUT`/`POST`), update (`PUT`/`POST`/`PATCH`), monitor (`GET`), and delete (`DELETE`) cloud resources. Cloud resource identifiers are specified in the *path* or the *body* of the request.

Typically, `PUT`, `POST`, and `PATCH` API requests require additional input-parameter values to be included in the request body. Such parameter values and their format are described in a JSON data *schema* that is part of the API specification. A combination of concrete input-parameter values included in a request body is called a body *payload*.

As an example, Figure 1 shows the schema for the body of the request `PUT DNS-Zone` that creates a new DNS zone in Azure (see https://github.com/Azure/azure-rest-api-specs under the directory `specification/dns/`). This schema can be viewed as a tree with 22 nodes. For instance, the root node (on line 1) is an object, which has 7 children. The first child is named `etag` (line 2) and is of type `string`. The second child is an object named `properties` (line 3). This object has itself a child named `registrationVirtualNetworks` (line 4) of type `array` (denoted with `[]`), and so on. In line 10, the node `zoneType` is of `enum` type and takes any value among the specified array of constants (here either the string constant `Public` or `Private`). In line 22, `tags` is an object which can have key-value pairs as children where both the key and value are of type `string`.

Because this schema includes objects, arrays, and strings of (a priori) unbounded sizes, as well as numerical values, there are infinitely (or astronomically) many ways to generate concrete input-parameter values, i.e., payloads, *satisfying* the schema. Worse, there are even more ways to generate body payloads *violating* the schema, which may also be worth testing in order to find bugs in the code processing API requests. Also, REST API data schemas are sometimes much larger than this simple example.

Given a REST API data schema, what are the most effective test-generation techniques to fuzz the body of REST API requests? The purpose of this paper is to address this question.

Many API requests with non-empty bodies are used to create or update service resources that can be reached only after creating parent resources. For instance, the Azure DNS service consists of 13 request types, and only 4 of these have non-empty bodies: a `PUT` request to create a `DNS-Zone` and whose body schema of 22 nodes is shown in Figure 1, a `PATCH` request to update a `DNS-Zone` with a schema of only 2 nodes, a `PUT` request to create a `DNS-Record-Set` with a schema of 65 nodes, and a `PATCH` request to update a `DNS-Record-Set` with a schema of 65 nodes. A valid `DNS-Zone` identifier must be provided in the path of a `PUT` request that creates a `DNS-Record-Set`, because a `DNS-Record-Set` is a child resource of a parent `DNS-Zone`.

In order to reach deeper service states where child resources can be created, hence increasing the number of requests with non-empty bodies we can fuzz, we build upon recent work on *stateful REST API fuzzing* [15]. Specifically, we leverage the tool *RESTler* [15], which performs an initial static analysis of a Swagger specification to infer the parent-child dependencies and generate *sequences* of requests (instead of single requests) to reach such deeper states. A test suite generated by *RESTler* attempts to cover as much as possible of the input Swagger specification, although full specification coverage is not guaranteed. While testing a service, *RESTler* reports all `Internal Server Error` responses (HTTP status code 500). These are unhandled exceptions (service crashes) that may severely damage service health.

In this work, we investigate how to extend stateful REST API fuzzing in general, and *RESTler* in particular, by intelligently fuzzing REST API data (body payloads) to find even more `Internal Server Error` bugs in service code that processes complex data.

```
// example schema
// nodes
V = { root,
  tag, properties,
  id, time
}
// edges
E = { (root, tag),
  (root, properties),
  (properties, id),
  (properties, time)
}
// type
T(root) = object
T(tag) = string
T(properties) = object
T(id) = string
T(time) = integer
```

```
// example payload
{
  "tag":"global",
  "properties": {
    "id":"abcd",
    "time":3600
  }
}
```

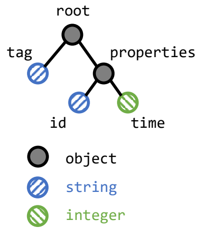


Figure 2: Example of schema and payload.

In the next Sections 3 to 5, we propose various JSON payload data fuzzing techniques, and we evaluate their effectiveness using the Azure DNS service as a benchmark. On the one hand, the DNS service is a real, non-trivial (with body schemas up to 65 nodes), widely-used Azure service. On the other hand, it is small enough (13 request types) to run many experiments quickly (in hours) and simple enough to allow non-experts to analyze results.

3 SCHEMA FUZZING

In this section, we define *schema fuzzing rules* that take as input a body schema and return a set of *fuzzed-schemas*.

3.1 Schema and Fuzzed-Schema

A request body schema is encoded in the JSON format. It can be viewed as a tree in which each node corresponds to a property field and is labeled with a type. Formally, a schema is defined as a tree-structure $G = (V, E, \Gamma, T)$, where V is a set of nodes, $E = \{(p, q) \mid p, q \in V\}$ is a set of edges, $\Gamma = \{\text{string, integer, Bool, object, array}\}$ is a set of supported types, and $T : V \longrightarrow \Gamma$ is a type-labeling function mapping each node to one type. A fuzzed-schema is defined similarly.

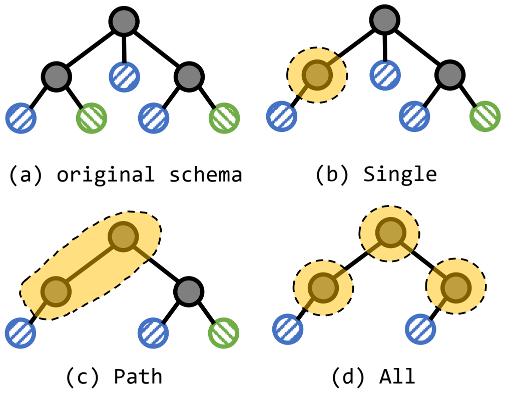
Figure 2 shows an example of schema with five nodes, an example of a concrete JSON payload satisfying the schema, and a pictorial representation of the schema tree structure with its labeled types.

3.2 Schema Fuzzing Rules

Given a schema G, a *schema fuzzing rule* modifies its tree structure (V and E) or its type-labeling function (T) to generate a *set* of fuzzed-schemas. Moreover, a schema fuzzing rule can be applied once or multiple times to a given schema.

3.2.1 Node Fuzzing Rules. A *node fuzzing rule* defines how to modify a node in a schema. In our schema fuzzer, we implemented four node fuzzing rules: (1) `Drop` (2) `Select` (3) `Duplicate`, and (4) `Type`.

Drop. Given an internal node $n \in V$ in the schema $G = (V, E, \Gamma, T)$, the node fuzzing rule `Drop` removes one child node c of node n, where $(n, c) \in E$. Other child nodes remain unchanged.


(a) original schema (b) Single

(c) Path (d) All

Figure 3: The original schema and the fuzzed-schemas generated by applying the node fuzzing rule `Drop` using different tree fuzzing rules (`Single`, `Path`, and `All`).

Select. Given an internal node $n \in V$ in the schema $G = (V, E, \Gamma, T)$, the node fuzzing rule `Select` keeps only one child node c of node n, where $(n, c) \in E$. All other child nodes of n are removed.

