Differential Regression Testing for REST APIs

Patrice Godefroid
Microsoft Research
United States
pg@microsoft.com

Daniel Lehmann*
University of Stuttgart
Germany
mail@dlehmann.eu

Marina Polishchuk
Microsoft Research
United States
marinapo@microsoft.com

ABSTRACT

Cloud services are programmatically accessed through REST APIs. Since REST APIs are constantly evolving, an important problem is how to prevent breaking changes of APIs, while supporting several different versions. To find such breaking changes in an automated way, we introduce *differential regression testing* for REST APIs. Our approach is based on two observations. First, breaking changes in REST APIs involve two software components, namely the client and the service. As such, there are also two types of regressions: *regressions in the API specification*, i.e., in the contract between the client and the service, and *regressions in the service itself*, i.e., previously working requests are "broken" in later versions of the service. Finding both kinds of regressions involves testing along two dimensions: when the service changes and when the specification changes. Second, to detect such bugs automatically, we employ *differential testing*. That is, we compare the behavior of different versions on the same inputs against each other, and find regressions in the observed differences. For generating inputs (sequences of HTTP requests) to services, we use RESTler, a stateful fuzzer for REST APIs. Comparing the outputs (HTTP responses) of a cloud service involves several challenges, like abstracting over minor differences, handling out-of-order requests, and non-determinism. Differential regression testing across 17 different versions of the widely-used Azure networking APIs deployed between 2016 and 2019 detected 14 regressions in total, 5 of those in the official API specifications and 9 regressions in the services themselves.

CCS CONCEPTS

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

KEYWORDS

REST APIs, differential regression testing, service regression, specification regression, client/service version matrix

ACM Reference Format:
Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential Regression Testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397374

1 INTRODUCTION

Cloud computing is exploding. Today, most cloud services, such as those provided by Amazon Web Services (AWS) [8] and Microsoft Azure [37], are programmatically accessed through REST APIs [21], both by third-party applications [7] and other services [40]. Since cloud services and their REST APIs are constantly evolving, breaking changes between different versions of a particular API are a major problem [19, 30]. For instance, a recent study of 112 "high-severity Azure production incidents caused by software-bugs" points out that in many cases "these bugs are triggered by software updates" [32]. Our paper discusses a fundamental software-engineering problem: *How to find such regressions effectively through automatic testing for cloud services with published APIs?*

Today, there are two main techniques to detect regressions when updating a REST API. First, testing *before* deploying a new API version is used to send hand-written or previously-recorded requests to the new service and check for specific responses. This is a laborious task, even though supported by a variety of commercial testing tools [2–5]. Checking the new specification is often even less automated and more incomplete, e.g., if only done by visual inspection. The second technique, especially for critical services, is to roll-out in a staged manner, e.g., starting with early-adopter deployments, then to select regions or datacenters, and then to all regions of a public cloud. Service traffic is constantly monitored and if clients stop working, errors will appear in traffic logs, and customers will complain. But catching regressions *after* rolling out a new service version is *painful and expensive* because incident management involves complex tasks (incident triage, prioritization, root-cause analysis, remediation, bug fixes, rolling-out patches, etc.) and disrupts engineers both on the service-provider side and on the customer side. Customer-visible regressions reduce customer satisfaction, and may also have direct financial impact when SLAs (Service Level Agreements) are broken.

This paper introduces *differential regression testing for REST APIs* as an automated technique to detect breaking changes across API versions. A first key observation is that breaking changes in REST APIs involve *two* software components, namely the client and the service. As such, we observe that there are also two types of regressions: *regressions in the API specification*, i.e., in the contract between the client and the service, and *regressions in the service itself*, i.e., previously valid requests stop working in later versions of the service. Finding both kinds of regressions involves testing along two dimensions: when the service changes and when new clients are derived from the changed specification.

A second key idea is to use *differential testing* to detect specification and service regressions automatically. We compare the

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

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.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397374

behavior of different versions of client-service pairs, and find regressions in the observed differences. Making this approach practical involves overcoming several technical challenges. Given N API versions, there are N^2 client-service pairs, and N^4 ways to compare them with each other. Fortunately, we show that not all these combinations need be considered in practice. We also discuss how to compare the outputs (HTTP responses) of a cloud service that contain non-determinism, and how to handle out-of-order or newly appearing requests when updating the specification.

To evaluate the effectiveness of differential regression testing, we present a detailed historical analysis of 17 versions of Microsoft Azure networking APIs deployed between 2016 and 2019. All their API specifications are publicly available on GitHub [36], and all 17 service versions are still accessible to anyone with an Azure subscription. Our approach and tools detected 5 regressions in the official specifications and 9 regressions in the services themselves. We also discuss how these regressions were fixed in subsequent versions.

The main contributions of this paper are:

- We introduce *differential regression testing for REST APIs*, including the key notions of *service* and *specification* regressions. We discuss the computational complexity of finding such regressions across N API versions.
- We discuss *how* to effectively detect service and specification regressions by comparing network logs capturing REST API traffic, using *stateful* and *stateless* diffing techniques.
- We present a detailed *API history analysis* for Microsoft Azure Networking, a widely-used core Azure service composed of more than 30 APIs. We found 14 regressions across 17 versions of those APIs deployed between 2016 and 2019.

2 DIFFERENTIAL REGRESSION TESTING FOR REST APIS

2.1 Regression Testing and Differential Testing

Before we describe our approach for differential regression testing for REST APIs, we define here some common terminology used throughout the remainder of the paper.

A *regression* in a program is a bug which causes a feature that worked correctly to stop working after a certain event, such as a software update. An *update* is the process of changing the program from an old version n to a new version $n + 1$. Regressions are also called *breaking changes*. The notion of breaking change only makes sense when the software versions before and after the update are in a compatibility relation with each other. *Major versions* in semantic versioning [42] are the canonical example of versions without a compatibility relation, i.e., which allow breaking changes by design (and thus should occur as little as possible). We only test versions that *are* in a compatibility relation with each other. Usually, those allow for *backwards-compatible changes*: New functionality may be added, but program inputs that "worked correctly" should still behave the same, if given to the new version.

Regression testing typically consists of running manually written and over-the-years accumulated test cases that check that the new program does not crash and that assertions specified in the program or in the test harness are not violated. Note that (classical) regression testing does *not* involve any comparison of program outputs with outputs from another run: each test of a regression test suite either passes or fails, and any failure indicates a potential regression (assuming all tests pass in the previous version).

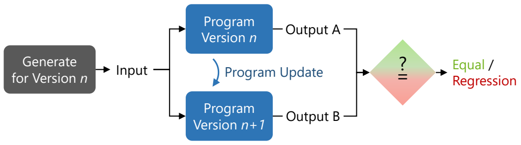


Figure 1: Differential regression testing for programs.

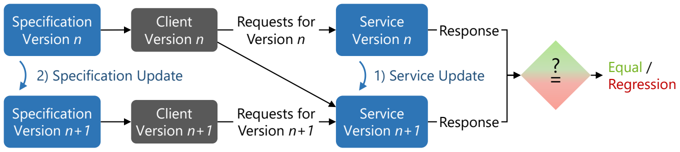


Figure 2: Differential regression testing for REST APIs involves two versions – for the specification *and* the service.

Differential testing means comparing the outputs of two different, but related programs on the same input [34]. This notion is independent of the notion of regression testing. Differential testing is a generic method to obtain a test oracle [13] in automatic testing, i.e., to decide for generated inputs, whether the output of the program under test is correct. For instance, differential testing can be used to find bugs in compilers by compiling randomly-generated C programs using different compilers (e.g., `clang` and `gcc`), and then comparing the runtime behavior of the binaries produced by each compiler against each other [48].

Differential regression testing uses differential testing to automatically find regressions, and has been applied previously to "traditional" programs, e.g., compilers [25]. For it to work, one needs two backwards-compatible versions n and $n + 1$ of the program under test and an input generator for said program. Differential regression testing then consists of the following steps, illustrated in Figure 1: both program versions n and $n + 1$ are executed with the same, generated input valid for program n; if the new version produces an output different from the previous version, a potential regression has been detected. We use differential regression testing for REST APIs. Below, we describe a crucial difference between traditional programs and REST APIs that presents additional challenges for differential regression testing.

2.2 Updates in REST APIs

A REST API is *not* a traditional program because it defines an *interface* between two software components: a *client*, which produces requests, and a *service*, which handles these requests and returns responses. Thus, while in traditional regression testing only a single program is updated, in the context of REST APIs, both the service and the clients are updated over time.

The contract between clients and the service is the API *specification*. It defines which requests clients may send, e.g., which HTTP methods (`GET`, `PUT`, etc.) are available for each endpoint (URI) and what is allowed in requests and responses. A common format for such specifications is Swagger (also known as OpenAPI) [46].

Developers write client code mostly based on this specification since the service itself is usually a black box (e.g., for commercial services, source code is not available). Given that the specification is authoritative information for client developers, it is crucial that it is correct. In many cases, whole clients are also automatically generated from specifications. For instance, the Swagger specifications of Microsoft Azure services are publicly available on GitHub [36], and software development kits (SDKs) for various languages like Python, JavaScript, or C# are auto-generated from them [6].

Because a REST API has two components – a client that is derived from a specification, and a service – Figure 2 illustrates that, unlike for regression testing of "traditional" programs, there is not just a single update happening, but in fact two orthogonal update steps:

- First, the *service* is updated to be able to handle added requests and functionality of the new API version. If the new API version is backwards-compatible with the old one, the service is also expected to handle existing requests correctly.
- Second, the updated *specification* is published so that developers know about the added functionality and can write new clients for the new service.

By publishing the updated specification, the service owner "commits" to the new API version – third parties can now write and generate their own clients to use the newly documented functionality. In other words, updating a REST API from a version n to a new version $n + 1$ generates *two* new component versions: a new service version $n + 1$, denoted s_{n+1}, and a new client version $n + 1$, denoted c_{n+1}, corresponding to the new API specification.

In order to test a service, it is necessary to exercise it with a client. In what follows, the pair (c_i, s_j) will denote testing the service s_j with a client c_i matching (or automatically generated from) specification version i. Figure 2 shows the three possible testing configurations between the client/service versions n and $n + 1$ of a REST API, which are each denoted by an edge in Figure 2:

- (c_n, s_n) corresponds to the vertical edge where inputs from old clients are sent to an old service version,
- (c_n, s_{n+1}) (the diagonal edge) is the state when the service is already updated, but the new specification not yet published, and
- (c_{n+1}, s_{n+1}) is the final configuration when both clients and service are updated.

The hypothetical configuration (c_{n+1}, s_n) is not considered here because a new client version c_{n+1} is usually not supposed to work with the old service version s_n (unless the service is fully *forward-compatible*, but this is unlikely in practice and therefore this case will not be considered further in this paper). From now on, we call a pair (c_i, s_j) a *testing configuration*.

2.3 Regressions in REST APIs

Given *two* new component versions, one for the specification and clients derived from it, and another version for the service, there are now *two possible types of breaking changes*:

(1) *Regressions in the service* occur when requests that worked with a previous version of the service are no longer accepted, or return different or wrong results.

(2) *Regressions in the specification* occur when previously-allowed requests or responses are removed (or modified) in the new specification in a way that breaks applications after switching to the new client version.

Therefore, in order to detect regressions of both types, *two types of differential regression testing* are required for REST APIs:

(1) Comparing the testing results of (c_n, s_n) with the results of (c_n, s_{n+1}) may detect *regressions in the service* (the client version c_n stays constant).

(2) Comparing the testing results of (c_n, s_{n+1}) with the results of (c_{n+1}, s_{n+1}) may detect *regressions in the specification* (the service version s_{n+1} stays constant).

This is what we call *differential regression testing for REST APIs*: how to compare the outputs of two different testing configurations in order to detect service or specification regressions in a REST API.

A typical example of *service regression* is when a request type suddently stops working (which is rare), or when the format of the response unexpectedly changes and is undocumented (or incorrectly documented), for instance removing, adding or renaming response properties (which is more frequent). Examples of *specification regressions* are when an optional property in the body of a request becomes required, or when an item in an enumerated list is removed from the specification.

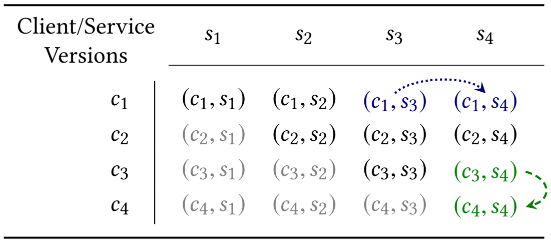
Since a specification and its service are closely related, one might think that regressions cannot appear independently in one or the other. However, in the context of cloud services, specifications and services are often authored and maintained by different people and written in different languages (e.g., the service can be written in C# and the specification in Swagger). So in practice, they can become inconsistent due to regressions in either of them.

Note that specification regressions may be *guessed* by *statically* comparing (diffing) specification descriptions. However, in order to *confirm* that such specification changes are either false alarms or actual regressions, it is *necessary to test* the new service with the old client in order to check whether the new service still supports the old functionality (now undocumented but perhaps still present for backwards compatibility), or is a true specification regression as defined above.

Do cloud services (like core Azure services) guarantee backwards compatibility, i.e., that the two types of regressions defined above should never occur? Strictly speaking no, but implicitly yes. Strictly speaking, clients are supposed to use the specification version $n + 1$ with the service version $n + 1$. But in practice, backwards compatibility is implicitly expected as API versions change frequently (e.g., for Azure ca. every month), and continuous support of previous functionality is expected to allow past clients (customers) to continue operating and evolving their own services [24].

2.4 Client/Service REST API Version Matrix

The updates to the client and service versions defined by a REST API can be visualized using a two-dimensional *client/service version matrix*, as shown in Figure 3. The columns of this matrix correspond to service versions, while the rows correspond to client versions, or more accurately, to the versions of the API specifications from which clients can be generated. The cell in the matrix at row i and column j corresponds to testing (c_i, s_j).



Client/Service Versions	s_1	s_2	s_3	s_4
c_1	(c_1, s_1)	(c_1, s_2)	(c_1, s_3)	(c_1, s_4)
c_2	(c_2, s_1)	(c_2, s_2)	(c_2, s_3)	(c_2, s_4)
c_3	(c_3, s_1)	(c_3, s_2)	(c_3, s_3)	(c_3, s_4)
c_4	(c_4, s_1)	(c_4, s_2)	(c_4, s_3)	(c_4, s_4)

Figure 3: Example of a REST API version matrix, client versions increasing row-wise and service versions increasing column-wise. The dotted blue arrow marks an example of a service update. The dashed green arrow marks an example of a client update (derived from a new specification). Client/service configurations that are not tested are grayed-out.