Duplicate. Given an internal node $n \in V$ in the schema $G = (V, E, \Gamma, T)$, the node fuzzing rule `Duplicate` adds a new child node r to n by copying an existing child c of n. The descendant nodes of c (i.e., the subtrees) are also copied.

Type. The node fuzzing rule `Type` changes the labeled type of a node $n \in V$ in a schema $G = (V, E, \Gamma, T)$ and generates a fuzzed-schema $G' = (V, E, \Gamma, T')$ where $T'(n) \neq T(n)$. Note that changing the type of an internal node may have side effects on the tree structure (e.g., changing an array to a string removes all the child nodes). In contrast, changing the type of a leaf node to object or array preserves the tree structure, because those objects or arrays are empty.

3.2.2 Tree Fuzzing Rules. A *tree fuzzing rule* defines how to apply a node fuzzing rule over a schema tree to produce a new fuzzed-schema tree. In our fuzzer, we implemented three different tree fuzzing rules: (1) `Single` (2) `Path`, and (3) `All`.

Single. Given a node fuzzing rule and a schema, the tree fuzzing rule `Single` applies the node fuzzing rule on one single node while keeping all other nodes unchanged. The rule `Single` applied exhaustively on the entire schema tree yields the smallest set of fuzzed-schema variants (linear in the original schema size).

Path. Given a node fuzzing rule and a schema, the tree fuzzing rule `Path` selects a path in the schema tree, then selects a set of nodes on that path, and finally applies the node fuzzing rule to every node in that set. The tree fuzzing rule explores more structural and type variants than `Single` does. However, multiple sibling nodes will never be modified.

All. Given a node fuzzing rule and a schema, the tree fuzzing rule `All` selects a set of nodes in the schema tree, then applies the node fuzzing rule to every node in that set. This rule generalizes both `Single` and `Path`, but can generate exponentially-many fuzzed-schema variants.

Figure 3 (a) shows an example schema, a tree of eight nodes with labeled types. Given this schema, Figure 3 (b), (c), and (d) shows one fuzzed-schema that can be generated by applying `Drop` using `Single`, `Path`, and `All`, respectively. The node fuzzing rule `Drop` is applied to the dash-circled (highlighted) nodes.

3.3 Experimental Evaluation

To evaluate the effectiveness of these node and tree fuzzing rules, we performed experiments with various fuzzing rule combinations using the Azure DNS service as a target.

3.3.1 *Evaluation Metric.*

The cloud services we aim to fuzz are black boxes to us: we cannot instrument their code to measure code coverage. In order to evaluate fuzzing effectiveness in a consistent way, not just by counting bugs found (since bugs are rather rare), we introduce a new coverage metric for cloud services tested through REST APIs: the response *error type coverage metric.*

Error Code. When a service fails to process a request, it returns an *error code* to notify the client of this failure. Minimally, every REST API request returns an HTTP status code, which is in the `40x` range when the failure is triggered by an invalid yet handled request, or in the `50x` range for unhandled conditions or generic failures to process the request. In addition, a service may define its own finer-grained error code that includes domain-specific information. For the DNS service example, a response with the error code `DomainNameLabelMissing` may be received if the request body payload does not provide the required labeling.

Error Message. In addition to an error code, the response for a failed request typically also includes an *error message.* This message is valuable in that it further describes how the payload is being processed, especially when the same error code is used for many invalid requests. For example, the error messages `"The resource record is missing field 'target'"` and `"Record type SRV is not supported for referencing resource type 'dnsZones'"` both return the same error code `BadRequest`. These two messages provide additional context for the errors, which cannot be distinguished by using the error code alone.

Error Type. We define an *error type* as a pair of error code and error message. (Error messages are sanitized by removing runtime specific information, such as timestamps, session ids, GUIDs, etc.) The number of distinct error types is used as the effectiveness metric in our experiments: we will favor fuzzing techniques that maximize error type coverage.

3.3.2 *Experiment Settings.*

We implemented our schema fuzzer as an extension of *RESTler.* In all the experiments reported in this paper, we run *RESTler* under its "test mode" where it attempts to generate one valid response for every request type [15]. When *RESTler* tests a request type for the first time, our new schema fuzzer is called to generate variants of the body payload of that request. Thus, our payload schema fuzzer is called once for each request type with a non-empty body schema.

We experimented on all 12 combinations (4 node fuzzing rules and 3 tree fuzzing rules) under a maximum bound of 1,000 fuzzed-schemas per request type. Thus, if any combination generates more than 1,000 fuzzed-schemas, only the first 1,000 will be tested. (We systematically enumerate possible fuzzed-schemas, and the process is deterministic.) Since DNS has 4 request types with non-empty payloads, at most 4,000 fuzzed-schemas were tested per node/tree fuzzing rule pairs.

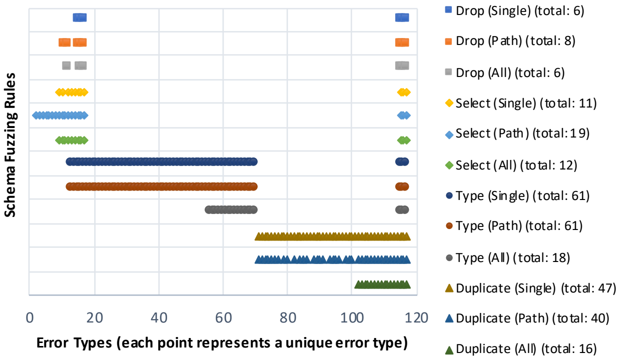


Figure 4: Error type coverage for each schema fuzzing rule.

Given a fuzzed-schema, a JSON payload is *rendered* by filling in concrete values based on the labeled type of each leaf node. In this experiment, we use `"fuzzstring"`, `0`, `false`, `{}`, and `[]` for leaf nodes labeled with type string, integer, Boolean, object, and array, respectively. The value rendering is only based on the labeled types. (We will discuss other value rendering strategies in Section 5.)

3.3.3 *Experiment Results.*

Figure 4 shows the error types discovered by the 12 schema fuzzing rules (node/tree fuzzing rule pairs). Each column represents a unique error type, whereas each row reports which error types were found by the specific schema fuzzing rule. The total count for each rule is shown in the legend.

Drop and Select. For node fuzzing rules `Drop` and `Select`, using the `Path` tree fuzzing rule covers more distinct error types than using other tree fuzzing rules. Both `Drop` and `Select` modify a schema by removing nodes from the original tree. A removed node can be either a required property, or optional and used only under certain conditions (e.g., in the absence of another node). Therefore, applying such structural modifications on different nodes (i.e., `Path` and `All`) is effective. However, since `All` generates an exponential number of fuzzed-schemas and the fuzzing budget is limited, it ends up testing many redundant combinations before quickly running out of budget. In contrast, `Path` modifies the nodes along a single path, restricting the fuzzing combinations of descendants of sibling nodes. Since child nodes of sibling nodes are usually independent, `Path` avoids generating many useless combinations of independent sub-trees. Therefore, the tree fuzzing rule `Path` works best for node fuzzing rules `Drop` and `Select` when the budget is limited.