If N denotes the number of versions of an API, the client/service version matrix is of size N^2. However, some of these testing configurations do not make sense in our context: testing (c_i, s_j) when $i > j$ is not necessary as *newer client* versions are not supposed to work *with older service* versions. In terms of the version matrix, this means the matrix is an *upper triangular matrix*. This reduces the number of testing configurations from N^2 to $N * (N + 1)/2$ (i.e., the number of cells in a triangular matrix, including its diagonal).

2.5 Complexity of Differential Regression Testing

Each cell of a client/service version matrix corresponds to one client-service testing configuration. An automated testing approach with a non-differential test oracle runs one test for each cell in the matrix. E.g., *RESTler* [12] can test that no request sent by a clent c_i ever leads to a `500 Internal Server Error` response by the service s_j. But to get a more interesting test oracle, we are employing *differential testing*, and thus have to ask how many of the test configurations should be *compared* with each other in order to find all possible regressions in N versions of an API? In other words, how many *pairs of cells* should be compared to achieve this goal?

When updating an API version from version n to $n + 1$ for any $1 \leq n < N$, both service or specification regressions may be introduced, as defined in the previous subsection. Checking for service regressions by comparing the testing results of (c_n, s_n) with the results of (c_n, s_{n+1}) corresponds to a horizontal edge in the version matrix as indicated by the blue dotted arrow in Figure 3. Similarly, checking for specification regressions by comparing the testing results of (c_n, s_{n+1}) with the results of (c_{n+1}, s_{n+1}) corresponds to a vertical edge in the version matrix, highlighted as the green dashed arrow in Figure 3.

Instead of checking separately for service regressions and specification regressions, why not simply compare the testing results of (c_n, s_n) with (c_{n+1}, s_{n+1}) directly? (Such comparisons would correspond to diagonal edges between cells in the version matrix.) The reason is that such comparisons would be harder, less accurate, and could miss detecting regressions. For example, if a request X is renamed to Y in version $n+1$, both (c_n, s_n) using X and (c_{n+1}, s_{n+1}) using Y may work fine (no errors), but automatically detecting that X has been renamed Y with high confidence (no false alarms) is a harder problem. In contrast, leaving the client version c_n or the service version s_{n+1} constant avoids this "generalized mapping" problem, and facilitates the detection of service or specification regressions, respectively.

By generalization of the previous argument, it is also unecessary to perform differential testing on any non-adjacent pairs of testing configurations in the version matrix: for any $i' > i$ and $j' > j$, it is not necesary to compare the testing results of (c_i, s_j) with $(c_{i'}, s_{j'})$. Indeed, any service regression between the old (c_i, s_j) and the new $(c_{i'}, s_{j'})$ will be found when differential testing of some (c_i, s_n) and (c_i, s_{n+1}) with $j \leq n < j'$. Similarly, any specification regression between the old (c_i, s_j) and the new $(c_{i'}, s_{j'})$ will be found when differential testing of some $(c_n, s_{j'})$ and $(c_{n+1}, s_{j'})$ with $i \leq n < i'$. In other words, comparisons corresponding to *transitive edges* need not be considered in the version matrix: comparing only *atomic updates* is sufficient.

Consequently, for any testing configuration (c_i, s_j) in the version matrix, we need to compare it to *at most* its two adjacent neighbors along the two dimensions, namely (c_i, s_{j+1}) (right neighbor) and (c_{i+1}, s_j) (bottom neighbor) if $i < j$.

If we want to cover all possible $N * (N + 1)/2$ client-service (c_i, s_j) testing configurations (i.e., all the cells in the upper triangular version matrix including the main diagonal), the number of differential testing comparisons we need to perform is therefore:

- For every testing configuration on the main diagonal of the version matrix, one can only compare its output to the right neighbor (since the bottom neighbor is an invalid configuration of old service and new client): N comparisons.
- For every testing configuration in the rightmost column, one can only compare to bottom neighbors (since there are no more recent service versions to the right): N comparisons.
- For the bottom right corner (most recent client and service version), we performed two unnecessary comparisons: -2.
- For all other "interior cells": 2 comparisons (one to the right, where the service is updated, one down, where the client is updated): $\frac{(N-1)(N-2)}{2}$ cells times 2 comparisons per cell.
- This gives a total number of comparisons for differential testing: $N + N - 2 + \frac{(N-1)(N-2)}{2} * 2 = N * (N - 1)$

In summary, given an API with N versions, there are $T(N) = \frac{N*(N+1)}{2}$ possible client/service testing configurations, and finding regressions in all possible service and specification updates requires $D(N) = N * (N - 1)$ comparisons (or diffs) between the outputs of those $T(N)$ client/service configurations.

2.6 Diagonal-Only Strategy

Additionally, we also propose a simpler and less expensive strategy for finding service and specification regressions:

- For every new version $n + 1$, only two new testing configurations have to be run: (c_n, s_{n+1}) and (c_{n+1}, s_{n+1}).
- Between the original configuration (c_n, s_n) and the two new configurations, perform only two comparisons: compare (c_n, s_n) with (c_n, s_{n+1}) (to check for service regressions) and compare (c_n, s_{n+1}) with (c_{n+1}, s_{n+1}) (to check for specification regressions).

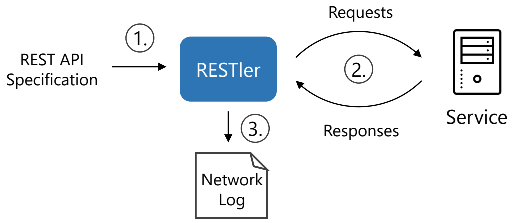


Figure 4: RESTler overview.

In other words, this strategy corresponds to only testing the configurations corresponding to the diagonal cells (c_{n+1}, s_{n+1}) of the version matrix, plus all the cells (c_n, s_{n+1}) immediately above these. Let us therefore call this strategy the *diagonal-only strategy*.

Given N API versions, the advantage of this strategy is that it requires only $T_{\text{diag}}(N) = 2 * (N - 1)$ client-service test combinations, and only $D_{\text{diag}}(N) = 2 * (N - 1)$ comparisons (diffs) among these. In theory, *assuming* services and specifications are backward-compatible, this strategy is as accurate as the upper-triangular matrix strategy, by transitivity of backward-compatibility, *if also assuming* a fixed deterministic test-generation algorithm from specifications, and *assuming* services responses are deterministic. However, if any of these three assumptions are not satisfied, the diagonal-only strategy provides less coverage among all the possible client-service version pairs, and therefore could miss more regressions compared to the upper-triangular matrix strategy, simply because it runs fewer tests.

In practice, is the diagonal-only strategy sufficient to find many, or even all the service and specification regressions that are found by the upper-triangular matrix strategy? In Section 4, we will empirically answer this question for 17 versions of a large complex API of a core Microsoft Azure service. But first, we need to define how to exercise a client version c_i with a service version s_j, and then how to compare testing results obtained with various such combinations. These are the topics discussed in the next section.