Type. The node fuzzing rule `Type` modifies a schema by changing the labeled types of its nodes. As shown in Figure 4, there is some overlap between the error types triggered by `Drop`, `Select`, and `Type`. This is due to the structural side effects of the node fuzzing rule `Type`, as discussed in Section 3.2.1. In addition to structural side effects, `Type` may also introduce deserialization errors caused by type mismatches, which often terminate the payload parsing process immediately. This makes it less effective to apply the node fuzzing rule `Type` on multiple nodes at a time (i.e., `Path` and `All`).

By analyzing the results further (data not shown here), we also see that changing the types of internal nodes is as effective as changing the types of leaf nodes. Further, when changing the labeled type of a node, the new type matters. For example, changing a string-typed node to an integer or an object can trigger different error types.

Duplicate. The node fuzzing rule `Duplicate`, in contrast to `Drop` and `Select`, modifies a schema by adding new nodes to the original tree. This can introduce the *duplicate-keys* error when inserting a duplicate key-value pair to an object-typed node. In other words, the payloads rendered from such fuzzed-schemas will violate the JSON format, and thus result in deserialization errors. Therefore, applying this kind of modification on multiple nodes at a time (i.e., `Path` and `All`) does not provide much benefit, although it consumes a great portion of the limited budget.

Rules are complementary. Although the node fuzzing rules `Drop` and `Select` discover fewer error types, there are some error types that cannot be triggered by `Type` or `Duplicate`. They are usually tree structure related, for example, the error type with the error code `"LocationRequired"` is only discovered by `Drop` and `Select`. Similarly, there are deserialization-related error types that are uniquely triggered by either `Type` or `Duplicate`. Error types covered by multiple fuzzing rules are mostly due to bad value rendering (e.g., `"Expect fully qualified resource Id that start with '...'"`) rather than due to a fuzzed structure or type.

3.3.4 Conclusion.

- The tree fuzzing rule `Single` works best for node fuzzing rules that trigger deserialization errors, such as `Type` and `Duplicate`.
- The tree fuzzing rule `Path` works best for node fuzzing rules that modify the tree structure without introducing deserialization errors, such as `Drop` and `Select`.
- Different node fuzzing rules are able to discover different kinds of error types and are thus complementary.

All the above observations hold for every single DNS request type with a non-empty body schema.

From these conclusions, we select 4 schema fuzzing rules as the building blocks of our payload fuzzer: `Drop` with `Path` (denoted DROP), `Select` with `Path` (denoted SELECT), `Duplicate` with `Single` (denoted DUPLICATE), and `Type` with `Single` (denoted TYPE).

4 COMBINING SCHEMA FUZZING RULES

In this section, we combine multiple schema fuzzing rules in *pipelines* and evaluate the effectiveness of such combinations.

4.1 Pipelining Schema Fuzzing Rules

Since the 4 schema fuzzing rules DROP, SELECT, DUPLICATE, and TYPE are complementary, perhaps combining these could trigger even more error types in the service under test. To explore this idea further, we combine schema fuzzing rules in a sequential pipeline: one fuzzing rule is applied to an initial body schema and generates a set of fuzzed-schemas, then a second fuzzing rule is applied to all these fuzzed-schemas and generates even more fuzzed-schemas, and so on. For example, consider a two-stage pipeline, denoted as DROP-TYPE, with its first stage associated with the schema fuzzing rule DROP and the second stage associated with TYPE. It first takes an original schema G and generates a set of fuzzed-schemas $\mathsf{DROP}(G) = \{G_1, G_2, \ldots, G_n\}$ by applying the schema fuzzing rule DROP. It then applies TYPE to every $G_i \in \mathsf{DROP}(G)$, to get the final set of fuzzed-schemas:

$$\mathsf{DROP\text{-}TYPE}(G) = \bigcup_{G_i \in \mathsf{DROP}(G)} \mathsf{TYPE}(G_i)$$

4.1.1 Search Heuristics.

Since pipelining schema fuzzing rules results in enormous numbers of new fuzzed-schemas but fuzzing budgets are limited, we propose and evaluate 3 heuristics to select fuzzed-schemas generated by pipelining fuzzing rules: (1) Depth-First (DF), (2) Breadth-First (BF), and (3) Random (RD).

Depth-First (DF). Given a maximum bound M, the search heuristic DF generates fuzzed-schemas in depth-first order with respect to the pipeline stages and selects the first M fuzzed-schemas. For example, with DF, a two-stage pipeline DROP-TYPE takes an initial input schema G, generates a first fuzzed-schema $G_1 \in \mathsf{DROP}(G)$, and then generates the set $\mathsf{TYPE}(G_1)$ of fuzzed-schemas. It then continues generating fuzzed-schemas $\mathsf{TYPE}(G_i)$ for other G_i in $\mathsf{DROP}(G)$ (one by one) until the bound M is reached. In other words, the search heuristic DF prioritizes more fuzzing in the later stages than in the earlier stages.

Breadth-First (BF). In contrast to DF, the search heuristic BF prioritizes fuzzing more in the earlier stages by generating fuzzed-schemas in breadth-first order. For example, with BF, a two-stage pipeline DROP-TYPE taking as input an initial schema G first generates all fuzzed-schemas G_i in $\mathsf{DROP}(G)$, then it will generate the fuzzed-schemas in $\mathsf{TYPE}(G_i)$ for some $G_i \in \mathsf{DROP}(G)$, and so on up to the given bound M.

Random (RD). While DF and BF prioritize fuzzing in either the later or earlier pipeline stages, respectively, the search heuristics RD uses a random search order that does not favor specific stages. For example, with RD and some random seed, a two-stage pipeline DROP-TYPE taking as input an initial schema G first generates some fuzzed-schema $G_1 \in \mathsf{DROP}(G)$, then generates some fuzzed-schema $G_2 \in \mathsf{TYPE}(G_1)$, then generates some fuzzed-schema $G_1' \in \mathsf{DROP}(G)$, then generates some fuzzed-schema $G_2' \in \mathsf{TYPE}(G_1')$, and so on until the given bound M is reached.

4.2 Experiments

To study the effectiveness of combining schema fuzzing rules as a pipeline, we compare various rule combinations under different search heuristics.

4.2.1 Experiment Settings.

Similar to the experiments of Section 3.3, we fuzz the Azure DNS service. Each schema fuzzing rule pipeline (regardless of the search heuristic) is bounded by a maximum number of 1,000 fuzzed-schemas per request type. For value rendering, we use the same type-value mapping as in Section 3.3. We compare different rule combinations and search heuristics based on the error types obtained from responses.

Rule Combination. Based on the results in Section 3.3, we group the four schema fuzzing rules into three groups: (1) DROP and SELECT that discover structure related errors, (2) TYPE that triggers

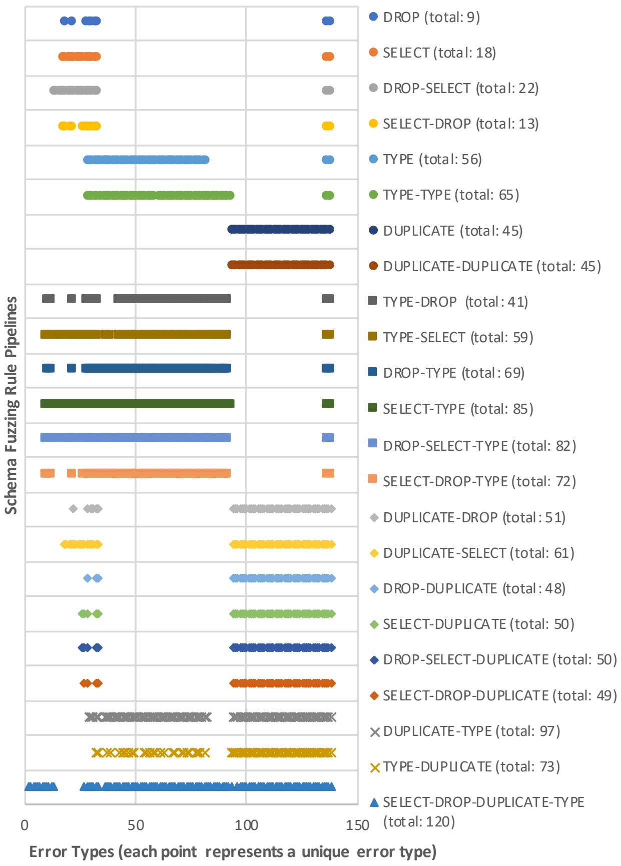


Figure 5: Error type coverage for each schema fuzzing rule pipeline (search heuristic: RD, random seed: 0).

deserialization errors due to type mismatches, and (3) DUPLICATE that discovers deserialization errors triggered by malformed JSON request payloads (duplicated keys). *The three schema fuzzing rule groups tend to have disjoint error type coverage.* In this experiment, we implemented 23 schema fuzzing rule pipelines to cover different combinations of the three groups. For example, we have two pipelines (TYPE-DUPLICATE and DUPLICATE-TYPE) for combining the second and third groups.

Search Heuristic. To compare how the search heuristics affect the schema fuzzing rule pipelines given a limited budget, we ran all the 23 pipelines using DF, BF, and RD. Experiments with RD are repeated five times using five different random seeds (0, 1, 2, 3, and 4). Each pipeline will thus be evaluated 7 times in total. (The only source of randomness is in RD with a random seed, whereas DF and BF are deterministic.)

4.2.2 Experiment Results.

Rule Combination. Figure 5 compares the error type coverage of each schema fuzzing rule pipeline when using the RD search heuristic with a random seed 0. It includes the results of all four DNS request types with non-empty body schemas. Pipelines combining DROP, SELECT, and TYPE are marked in squares. Pipelines combining DROP, SELECT, and DUPLICATE are marked in diamonds. Pipelines combining TYPE and DUPLICATE are marked in crosses. The pipeline that combines all the three groups (i.e., SELECT-DROP-DUPLICATE-TYPE) is marked in triangles. As a baseline, pipelines that do not combine multiple groups are marked in circles.

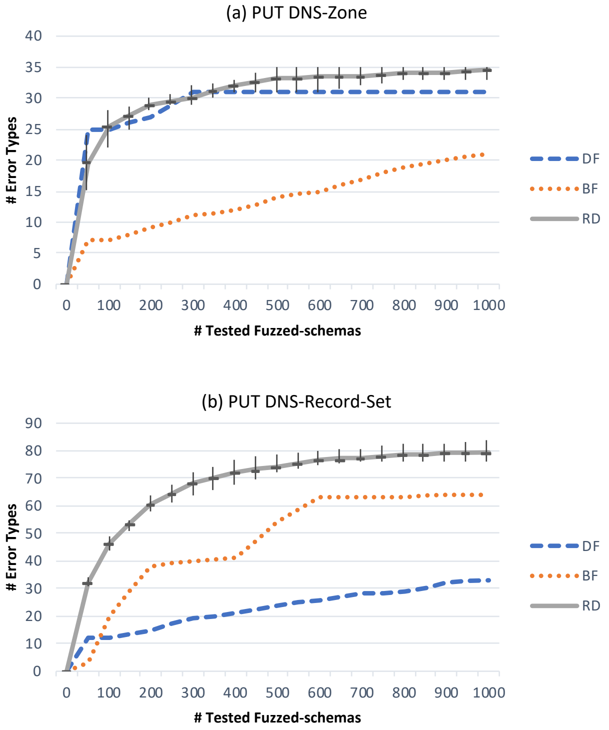


Figure 6: The growth trends of the number of error types discovered over the number of tested fuzzed-schemas (schema fuzzing rule pipeline: SELECT-DROP-DUPLICATE-TYPE).

We can observe the following. Combining schema fuzzing rules DROP, SELECT, and TYPE as a pipeline is beneficial, in that it helps discover *new* error types that are not triggered by DROP, SELECT, or TYPE alone. Furthermore, based on a finer-grained analysis of the results, having DROP or SELECT at stages earlier than TYPE usually has a better error type coverage than the opposite. On the other hand, combining DUPLICATE with other schema fuzzing rules does not provide significant improvements: although the total number of covered error types is higher, the coverage is mostly the union of the individual ones. Note that the above observations hold for all 7 runs with different search heuristics, not only just the one using RD with a random seed 0.

Search Heuristic. Figure 6 compares how the total number of covered error types grows as additional fuzzed-schemas are tested, using different search heuristics. These results are obtained by applying the pipeline SELECT-DROP-DUPLICATE-TYPE to the requests PUT DNS-Zone and PUT DNS-Record-Set, shown in parts (a) and (b), respectively. The dashed (blue) and dotted (orange) lines correspond to using DF and BF, respectively, while the gray line shows the average of all 5 runs using RD. The gray bars show the variation ranges among the 5 runs using different random seeds.

From these experiment results, we see that using RD, regardless of the random seed used, provides a more stable growth rate. This is desirable and important when only a subset of the fuzzed-schemas (the first few generated) can be tested given a limited fuzzing budget. Remarkably, besides the two configurations (a) and (b) shown in Figure 6, we observed (data not shown here) similar conclusions for all experimented schema fuzzing rule pipelines and for all DNS request types with non-empty body schemas. This indicates that the search heuristic RD is effective regardless of the schema structure.

4.2.3 Conclusion.

- Combining schema fuzzing rules DROP, SELECT, and TYPE as a pipeline is helpful, especially when having DROP and SELECT before TYPE.
- Combining the schema fuzzing rule DUPLICATE with other rules does not provide significant benefit in covering new error types.
- The RD search heuristic provides a more stable growth rate in covering unique error types, and is therefore more favorable when the budget is limited.

All of the above conclusions were observed for all DNS request types with a non-empty body schema.