3 TECHNICAL CHALLENGES

3.1 From Specifications to Tests

In order to automatically generate client code from a REST API specification and comprehensively exercise a service under test, we use a recent automatic test generation tool *RESTler* [12]. As shown in Figure 4, *RESTler* generates and executes sequences of the requests defined in the API specification, while recording all requests and responses in a *network log*. *RESTler* supports the Swagger (recently renamed OpenAPI) [46] 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. Most Azure services have public Swagger API specifications available on GitHub [36]. A test suite automatically generated by *RESTler* attempts to cover as much as possible of the input Swagger specification, but full coverage is not guaranteed. This limitation is acceptable because the purpose of this work is *only* to detect specification or service regressions through automated testing, *not to prove* the absence of regressions (verification).

```
[ [ { // a pair of sent request and received response
  "request": {
    "method": "PUT",
    "uri": ["blog", "post-42"],
    "queryString": { "api-version": "2017" },
    "headers": {
      "Authorization": "token-afe2d391031afe...",
      "Content-Type": "application/json",
      ... }, // more headers
    "body": { // body as JSON, not plain string
      "title": "My first blogpost", // property-1
      "content": "..." } //property-2
  },
  "response": {
    "status": "201 Created",
    "headers": {
      "Content-Length": "566",
      "Date": "Wed, 31 Jul 2019 18:00:00 GMT",
      "Server": "Apache/2.4.1 (Unix)",
      "x-request-id": "fae12396-4557-4755-bd1b-7380d40d033c",
      ... },
    "body": {
      "id": "/blog/2019-07-31/post-42", //property-3
      ... }
  },
  ... // more requests and responses for this sequence
} ],
... // more sequences of requests and responses
]
```

Figure 5: Example of a network log.

$$\begin{aligned}
\textit{log} &::= \textit{requestSequence}^{*}\\
\textit{requestSequence} &::= \textit{requestResponsePair}^{*}\\
\textit{requestResponsePair} &::= (\textit{request}, \textit{response}^{?})\\[4pt]
\textit{request} &::= \textit{method}, \textit{uri}, \textit{headers}, \textit{body}\\
\textit{response} &::= \textit{statusCode}, \textit{statusDescription}, \textit{headers}, \textit{body}\\[4pt]
\textit{method} &::= \texttt{GET} \mid \texttt{PUT} \mid \texttt{DELETE} \mid \texttt{PATCH} \mid \dots\\
\textit{uri} &::= \textit{path}, \textit{queryString}\\
\textit{path} &::= \textit{pathComponent}/*\\
\textit{queryString} &::= (\textit{key}, \textit{value})^{*}\\[4pt]
\textit{statusCode} &::= \texttt{200} \mid \texttt{201} \mid \dots \mid \texttt{400} \mid \dots \mid \texttt{500}\\
\textit{statusDescription} &::= \texttt{OK} \mid \texttt{Created} \mid \dots \mid \texttt{Bad Request} \mid \dots\\
&\quad\ \mid \texttt{Internal Server Error}\\[4pt]
\textit{headers} &::= (\textit{key}, \textit{value})^{*}\\
\textit{key}, \textit{value}, \textit{pathComponent} &\in \text{strings}\\
\textit{body} &\in \text{JSON values}
\end{aligned}$$

Figure 6: Abstract syntax of network logs. Common symbols: * means zero or more repetitions, ? is zero or one.

Since we use *RESTler* as a building block in our approach, this limitation is orthogonal to our approach. Future improvements to RESTler that exercise services more thoroughly could potentially lead to more regressions found by our approach as well.

3.2 Network Logs

The network logs considered in this work are standard, and are not *RESTler* specific. REST API traffic consists of sequences of request-response pairs. An example of a request and its response are shown in Figure 5 in the JSON format. Each request consists of a *method*, a *uri*, *headers*, and a *body*. Each response consists of the HTTP *status code and description*, *headers*, and a *body*. Figure 6 shows the abstract syntax of network logs capturing REST API traffic.

Resources managed through a REST API typically need to be created first (with a PUT or POST request), then can be accessed (with a GET request) or updated (with a PATCH or POST request), and then deleted (with a DELETE request). Some resources require a parent resource in order to be created. For example, the request PUT /blog/post-42?api-version=2017 shown in Figure 5 creates a new *blog post* with id post-42; the response includes 201 Created, which means the new resource has been successfully created; only then a request, for example, PUT /blog/post-42/comment/3?api-version=2017, may be executed to add a comment (child resource) associated with that blog post (parent resource). *RESTler* is *stateful*: it can automatically generate such *sequences* of causally-related requests in order to exercise "deeper" parts of an API specification (like the comment part of the API in the above example).

We define a network log as a sequence of sequences of request-response pairs. Each subsequence represents a "testing session" of usually-related requests with their responses. Note that the structure of HTTP requests and responses is explicitly represented in JSON, e.g., headers are a map from names to values, and URL paths are a sequence of path components. Also note that if the body is of type application/json (as is common for REST APIs), it is saved in the log a structured manner and not just as a plain string.

3.3 Diffing of Network Logs

A naïve implementation for comparing two network logs might use an off-the-shelf textual diffing tool or structural diffing tool such as json-diff [23]. There are several reasons why this approach does not work well for identifying potential regressions in network logs:

First, not every part of the exchanged HTTP messages is relevant for testing the behavior of a cloud service. For example, HTTP header fields such as Authorization are expected to contain different values in each log, since authentication tokens are refreshed at the start of each test run. Another example is the Content-Length field of the response, which can be ignored because it adds only noise to the comparison. (The length of a body will be different only if there already was a difference in the contents.)

Second, requests automatically generated by *RESTler* may contain randomly chosen concrete values, which do not indicate service behavior changes. For example, unique names are generated when creating resources, and those names can also appear in responses.

Third, some values in returned responses are non-deterministic values generated by the service, which also do not indicate a difference in behavior. Examples of such values that are out of the control of the client are the Date response header, unique IDs for requests and responses (used for debugging in case of a service error) such as the x-request-id or Etag headers, or service dependent dynamic values. (E.g., for a DNS service API, the exact nameservers returned for a domain can vary for load balancing purposes).

Thus, prior to performing regression diffing, we perform an *abstraction* of the network logs that excludes such values from the subsequent diff. Abstractions work by matching certain values in the network logs against a regular expression and replacing them by (fresh) constants, in order to equate all of their (possibly different, but irrelevant) values. The abstractions we apply are:

- Replace the values of the Date, Etag, and x-*-request-id headers by constant strings.
- Replace IDs that were randomly generated by *RESTler* with constant strings, e.g., post-42 and post-23 both match the regular expression post-\d+ and would be replaced by post-id.
- A set of service-specific replacements, e.g., non-deterministic name servers, UUIDs, etc.

Abstractions are specified by a single command-line parameter, so effort is minimal and done once per API.

Recursive Comparisons. After having applied abstractions, we compare two network logs by recursively comparing their subelements. To start with, we compare the first sequence of requests in the first network log with the first sequence of requests in the second network log. For sequences of strings, we use an algorithm (such as Myers diff [38]) that minimizes the number of edits between two such sequences. In the general case (i.e., diffing two sequences of requests), we do not minimize the size of our diff. Instead, if the first sequence is shorter than the second sequence, we regard elements after the common prefix as insertions, whereas for the case where the first sequence is longer than the second sequence, we regard those elements as deleted. We leave further minimizing the diffs, e.g., by employing tree diffing algorithms like in the GumTree tool [20] to future work.

We then recursively apply this comparison in a bottom-up fashion on the structured network logs to obtain a full network log diff, which has the same structure as a single network log, only with intermediate *insert*, *delete*, and *edit* nodes, at all subtrees where the two network logs were not equal.