5 DATA VALUE RENDERING

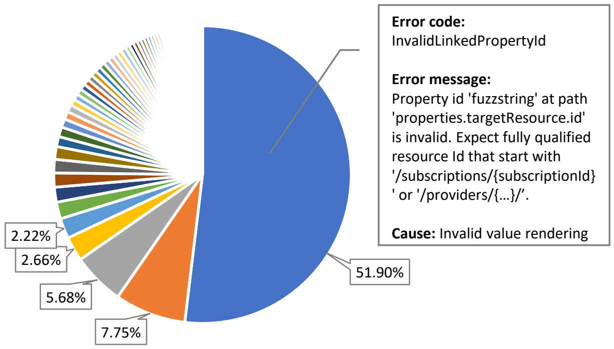
We now discuss several data value rendering strategies, i.e., how we render fuzzed-schemas with concrete values.

5.1 Challenges in Value Rendering

As described in Section 3.1, a body (fuzzed-)schema defines an overall tree structure and labeled types. Every leaf node represents a property field that needs to be rendered with a concrete value to form a complete JSON payload. Unfortunately, this rendering process is non-trivial and may require some domain knowledge of the service under test. For instance, a specific service request with a string-typed node `location` might accept the value `"global"` but not `"us"` or `"europe"`, even though all of these are syntactically-valid string-typed values and, moreover, may be accepted in other contexts for `location`. In practice, many requests end up being rejected due to a single specific invalid value rendering of one single node in their body payload.

Figure 7 compares how often each error type is triggered during one of the experiments of Section 4.2 (the one with schema fuzzing rule pipeline DROP-SELECT-TYPE). Each partition corresponds to a unique error type, showing the percentage of tests (over all tests) that trigger that error type. As the figure shows, the top-5 most frequent error types consume more than 70% of the total fuzzing budget. Based on the error messages, *all* these error types are due to invalid value renderings, such as rendering the node `id` with value `"fuzzstring"`. In other words, regardless of what the tree structures and labeled types of these fuzzed-schemas are, the service under test rejects these payloads due to a specific invalid value rendering of one single node (e.g., node `id`).

This value rendering barrier can be broken down into the following root causes.



Error code:
InvalidLinkedPropertyId

Error message:
Property id 'fuzzstring' at path 'properties.targetResource.id' is invalid. Expect fully qualified resource Id that start with '/subscriptions/{subscriptionId}' or '/providers/{...}/'.

Cause: Invalid value rendering

Figure 7: Percentage of tests triggering each error type (schema fuzzing rule pipeline: DROP-SELECT-TYPE).

- Lack of **client-specific information**, such as subscription ID and resource group name.
- Lack of **domain-specific information**, for example, only `"local"` and `"global"` are valid location values, and a timeout value can only be a positive integer smaller than 3,600.
- Lack of **run-time dependent information**, such as the name of a resource dynamically created by a previous request.

5.2 Value Rendering Strategies

Given a body schema, *RESTler* can only replace body values by other values of the same type selected from a user-defined dictionary of values [15]. Unfortunately, this simple strategy is insufficient to address the above challenges. We now discuss several new *value rendering strategies* that significantly extend the *RESTler* functionality and effectiveness for dealing with body payloads.

5.2.1 Static Type-Value Mapping. The simplest way of assigning a concrete value to a leaf node in a fuzzed-schema is to have a type-value mapping, which maps each type to a single value. In our body schema fuzzer, we use the same mapping as discussed in Section 3.3, namely `"fuzzstring"`, 0, `false`, `{}`, and `[]` for leaf nodes labeled with type string, integer, Boolean, object, and array, respectively. This simple strategy can be used by default, as a baseline, but it does not address the lack of either client-specific, domain-specific, or run-time dependent information.

5.2.2 Examples from the Swagger Specification. As discussed in Section 2, a Swagger specification may contain examples of concrete JSON payloads. These examples, if present, are useful for getting concrete values for nodes, especially those that require domain-specific information.

However, examples do not help discover client-specific and run-time dependent values. Moreover, the provided examples usually cover only a subset of nodes (e.g., the required property fields) and leave the rest unspecified.

5.2.3 Learning from Responses. The response to a valid request may contain information on the service state, as opposed to an error message when the request is invalid. For example, the response to a

```
{
  "name":"object-1234-abcd",
  "properties": {
    "type":"Public",
    "numberOfTags":2,
    "maxNumberOfTags":5000
  },
  "location":"global",
  "id":"/subscriptions/subid/resourceGroup/..."
}
```

Figure 8: Example response payload of a successful request.

successful PUT request may contain the identification of the newly created resource, i.e., run-time dependent information. Similarly, the responses to successful GET and PATCH requests may return details of the target resources. Unlike the request examples provided in a Swagger specification (which is based on the body schema), the responses may have properties not declared in the body of the request. Often, the response schema is actually similar to the request body schema, which makes it possible to re-use response values for some parameters in the body of future requests. In other words, learning from responses may reveal the context of the current client-service interaction and potentially provides client-specific, domain-specific, and run-time dependent information.

Matching values in responses. The payload body of a response is represented as a JSON object, like the example provided in Figure 8. We extract the value of each leaf node in the response payload body and tag it with its path in the JSON hierarchy. For instance, "object-1234-abcd" and "Public" will be tagged with *name* and *properties.type*, respectively. We collect this pool of tagged values on-the-fly by analyzing responses, and keep only the most recent value for each tag. These values are then used as candidate values when rendering a fuzzed-schema.

In selecting a concrete value for a node, we use pattern matching to compare the tags of candidate values to the node path in the fuzzed-schema tree structure. Two levels of precision are considered: (1) conservative and (2) aggressive. When in conservative mode, a candidate value is chosen for a node n only if its tag exactly matches the node path of n in the fuzzed-schema. For the example in Figure 8, given a node n_{type} in the fuzzed-schema, we select the candidate value "Public" for it only if its parent is $n_{properties}$ and there are no other parents. On the other hand, under aggressive mode, we only compare the last level (leaf) in the hierarchy. For the example in Figure 8, as long as a candidate value has a tag suffixed with type, it will be chosen for the node n_{type}, regardless of the parent nodes.

5.2.4 *Supported Value Rendering Strategies.*

Following the previous discussion, we implemented 6 different value rendering strategies.

(1) **Baseline** (BAS): Select a value for a node using only the type-value mapping.
(2) **Examples only** (EXM): Select a value for a node by using the examples; use the type-value mapping if no example is available.
(3) **Responses only (conservative)** (CON): Select a value for a node using the responses in conservative mode; use the type-value mapping if no candidate value is available.

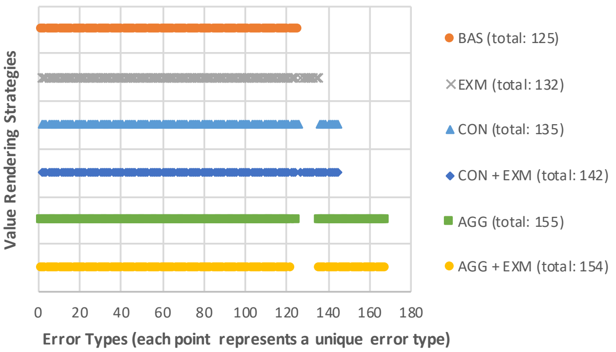


Figure 9: Error type coverage for each value rendering strategy (pipelines: DUPLICATE and DROP-SELECT-TYPE).