3.4 Stateless Diffing of Network Logs

The diffing approach of the previous section enables a precise analysis of all differences in network logs from two testing configurations. However, even after applying the above abstractions, the following problems remain that cause irrelevant differences to be shown:

- *Sensitivity to ordering and deletions/insertions in the specification.* Changes in the specification may produce a difference in the sequences of requests generated by *RESTler*, which follows the ordering of the specification. For example, if a new request is added, comparing the sequences in order will produce many false positives, since the request sequences will be "out of sync".
- *Non-determinism due to the global state of the service.* During exercising a service, many resources are created, which are asynchronously garbage collected in order to prevent exceeding resource limits. Due to timing differences between multiple runs, a GET request that, e.g., lists all blog posts, may return a different set of blog posts each time it is invoked (depending on how many have been deleted).

Although the above problems can be solved by further transformations of the precise diffs, our analysis of these diffs (see Section 4) motivated us to introduce a more compact format, focused on maximizing the relevant differences shown, to be able to more quickly discover and confirm a subset of regressions. We call this approach *stateless diffing* of network logs, because each request-response pair is considered in isolation, without taking into account the full

```
  PUT /[...]/virtualNetworks/virtualNetworkName
   -  Request hash=abe64568e3c0b694c7413c6df808cf4c
   +  Request hash=b2b7584635adccd0786cbeb240f4b867
      Responses:
   -    201 Created
   +    400 Bad Request
 -DELETE /[...]/virtualNetworks/virtualNetworkName
```

Figure 7: Stateless diff example.

sequence of requests executed in the test run prior to the request. Specifically, stateless diffing transforms the network logs as follows:

- Group request-response pairs into equivalence classes, instead of the original ordered sequences.
- Consider only the response's status, instead of the full content of its response.
- Do not show *added* requests and responses in the diff, since those are backwards-compatible feature additions of a specification and service.

Expanding on the first point, stateless diffing first groups together all requests of a particular type. A *request type* consists of the request's HTTP method, the abstracted path, and a hash of its headers and body. As an example, a request like `GET /blog/post-42 <headers...> <body...>` has the following tuple as its type: `GET`, `/blog/post-id` (as described earlier, our abstractions replace the random id 42 with a constant), and *hash(*headers||body*)*. Each unique request type can correspond to many actual requests, because (1) the same request can be sent multiple times in different sequences, and (2) because abstractions map different concrete requests to the same abstracted one. Under each request type, we then list all the corresponding responses to those requests. For responses, to exclude any global state, we include only the response status code and description.

Figure 7 gives an example of a stateless diff. The first line states that a `PUT` request to the `virtualNetworks` API with the endpoint `virtualNetworkName` is present in both network logs (shown in black). The second and third line indicate that the full requests, including their bodies (shown by the hashes) changed, e.g., because a change in the specification added or removed properties in the request body. As a result, the server responds with `400 Bad Request` after the update instead of `201 Created`, a potential regression. Finally, in the last line, we also see that a `DELETE` request is missing in the network log after the update. In this case, the `DELETE` request is missing, because the resource to be deleted could not be created by the earlier `PUT`.

Stateful and stateless diffing will be experimentally evaluated in Section 4. In total, their implementation required about 2,200 lines of F# code.

4 EVALUATION

4.1 Tested REST APIs and Versions

To evaluate our approach, we regression-tested REST APIs of widely used Microsoft Azure cloud services related to networking. They are used, for example, to allocate IP addresses and domain names, or to provide higher level infrastructure, such as load balancers and firewalls.[1] Their Swagger specifications are available on GitHub [36].

Table 1: Overview of the specifications of the tested APIs.

Specifications	# Versions		Last Version		Sum Tested Versions	
in Directory	Total	Tested	# Files	# Lines	# Files	# Lines
`dns/`	4	3	60	4,142	129	9,585
`network/`	23	17	435	58,419	4,204	613,953
Sum			495	62,561	4,333	623,538

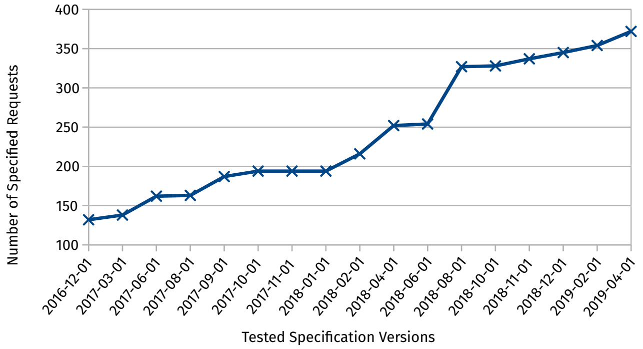


Figure 8: Evolution of the Network APIs over time in terms of specified requests.

Concretely, we looked at a collection of over 30 different REST APIs, split into two groups. First, the individual API of the Azure DNS service (in the GitHub repository under `specification/dns/`) and second, a large collection of APIs related to other networking services (jointly specified in `specification/network/`). We will refer to the first individual API as "DNS API" and to the large second group as "Network APIs" throughout the rest of the evaluation.

Since counting APIs is somewhat arbitrary, Table 1 gives an impression of the size of the tested APIs in more well-defined measures. The DNS API has 4 versions, where the most recent version is specified in 60 files containing 4,142 lines of JSON in total. The Network APIs are a substantially larger set, having 23 versions over time, with the latest version alone having a Swagger specification that spreads over 435 files with 58,419 lines of JSON. A large fraction of the specifications are examples of concrete requests (for the latest Network APIs: 395 files with 24,267 lines in total, so roughly 40% in terms of lines). Those examples are an important part of the specifications since (1) documentation is generated from them and (2) developers often use examples in their very first interaction with the service, so their correctness is important to service adoption.

Of all API versions, we selected those where the specification contained examples, which was the case for 3 versions (out of 4) for Azure DNS and 17 (out of 23) versions for the Network APIs. We selected those for two reasons: they are the most-recent ones (only the oldest versions do not include examples), and *RESTler* generates better test suites when given a set of examples to start from.

Many Azure APIs, including the ones we tested, are versioned by date (e.g., a version is `2019-04-01`). A new version of the DNS and

[1] https://azure.microsoft.com/en-us/product-categories/networking/

Table 2: Overview of regressions found in Azure DNS and Network REST APIs.

#	API Endpoint	Description	Correct vs. Regressed		Status
Regressions found with *stateless diffing* across *different specification* versions ("↕ direction"):					
1	`publicIPAddress`	Property in example incorrectly moved	2018-06-01	2018-07-01	Example corrected in 2018-08-01
2	`publicIPAddress`	Property in example incorrectly moved	2018-10-01	2018-11-01	"Re-regression" of bug above, corrected again in 2018-12-01
3	`virtualNetworks`	Property in example incorrectly moved	2018-10-01	2018-11-01	Example corrected in 2018-12-01
4	`networkSecurityGroups`	Required property in example removed	2018-10-01	2018-11-01	Example corrected in 2018-12-01
5	`interfaceEndpoints`	Whole endpoint removed / renamed	2019-02-01	2019-04-01	Confirmed, but deemed acceptable
Regressions found with *stateful diffing* across *different service* versions ("↔ direction"):					
6	`dnsZones`	Unspecified property `zoneType` in service response	2017-09-01	2017-10-01	Property specified in 2018-05-01
7	`publicIPAddress`	Unspecified property `ipTags`	2017-09-01	2017-10-01	Property specified in 2017-11-01
8	`loadBalancers`	Unspecified property `enableDestinationServiceEndpoint`	2017-10-01	2017-11-01	Property never specified
9	`loadBalancers`	Unspecified property `allowBackendPortConflict`	2018-10-01	2018-11-01	Property never specified
10	`virtualNetworks`	Unspecified property `enableDdosProtection`	2017-06-01	2017-08-01	Property specified in 2017-09-01
11	`virtualNetworks`	Unspecified property `enableVmProtection`	2017-06-01	2017-08-01	Property specified in 2017-09-01
12	`virtualNetworks`	Unspecified property `delegations`	2018-02-01	2018-04-01	Property specified in 2018-08-01
13	`virtualNetworks`	Unspecified property `privateEndpointNetworkPolicies`	2019-02-01	2019-04-01	Property specified in later commit
14	`virtualNetworks`	Unspecified property `privateLinkServiceNetworkPolicies`	2019-02-01	2019-04-01	Property specified in later commit

Network APIs is released approximately every two months (median interval: 61 days). Given this update frequency, it is crucial that new versions are backwards-compatible with previous versions, otherwise developers could not keep up with such frequent changes. Figure 8 shows the 17 versions we tested on the x-axis, from an early version `2016-12-01` up to the most recent version (at the time of testing) of `2019-04-01`. It also shows the number of specified requests (i.e., HTTP method and endpoint) at each version, which grows from 132 to 372 over time. The large number of versions especially of the Network APIs lends itself well to historical analysis we perform in the following, i.e., finding regressions not only in the most recent update, but also in updates between earlier versions.

In total, we tested versions of REST APIs that sum up to more than 600,000 lines of Swagger specifications. This does not include the many more lines of code needed to implement these services, since we view the service implementations as black boxes. To keep up with these changes and find regressions in this amount of data, developers clearly need tooling, such as our approach.

4.2 Experimental Setup

To test the full upper-triangular API-version matrix as described in Section 2, we first download the mentioned specification versions. For each specification version, we then generate a client with *RESTler*. This client automatically sends requests conforming to said specification. Each generated client (one per specification version) is run multiple times, each time targeting a service of the same or a higher API version (by setting the `api-version` query parameter). Each test run (one combination of specification and service version) produces a network log. In total, testing $N = 3$ versions of the DNS API results in $T(N) = \frac{N(N+1)}{2} = 6$ network logs and testing 17 versions of the Network APIs results in 153 network logs.

For *differential* regression testing, we then diff each pair of network logs that correspond to adjacent cells in the client-server version matrix (i.e., where either the client or the service was updated by one version). This gives $D(N) = N(N-1) = 6$ diffs for testing the DNS API and 272 diffs for the Network APIs. For service updates along the horizontal direction, we used the more precise stateful diffing, for updates of the underlying specification, we use stateless diffing. We substantiate this choice in Section 4.5. Manual inspection of the produced diffs resulted in 14 found regressions, which we discuss next.

4.3 Found Regressions – Overview

Table 2 gives an overview of the regressions we found in the tested DNS and Network APIs. In total, we discovered 14 unique regressions by differential testing. 5 were found by stateless diffing across different specification versions, i.e., when we updated the specification and generated a new client from it, but targeted the same service version. In these cases, the *Correct vs. Regressed* column shows the two versions of the specification update that introduced the regression. 9 regressions were found by stateful diffing across different service versions. In those cases, *Correct vs. Regressed* shows the service update, and not the specification, since the latter remains constant. The presence of both types of regressions shows that both directions of diffing in the version matrix are necessary. We also see from the *API Endpoint* column (the last path component of the URI) that we found regressions in different APIs end endpoints.

In the *Status* column, we show a short assessment of the bugs. In all but two cases (regressions #8 and #9), later versions of the API specification include fixes for the found regression, or other evidence (regression #5) is available to confirm the regression; these regressions were thus eventually found by the Azure service owners *after deployment.*

4.4 Found Regressions – Examples

We now discuss in more detail examples of regressions we found and why it is important to find them, ideally *before deployment.*

Regressions #1 to #4: Broken Example Payloads. Errors in examples – from which documentation is generated and that illustrate

```
{
  "parameters": {
    ...
    "publicIpAddressName":
      "test-ip",
    "parameters": {
      "location": "eastus"
    }
  }
}
```

Correct versions: 2018-06-01 and 2018-10-01.

```
{
  "parameters": {
    ...
    "publicIpAddressName":
      "test-ip",
    "parameters": {},
    "location": "eastus"
  }
}
```

Regressed version 2018-07-01 and re-regressed version 2018-11-01.

Figure 9: Regression in the examples of the specification for `publicIpAddress`. The erroneous change is highlighted.

how to use an API – are frustrating for users because they are typically the first attempt to use the service. In specification version 2018-07-01, a specification author erroneously moved the `location` property one level up in an example request for the `publicIpAddress` API (regression #1), as shown in Figure 9. This was caught by stateless diffing, because both network logs contain the request, but the response was different: `200 OK` in the old version, but `400 Bad Request` in the regressed version. In the next specification version, 2018-08-01, the change was reverted and the bug fixed. Interestingly though, the example was regressed again in version 2018-11-01 (regression #2). We can see how the nesting of two `parameters` properties might have led to the confusion. From the double regression we can also take that tooling such as described in this paper would help developers catch specification regressions prior to release. Even more interestingly, our tool also found similar regressions in examples for other APIs (regressions #3 and #4).

Regression #5: Renamed Endpoint. A renamed or removed endpoint in the specification is a severe regression, because existing clients that use it will break when updating to the next service version. In this case, the stateless diff of two network logs showed:

```
-GET /[...]/providers/Microsoft.Network/interfaceEndpoints
```

That is, the request to this endpoint was previously sent by the client generated from the old specification, but no longer after the specification update. When investigating the bug, we found that actual users had already detected such breaks and opened a GitHub issue. The response of the service owners was that this breaking change was intentional, and acceptable to them because the API was not yet officially announced. This nevertheless appears to be a regression, particularly because the previously working version did not end with *"-preview"*, a suffix typically used to differentiate such new unstable APIs that may change in the future.

Regressions #6 to #14: Unspecified Properties in Responses. An *additional*, but unspecified JSON property in the response of an API may break clients, because the property cannot be parsed by a generated SDK or is not handled by custom code written by the user based on the specification. Microsoft's REST API Guidelines [24] (Sec 12.3) mention this explicitly: "Azure defines the addition of a new JSON field in a response to be not backwards compatible". For instance, a property named `ipTags` started to be returned in responses by service version 2017-10-01 (regression #7). However, this property was added in the specification only a month later at version 2017-11-01, thus breaking clients in the meantime. Perhaps the service owners intended to expose these properties to clients, but did so "too early", namely in service versions where the specifications do not yet mention these new properties. Our approach found nine of such cases, because the stateful diff showed those properties as inserts in the received response bodies.

Table 3: Size comparison of stateful vs. stateless diffs for DNS and Network APIs depending on the update direction.

Diffing Method	Update Direction	Sum Lines	Sum Bytes	Empty Diffs (of 139 in total)
stateful	specification	8,784,417	677,605,439	14 (10%)
stateful	service	388,757	34,951,726	0
stateless	specification	2,197	505,298	56 (40%)
stateless	service	95	29,394	136 (98%)

Table 4: Diff sizes with the diagonal-only strategy.

Diffing Method	Update Direction	Sum Lines (Reduction)	Sum Bytes (Reduction)	Found Bugs
stateful	service	36,637 (9%)	3,690,016 (11%)	9 (of 9)
stateless	specification	302 (14%)	63,774 (13%)	5 (of 5)

4.5 Quantitative Evaluation

We now evaluate individual parts of our approach in more detail.