(4) **Responses only (aggressive)** (AGG): Select a value for a node using the responses in aggressive mode; use the type-value mapping if no candidate value is available.
(5) **Responses (conservative) and examples** (CON+EXM): Select a value for a node using the responses in conservative mode; use examples if no candidate value is available; otherwise, use the type-value mapping.
(6) **Responses (aggressive) and examples** (AGG+EXM): Select a value for a node using the responses in aggressive mode; use examples if no candidate value is available; otherwise, use the type-value mapping.

5.3 Experiments

5.3.1 *Experiment Settings.*

We now evaluate those 6 different value rendering strategies. Similar to the experiments of Section 4.2, we use the Azure DNS service as a fuzzing target. Following the results of Section 4, we select the two "best" schema fuzzing rule pipelines to generate a set of fuzzed-schemas: DUPLICATEand DROP-SELECT-TYPE. Each pipeline is bounded by a maximum number of 1,000 fuzzed-schemas per request type, and uses the RD search heuristic with a random seed 0. Based on the same set of fuzzed-schemas, we compare all the 6 value rendering strategies described in Section 5.2.4. In other words, we fuzz Azure DNS 6 times using the same set of fuzzed-schemas but render them using different value rendering strategies. All 6 execute the same number of tests. As before, we evaluate fuzzing effectiveness using the covered error types obtained from the responses received.

5.3.2 *Experiment Results.*

Figure 9 shows the error types covered by each of the 6 value rendering strategies. The results include all 4 DNS request types with non-empty body schemas.

Compared to the baseline (BAS), both using the examples from the Swagger specification (EXM) and matching the values in the responses (CON) are helpful in covering more error types. For example, an error type with an error message "Record type TXT is not supported for referencing resource type dnsZones" is never triggered when using the baseline strategy BAS.

Although CON already shows some benefit of learning from responses, its effectiveness is still limited when the request body does not share the same schema with the response body. This can be seen

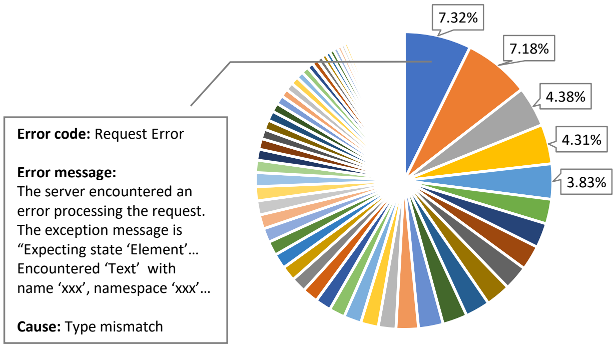


Figure 10: Percentage of tests triggering each error type (rendering strategy: AGG+EXM, pipeline: DROP-SELECT-TYPE).

in the comparison between AGG vs. CON and AGG+EXM vs. CON+EXM, where matching response values in the aggressive mode brings even more improvements than in the conservative mode.

In general, using both the responses and the examples yields better results. However, AGG+EXM does not outperform AGG significantly. This is because the value rendering strategy AGG+EXM prioritizes the values from responses over those from examples. Furthermore, in aggressive mode, almost every node will find a match in some responses, and thus examples are barely utilized.

After using the new value rendering strategies introduced in Section 5.2.4, the distribution of error types significantly changes and becomes more uniform: value-rendereing bottlenecks now disappear. Figure 10 shows the percentage of tests triggering each error type when applying the AGG+EXM value rendering strategy. Compared to Figure 7, which uses BAS, the distribution is more even and the most-frequently-triggered error type is now due to type mismatch.

5.3.3 *Conclusion.*

- Using examples from the Swagger specification when selecting a value for a leaf node is helpful in covering more error types.
- Learning from responses when selecting a value for a leaf node provides a significant improvement in covering more error types, especially when matching the values in the aggressive mode.
- Utilizing both responses and examples is usually beneficial in covering more error types.

6 BUG HUNTING IN CLOUD SERVICES

6.1 REST API Data Fuzzing Algorithm

Based on the evaluation results presented in Sections 3 to 5, we now define our overall REST API data fuzzing algorithm used for longer fuzzing experiments to find bugs in cloud services. The algorithm is divided into two phases, using the fuzzing rules DUPLICATE and DROP-SELECT-TYPE, respectively. Each phase has a budget of 10 times the number of nodes in the schema. The first phase focuses on generating data payloads violating the JSON format, while the second phase focuses on testing various tree structures and type mismatches. When pipelining schema fuzzing rules, we use the RD search heuristic with a random seed 0. We render every selected fuzzed-schema using the AGG+EXM value rendering strategy.

The second phase DROP-SELECT-TYPE is itself divided into two steps. In the first step, we apply the schema fuzzing rule pipeline DROP-SELECT to the original schema and test the generated fuzzed-schemas up to 10% of the budget; during this testing process, we analyze the responses on-the-fly and select one fuzzed-schema for every unique error type. In the second step, we apply the fuzzing rule TYPE to all previously selected fuzzed-schemas for up to 90% of the budget.

All the results of this Section were obtained using this algorithm.

6.2 Experimental Setup

To evaluate the effectiveness of our new REST API data fuzzing algorithm at finding new bugs in existing cloud services, we fuzzed Microsoft Azure cloud services related to networking. They are used, for example, to allocate IP addresses and domain names, or to provide higher-level infrastructures, such as load balancers and firewalls.[2] Specifically, we fuzzed the Azure DNS service already used as a benchmark in the previous sections, and a large collection of other networking services. Their Swagger specifications are publicly available on GitHub [25] under the specification/dns/ and specification/network/ directories, respectively.

The Azure DNS API consists of 13 request types described in about 4,000 lines of Swagger specifications (including examples).

The Azure networking API is much larger: it is a collection of 38 different APIs, targeting different services, that are written and maintained by different service owners. Overall, this API consists of 371 request types described in about 58,000 lines of Swagger specifications across 435 files (including examples).

All experiments were conducted on a single commodity PC with an Intel i7 processor and 32GB of main memory under Windows 10. We ran *RESTler* extended with our new REST data fuzzing algorithm in a single thread and process. We used a regular Azure subscription[3] to authenticate and access the Azure services being tested. No other special test setup or service knowledge was required.

6.3 Azure DNS Bugs Found

Table 1 summarizes the bugs found while fuzzing the bodies of the 4 (out of 13) DNS requests with non-empty body schemas. Overall, in 232 minutes of fuzzing, we generated 14,964 tests (API requests) and found 27 instances of Internal Server Error bugs. After triaging these, we reported 7 unique bugs to Azure developers. These bugs are briefly described in Table 1. Bug 1 was found by replacing a valid location (like "global") by an empty array. This bug was also found multiple times in Azure networking services, as will be discussed below. Bug 2 was found when the type of DNS record-set specified in the path of the REST API request did not match the type of Records in the body of the request after fuzzing that body; 7 variants were found by our fuzzer. Bug 3 occurs when Records becomes empty or an empty array. Bug 4 was triggered when replacing an integer value by a boolean value like false; in total, 13 integer nodes were subject to this bug. Bugs 5, 6 and 7

[2] See https://azure.microsoft.com/en-us/product-categories/networking/
[3] Anyone can get a free trial Azure account from azure.com.