4.5.1 Comparing Stateful vs. Stateless Diffing. As expected, stateful diffs contain more information on average (2.7 MB per textual diff file) than stateless diffs (6 KB). Indeed, stateless diffs only consider request types (i.e., HTTP methods and endpoints) and the response status, but ignore request and response bodies.

Table 3 shows that the sizes of the diffs also depend on the direction of the comparison, i.e., whether the specification or the service version remains constant when diffing a pair of network logs. Stateful diffing across specification updates produces a lot of differences, in total 678 MB of text files. The root cause for these large diffs is that new requests or parameters in the specification will appear in the network log of the new client, and add up quickly if the request is tested many times. Since this is a lot of data to manually inspect, we do not make use of this output in our approach (and show it grayed-out in the table). In contrast, state*less* diffing across specification updates (third row), produces a much more manageable amount of diffs to inspect, namely 505 KB or about two thousand lines in total.

For service updates (i.e., where the compared network logs come from the same client derived from a constant specification version), stateless diffing abstracts away too much information and brings up almost no warnings (last row). 98% of the diffs in that case are empty. When diffing network logs from the same client, we thus use stateful diffing (second row), which is both feasible in terms of output produced and more precise, because the diff may contain changes in the services response bodies (e.g., added properties).

Table 5: Relation of regressions to diffs in which they manifest and if the diagonal-only diffing strategy finds them.

Regression:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
In # Diffs	7	4	4	4	1	1	5	6	13	3	3	9	12	12
On Diagonal?	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

4.5.2 Upper-Triangular Matrix vs. Diagonal-Only Strategy. For a comprehensive evaluation, we inspected all 278 diffs (DNS: 6, Network APIs: 272) of the upper-triangular matrix of specification and service versions, and found the 14 regressions from Table 2. In Section 2.6 we additionally propose a diagonal-only strategy that requires inspection of only two updates (and thus diffs) per new API version: First, diffing the network logs of (c_n, s_n) against (c_n, s_{n+1}) when the service is updated, and then diffing (c_n, s_{n+1}) against (c_{n+1}, s_{n+1}) when the new specification is published. This reduces the number of diffs greatly – linear instead of quadratic in terms of API versions. With the tested APIs, the diagonal-only subset of diffs comprises only 36 files (DNS: 4, Network APIs: 32) instead of 278. Table 4 shows that this results in a large reduction in terms of lines and bytes: stateful diffing across specification versions produces only 36,637 lines to inspect instead of 388,757, i.e., a reduction to 9% of the original effort. For stateless diffing across specification versions, there are only 302 lines instead of 2,197 to inspect, or a reduction to 14%.

Table 5 shows the mapping of regressions to diffs, i.e., in how many diffs each regression appears. For instance, regression #9 appears in 13 distinct diff files. We can see that despite this reduction in inspection effort, the diagonal-only strategy still manages to find *all 14 regressions* in total. This strategy thus provides a more *attractive diffs-to-regression ratio* than the upper-triangular-matrix strategy, especially for large numbers N of API versions.

4.5.3 Runtime and Manual Inspection Effort. All experiments were conducted on a single commodity PC with an Intel Core i7 processor and 32GB of main memory under Windows 10. For the DNS API, collecting the network logs for all valid combinations of specification and service versions took 11 minutes and 28 seconds. For the much larger Network APIs, this step took 3 days, 8 hours, and 4 minutes. During this interaction with the Azure services, a total of 92,381 requests were sent by *RESTler*. Running our diffing tools for all network logs took less than an hour for all valid pairs of logs in total. That is, the runtime of our approach is clearly dominated by exercising the web services, not the diffing of the collected logs.

Analyzing all 278 diffs of all the tested APIs for this historical analysis took about a week of work for a PhD student. In particular, analyzing stateless diffs was relatively quick since 40% were empty and the remaining ones were often duplicates of each other. Since stateful diffs contain more information (e.g., bodies of requests and responses), inspecting them took more time but was still manageable since differences, e.g., in the response of a particular request, re-appear several times and can be skipped. For this, we made use of code folding for JSON files (which stateful diffs are).

4.6 Threats to Validity

Even though we evaluate our approach on a large collection of real-world REST APIs, some threats to validity remain.

Our approach uses *RESTler* to automatically generate requests from an API specification, and test the service with those sequences. The amount of regressions that can be uncovered thus depends on how well *RESTler* can exercise a service under test. However, our approach makes no assumption on how requests are generated. The method for generating requests is a black box that can be exchanged or improved independently of our core differential testing idea.

The results of our evaluation also depend on the effectiveness of our diffing described in Section 3. Future work could improve our implementation by using more elaborate diffing algorithms and thus reduce the number of false positives. The core idea of our approach, namely the difference between specification and service regressions and that testing for both is essential, remains valid.

Finally, our experimental results also depend on the specific APIs considered, as well as the bugs in those APIs and services. While the specifications of the Azure networking APIs live in a single repository, this is merely an artifact of centralizing these specifications for developer convenience. These APIs are distinct, targeting different services, and written and maintained by different service owners.

4.7 Continuous Differential Regression Testing

We presented a *historical analysis* of Azure networking APIs in order to *evaluate* differential regression testing on a large amount of data, to see how many changes (diffs) there are, how many regressions we can find using our approach, what these regressions look like, and to determine which differential testing strategy (upper-triangular matrix or diagonal-only) works best.

However, the main practical use of our approach is *continuous differential regression testing*, whereby differential testing is applied to detect regressions in the *latest-only* API version update *before deployment*. Another similar main application is the pre-deployment detection of service regressions across (daily or weekly) service *revisions*, i.e., service updates which do *not* involve a new API version. In this scenario, the latest API specification is fixed and the testing results obtained with today's service are compared only with those obtained during the last testing results available.

In such a *continuous testing* context, there are *at most two diff* files to inspect, and their sizes is typically *much smaller* than when comparing very different versions as in our historical analysis. In particular, they are empty when adding a new endpoint, adding new examples, or updating description text. Also, within a continuous-integration environment, API changes are already reviewed since they may break clients (e.g., see [35]). Our automation provides a systematic way for API changes to be presented to developers. Inspecting a small diff file only requires minutes by API experts.

Yet regressions can still be found this way. As an example, we found a regression across a revision of the DNS service version 2017-10-01 between May 5, 2019 and May 19, 2019 with respect to the unspecified property `zoneType` (see regression #6 in Table 2). A manual analysis revealed that, on May 5, all versions of the service returned the extra property, even though it was only declared in the newer specification of 2018-05-01. Then, on May 19, it appears that a fix was attempted: the property was removed from the (dynamic) response of several of the earlier service versions, matching the corresponding specifications, except for service version 2017-10-01, where the specification and service were still out-of-sync.

5 RELATED WORK

REST APIs and Their Versioning. Since the REST principle was introduced in [21, 22], REST APIs have become a building block of the modern web, and a foundation of cloud services. REST API versioning is discussed in books like [7, 33, 40], where backwards compatibility is emphasized as a golden-rule of good REST API design and evolution. However, we are not aware of any other work discussing the two types of regressions, namely service vs. specification regressions, defined in Section 2 and how to detect these with differential testing. This paper introduces a *principled foundation* to this important topic.

Regression Testing for REST APIs. Regression testing [39], i.e., running a test suite after modifying software in order to ensure that existing features are still working correctly, is a widely-adopted form of testing [29], including in industry and practice [41]. With the recent explosion of web and cloud services, regression testing has naturally been applied to REST APIs as well, mostly in commercial tools, such as SoapUI [5], Postman [3], and others [2, 4]. These tools are helpful to develop a fixed regression test suite, possibly by first recording live or manually-generated test traffic. However, these tools (1) do not generate tests automatically and they (2) do not perform differential testing.

Test Generation Tools for REST APIs. Other tools available for testing REST APIs generate *new tests* by capturing API traffic, and then parsing, fuzzing and replaying the new traffic with the hope of finding bugs [9, 10, 14, 15, 43, 45, 47]. Other extensions are possible if the service code can be instrumented [11] or if a functional specification is available [44]. *RESTler* [12] 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.* These tools can find bugs like `500 Internal Server Errors`, but none of these perform differential testing or target regressions specifically.

Differential Testing. Originally devised by McKeeman [34], differential testing is now a well-known technique for solving the oracle problem [13] in automated testing by comparing two different, but related programs on the same inputs. One way of comparing different programs is by taking different implementations, e.g., for compiler testing [16, 18, 27, 31, 48]. Since then, differential testing has also been applied to other development tools, such as IDEs [17], interactive debuggers [28], and program analyzers [26]. In this work, we also use differential testing, but apply it to REST APIs in order to compare various client-service configurations.

Static Diffing to Find Specification Regressions. Our stateful and stateless approaches to diffing network logs are built upon standard algorithms for computing edit distances and edits scripts [38]. Diffing can be applied directly to Swagger specifications in order to find some types of clear regressions, such as removing a request from an API [1]. However, static diffing of specifications has its own challenges. First, these diffs can be large: for instance, the 16 diffs files obtained by pairwise diffing all 17 Azure Network API Swagger specifications in chronologic order result in a total of 15,571 lines and 465,624 bytes. Second, most differences are harmless: adding new files, reorganizing API requests, adding or modifying examples, etc. Third, even when modifying specific parameters in a new specification version, the old parameters are often still handled correctly by the new service for backwards-compatibility reasons, and therefore are *not* regressions; the only way to eliminate such false alarms is by *dynamically* testing the new service against (client code generated from) the old specification. In contrast, specification regressions can be found with stateless diffing and the diagonal-only strategy (16 diffs) by inspecting only 302 lines (see Table 4), i.e., two orders of magnitude less data compared to 15,571 lines with static diffing. Overall, static specification diffing is complementary to the dynamic testing-based approach developed in this work.

6 CONCLUSION

This paper introduces differential regression testing for REST APIs. It is the first to point out the key distinction between *service and specification regressions.* We discussed how to detect such regressions by comparing network logs capturing REST API traffic using stateful and stateless diffing. To demonstrate the effectiveness of this approach on a large set of services and APIs, we presented a detailed API history analysis of Microsoft Azure networking APIs, where we detected 14 regressions across 17 versions of these APIs. We discussed how these regressions were later fixed in subsequent specification versions and service deployments.

The main application of our approach is continuous differential regression testing, whereby differential testing is applied to detect regressions in the latest API version of a service *before* its deployment, hence avoiding expensive regressions affecting customers. We plan to deploy our tools widely soon, and we hope the evidence provided in this paper will encourage adoption.

ACKNOWLEDGMENTS

We thank Albert Greenberg, Anton Evseev, Mikhail Triakhov, and Natalia Varava from the Microsoft Azure Networking team for encouraging us to pursue this line of work and for their comments on the results of Section 4.

REFERENCES

[1] [n.d.]. *42crunch.* https://42crunch.com/
[2] [n.d.]. *Apigee Docs.* https://docs.apigee.com/
[3] [n.d.]. *Postman | API Development Environment.* https://www.getpostman.com/
[4] [n.d.]. *vREST – Automated REST API Testing Tool.* https://vrest.io/
[5] [n.d.]. *The World's Most Popular Testing Tool | SoapUI.* https://www.soapui.org/
[6] 2019. *Azure SDK.* https://github.com/Azure/azure-sdk
[7] S. Allamaraju. 2010. *RESTful Web Services Cookbook.* O'Reilly.
[8] Amazon. 2019. *Amazon Web Services (AWS) - Cloud Computing Services.* https://aws.amazon.com/
[9] APIFuzzer [n.d.]. APIFuzzer. https://github.com/KissPeter/APIFuzzer.
[10] AppSpider [n.d.]. AppSpider. https://www.rapid7.com/products/appspider.
[11] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology* 28, 1 (2019).
[12] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19).* IEEE Press, Piscataway, NJ, USA, 748–758. https://doi.org/10.1109/ICSE.2019.00083

[13] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

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

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

[16] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 197–208.

[17] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated Testing of Refactoring Engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. ACM, New York, NY, USA, 185–194. https://doi.org/10.1145/1287624.1287651

[18] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133917

[19] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. 2014. Web API growing pains: Stories from client developers and their code. In *Proceedings of the 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)* (Antwerp, Belgium).

[20] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE âĂŹ14)*. Association for Computing Machinery, New York, NY, USA, 313âĂŞ324. https://doi.org/10.1145/2642937.2642982

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

[22] Roy T. Fielding and Richard N. Taylor. 2002. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)* 2, 2 (2002), 115–150.

[23] Zack Grossbart. 2019. *JSON Diff – The semantic JSON compare tool*. http://www.jsondiff.com/

[24] Microsoft REST API Guidelines Working Group. 2019. *Microsoft REST API Guidelines*. https://github.com/microsoft/api-guidelines/blob/vNext/Guidelines.md

[25] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashm, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. 2013. Will You Still Compile Me Tomorrow?. In *Proceedings of the 2013 21st ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA.

[26] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 239–250.

[27] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

[28] Daniel Lehmann and Michael Pradel. 2018. Feedback-directed Differential Testing of Interactive Debuggers. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 610–620. https://doi.org/10.1145/3236024.3236037

[29] Hareton KN Leung and Lee White. 1989. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance-1989*. IEEE, 60–69.

[30] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. 2013. How Does Web Service API Evolution Affect Clients?. In *Proceedings of the 2013 IEEE 20th International Conference on Web Services* (Santa Clara, CA).

[31] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986

[32] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What Bugs Cause Production Cloud Incidents?. In *Proceedings of HotOS'2019* (Bertinoro, Italy).

[33] Mark Masse. 2011. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.".

[34] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[35] Microsoft. 2019. *Azure ARM API Review Checklist*. https://github.com/Azure/azure-rest-api-specs/pull/6632

[36] Microsoft. 2019. *Azure REST API Specifications*. https://github.com/Azure/azure-rest-api-specs

[37] Microsoft. 2019. *Microsoft Azure Cloud Computing Platform & Services*. https://azure.microsoft.com/en-us/

[38] Eugene W. Myers. 1986. An O(ND) difference algorithm and its variations. *Algorithmica* 1, 1 (01 Nov 1986), 251–266. https://doi.org/10.1007/BF01840446

[39] G. J. Myers. 1979. *The Art of Software Testing*. Wiley.

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

[41] Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. 1998. Regression Testing in an Industrial Environment. *Commun. ACM* 41, 5 (May 1998), 81–86. https://doi.org/10.1145/274946.274960

[42] Tom Preston-Werner. 2019. *Semantic Versioning 2.0.0*. https://semver.org/

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

[44] 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).

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

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

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

[48] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532