Table 1: Overview of the bugs found in Azure DNS REST APIs.

#	API Request	Schema node	Bug description	Variants
1	PUT DNS-Zone	location	Type mismatch (string to array or boolean)	2
2	PUT DNS-Record-Set	Records	Enum type mismatch between request path and body (e.g., SRV and caaRecords)	7
3	PUT DNS-Record-Set	Records	Missing node (e.g., "ARecords":{} or "ARecords":[{}])	3
4	PUT DNS-Record-Set	value	Type mismatch (integer to boolean or string)	13
5	PATCH DNS-Record-Set	Records	Enum type mismatch between request path and body (e.g., AAAA and SRVRecords)	5
6	PATCH DNS-Record-Set	Records	Missing node (e.g., "TXTRecords":{} or "TXTRecords":[{}])	2
7	PATCH DNS-Record-Set	TTL	Type mismatch (integer to boolean or string)	7

Table 2: Overview of the bugs found in Azure Networking REST APIs.

#	API Request	Schema node	Bug description	Variants
1	PUT virtualWans	location	Type mismatch (string to array or boolean)	28
2	PUT virtualNetworks	addressSpace	Type mismatch (object to boolean)	1
3	PUT virtualNetworks	addressPrefixes	Type mismatch (array to boolean)	1
4	PUT virtualNetworks	addressPrefixes_array	Type mismatch (object to boolean)	1
5	PUT virtualNetworks	subnets_name	Type mismatch (string to integer)	1
6	PUT virtualNetworks	subnets_addressPrefix	Type mismatch (array to boolean)	1
7	PUT virtualNetworks	subnets_delegations_name	Type mismatch (string to boolean)	1
8	PUT virtualNetworks	subnets_delegations_properties	Type mismatch (array to boolean)	1
9	PUT virtualNetworks	subnets_delegations	Missing node (e.g., properties)	1
10	PUT virtualNetworks	subnets_delegations_properties	Missing node (e.g., "properties":{})	7

for PATCH DNS-Record-Set were similar to bugs 2, 3 and 4 for PUT DNS-Record-Set, respectively. Note that these two requests have the same body schema, which would explain similar bugs can be found in both PUT and PATCH requests for DNS-Record-Set.

For DNS, we fuzzed API version 2017-10-01. This API has 13 requests, including 4 requests with a non-empty body whose schema size n is 2, 22, 65 and 65, respectively. We found Internal Server Errors in 3 out of these 4 requests; the 4th request PATCH DNS-Zone has only a small body schema with 2 nodes. During this fuzzing session, 202 different error types were found. In other words, beside the one error-type "500/Internal-Server-Error", there were 201 other error types returned during these 14,964 tests, for a total of 202.

6.4 Azure Networking Bugs Found

Table 2 summarizes the bugs found while fuzzing the bodies of the 49 (out of 371) Azure networking requests with non-empty body schemas. Overall, in 203 minutes of fuzzing, we generated 47,434 tests (API requests) and found 273 instances of Internal Server Error bugs. After triaging these[4], we found 43 variants of 10 distinct bugs, which are briefly described in Table 2 and which we reported to Azure developers. Bug 1 is similar to Bug 1 of Table 1: replacing a valid location value by an empty array or a boolean value. triggers an Internal Server Error. This bug can actually be found using many other requests which take a location value in their bodies, and all have a common root cause (as confirmed by Azure developers). Bugs 2 to 10 were found while fuzzing the very complex body of the PUT virtualNetworks request. Bugs 2 to 8 are found while fuzzing the type of various parts of the schema of this request. In contrast, bugs 9 and 10 are found using structural fuzzing rules and removing two specific parts of the schema of that request. The 7 variants of bug 10 are obtained by dropping various parts of the properties sub-tree of a subnets_delegations, but we grouped all these together for the sake of brevity.

For the Azure networking API, we fuzzed API version 2019-04-01. This API has 371 requests, including 49 requests with a non-empty body whose schema size n varies from 2 to 1,356 nodes. We found Internal Server Errors across 29 of these 49 request types. Note that 11 of the remaining 20 requests where we did not find any bug have simple bodies with schemas of less than 10 nodes. During this fuzzing session, 397 different error types were found.

As can be seen from Table 2, most of the bugs were found when fuzzing the request PUT virtualNetworks, which has a large body schema with 521 nodes. However, fuzzing even larger body schemas, such as the body of the PUT request for networkInterfaces with 1,247 nodes, did not find any new bug there, except for another variant of bug 1.

6.5 Discussion

The number of tests executed per minute was slower with the DNS service than with the Azure networking services. We repeated the above experiments several times, but the overall results (i.e., the number of bugs found) did not vary significantly. As we progressively designed and evaluated the algorithms discussed in Sections 3 to 5, we started finding our first Internal Server Error bugs in DNS only after introducing pipelines: without pipelines, we would not have found any of the bugs of Tables 1 and 2. Note that *RESTler* as-is (i.e., without our new extension) did not find any of these bugs either. As can be seen from these two tables, the TYPE fuzzing rule was key to find many of these 17 bugs in conjunction with structural fuzzing rules, which alone also found several other bugs. In contrast, the DUPLICATE fuzzing rule, which may generate malformed JSON

[4] Our tool automatically generates information as shown in columns 2-5 of Table 2 and triaging took less than one hour.

request bodies, did not trigger any `Internal Server Error` bug in our experiments; however, it did trigger two other bugs in DNS where the response format was erroneously set to XML instead of JSON, and did not follow the response schema defined in the Swagger specification.

We reported all the bugs we found to Azure service developers. All of them are easily reproducible (deterministic) and have been acknowledged. We have been told that *all* of these bugs will be fixed, and most of them have been fixed already. Indeed, `Internal Server Error` bugs are unhandled exceptions, which may cause severe service back-end state corruptions or leaks. It is safer to fix these bugs rather than risk a live incident or outage with unknown consequences.

7 RELATED WORK

Test Authoring Tools for REST APIs. With the recent explosion of web and cloud services, testing has been applied to REST APIs as well, mostly in commercial tools, such as Postman [2], SoapUI [3], and others [1, 4–6, 30]. These tools enable the automatic execution of tests *manually written* by developers (like JUnit does in Java).

Test Generation Tools for REST APIs. Other tools available for testing REST APIs *generate new tests* starting from manually-defined or previously recorded API traffic, and then by fuzzing and replaying new traffic to try finding bugs [12, 13, 16, 17, 20, 21, 29, 33, 36, 37]. Some of these can leverage REST API Swagger specifications and then fuzz HTTP requests using either pre-defined heuristics [13, 20, 21, 29, 36] or user-defined rules [16, 33]. Other extensions are possible if the service code can be instrumented [14] or if a functional specification is available [31]. *RESTler* [15] is a recent tool that performs a lightweight static analysis of a Swagger specification in order to infer dependencies among request types, and then automatically generates *sequences* of requests (instead of single requests) in order to exercise the service behind the API more deeply, in a *stateful manner*, and *without pre-recorded API traffic*. All these tools can find bugs like `500 Internal Server Errors`, but they do not fuzz body payloads *intelligently* using JSON body schemas and fuzzing rules as in this paper.

Grammar-based Fuzzing. The general-purpose grammar-based fuzzers, like Peach [28] and SPIKE [32] among others [33, 34], 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 specifying what and how to fuzz. For structured input formats like XML dialects, automatic XML-aware fuzzers parse the high-level tree structure of an input and include custom fuzzing rules (like reordering child nodes, increasing their number, etc.) that will challenge the application logic while still generating syntactically-correct XML [34]. Our JSON schema fuzzing rules are inspired by related XML-fuzzing rules which have successfully found new bugs in XML-based file parsers, such as Microsoft Office XML-based formats (docx, xlsx, etc.) [22]. Compared to this prior work, our main contributions are (1) to adapt such rules to the context of REST API JSON data (Sections 3 to 5), (2) to conduct a detailed evaluation of these rules, and (3) to demonstrate their effectiveness in the REST API context by finding new bugs in mature cloud services (Section 6). Unlike prior off-line grammar-based fuzzing, we also leverage *dynamic feedback* extracted from service responses in order to *learn* new state-dependent data values and then *refine* body payloads embedded in subsequent service requests.

API Attacks Based on Feedback from Responses. In our implementation, extra properties returned in responses that are not specified in the (request or response) schema are simply ignored. However, detecting and using such extra properties is relevant for *mass assignment* vulnerabilities. A mass assignment vulnerability may be present when JSON data is bound to service-internal data structures without proper filtering [7, 9]. For example, an attacker may gain unauthorized access to parts of an API by observing an `is_admin` or `is_debug` property in a response, then discovering requests that allow setting it to `true` without having proper permissions. In future work, we plan to explore how to extend our implementation to check for such vulnerabilities.

8 CONCLUSION

This paper has 5 main contributions and takeaways.

(1) We demonstrate that REST API data processing bugs do exist in widely-used cloud services.
(2) We show that such bugs are not that difficult to find provided *intelligent* data fuzzing techniques are used.
(3) We describe several such data fuzzing techniques and evaluate their effectiveness in the REST API context.
(4) Cloud service developers do not already use these techniques (otherwise they would have found already the bugs we reported in this paper).
(5) Cloud service developers care about these bugs (otherwise they would not fix them).

To the best of our knowledge, this paper is the first to highlight points 1-3 above. To further validate points 4 and 5, more REST API data fuzzing experiments are needed with more REST APIs and services.

Update. Since the first writing of this paper, we have fuzzed several other large Azure services over the past few months, and we have found over 100 new unique bugs in these so far (for a total of thousands of crashes). We have reported these bugs to Azure developers, and they are currently being reviewed and fixed.

ACKNOWLEDGMENTS

We thank Anton Evseev, Mikhail Triakhov, and Natalia Varava from the Microsoft Azure Networking team for their comments on the results of Section 6. We also thank the Azure developers that confirmed and fixed the new bugs reported in this work. More broadly, we thank Albert Greenberg, Mark Russinovich, John Walton, and Craig Wittenberg, for encouraging us to pursue this line of research.

REFERENCES

[1] [n. d.]. Apigee Docs. https://docs.apigee.com/ Last accessed 2019-11-22.
[2] [n. d.]. Postman | API Development Environment. https://www.getpostman.com/ Last accessed 2019-11-22.
[3] [n. d.]. SoapUI. https://www.soapui.org/ Last accessed 2019-11-22.
[4] [n. d.]. vREST – Automated REST API Testing Tool. https://vrest.io/ Last accessed 2019-11-22.
[5] 2019. APIFortress. http://apifortress.com Last accessed 2019-11-22.
[6] 2019. HttpMaster. http://www.httpmaster.net Last accessed 2019-11-22.

[7] 2019. Mass Assignment Cheat Sheet. https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Mass_Assignment_Cheat_Sheet.md Last accessed 2019-11-22.

[8] 2019. Microsoft Azure DNS Service Documentation. https://docs.microsoft.com/en-us/azure/dns/ Last accessed 2019-11-22.

[9] 2019. OWASP API Security. https://www.owasp.org/index.php/OWASP_API_Security_Project Last accessed 2019-11-22.

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

[11] Amazon. 2019. Amazon Web Services (AWS) - Cloud Computing Services. https://aws.amazon.com/ Last accessed 2019-11-22.

[12] APIFuzzer [n. d.]. APIFuzzer. https://github.com/KissPeter/APIFuzzer.

[13] AppSpider [n. d.]. AppSpider. https://www.rapid7.com/products/appspider.

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

[15] V. Atlidakis, P. Godefroid, and M. Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 748–758. https://doi.org/10.1109/ICSE.2019.00083

[16] Boofuzz [n. d.]. BooFuzz. https://github.com/jtpereyda/boofuzz.

[17] Burp [n. d.]. Burp Suite. https://portswigger.net/burp.

[18] Roy T. Fielding. 2000. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Doctoral dissertation.

[19] J. E. Forrester and B. P. Miller. 2000. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*. Seattle.

[20] fuzz-lightyear [n. d.]. Fuzz-Lightyear. https://github.com/Yelp/fuzz-lightyear.

[21] Fuzzy-Swagger [n. d.]. Fuzzy-Swagger. https://github.com/namuan/fuzzy-swagger.

[22] T. Gallagher, B. Jeffries, and L. Landauer. 2006. *Hunting Security Bugs*. Microsoft Press.

[23] P. Godefroid, M.Y. Levin, and D. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*. San Diego, 151–166.

[24] H. Liu, S. Lu, M. Musuvathi, and S. Nath. 2019. What Bugs Cause Production Cloud Incidents?. In *Proceedings of HotOS'2019*.

[25] Microsoft. 2019. Azure REST API Specifications. https://github.com/Azure/azure-rest-api-specs Last accessed 2019-11-22.

[26] Microsoft. 2019. Microsoft Azure Cloud Computing Platform & Services. https://azure.microsoft.com/en-us/ Last accessed 2019-11-22.

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

[28] Peach 2019. Peach Fuzzer. http://www.peachfuzzer.com/. Last accessed 2019-11-22.

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

[30] REST-assured 2019. REST Assured. http://rest-assured.io/. Last accessed 2019-11-22.

[31] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2018. Metamorphic Testing of RESTful Web APIs. *ACM Transactions on Software Engineering* 44, 11 (2018).

[32] SPIKE 2019. SPIKE Fuzzer. http://resources.infosecinstitute.com/fuzzer-automation-with-spike/. Last accessed 2019-11-22.

[33] Sulley [n. d.]. Sulley. https://github.com/OpenRCE/sulley.

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

[35] Swagger [n. d.]. Swagger. https://swagger.io/.

[36] Swagger-Fuzzer [n. d.]. Swagger-Fuzzer. https://github.com/Lothiraldan/swagger-fuzzer.

[37] TnT-Fuzzer [n. d.]. TnT-Fuzzer. https://github.com/Teebytes/TnT-Fuzzer.

[38] M. Zalewski. 2015. AFL (American Fuzzy Lop). http://lcamtuf.coredump.cx/afl/.