AGORA: Automated Generation of Test Oracles for REST APIs

Juan C. Alonso
javalenzuela@us.es
SCORE Lab, I3US Institute,
Universidad de Sevilla
Seville, Spain

Sergio Segura
sergiosegura@us.es
SCORE Lab, I3US Institute,
Universidad de Sevilla
Seville, Spain

Antonio Ruiz-Cortés
aruiz@us.es
SCORE Lab, I3US Institute,
Universidad de Sevilla
Seville, Spain

ABSTRACT

Test case generation tools for REST APIs have grown in number and complexity in recent years. However, their advanced capabilities for automated input generation contrast with the simplicity of their test oracles, which limit the types of failures they can detect to crashes, regressions, and violations of the API specification or design best practices. In this paper, we present AGORA, an approach for the automated generation of test oracles for REST APIs through the detection of *invariants*—properties of the output that should always hold. In practice, AGORA aims to learn the expected behavior of an API by analyzing previous API requests and their corresponding responses. For this, we extended the Daikon tool for dynamic detection of likely invariants, including the definition of new types of invariants and the implementation of an *instrumenter* called Beet. Beet converts any OpenAPI specification and a collection of API requests and responses to a format processable by Daikon. As a result, AGORA currently supports the detection of up to 105 different types of invariants in REST APIs. AGORA achieved a total precision of 81.2% when tested on a dataset of 11 operations from 7 industrial APIs. More importantly, the test oracles generated by AGORA detected 6 out of every 10 errors systematically seeded in the outputs of the APIs under test. Additionally, AGORA revealed 11 bugs in APIs with millions of users: Amadeus, GitHub, Marvel, OMDb and YouTube. Our reports have guided developers in improving their APIs, including bug fixes and documentation updates in GitHub. Since it operates in black-box mode, AGORA can be seamlessly integrated into existing API testing tools.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Information systems → RESTful web services**.

KEYWORDS

REST APIs, test oracle, invariant detection, automated testing

ACM Reference Format:
Juan C. Alonso, Sergio Segura, and Antonio Ruiz-Cortés. 2023. AGORA: Automated Generation of Test Oracles for REST APIs. In *Proceedings of Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ISSTA '23, July 17–21, 2023, Seattle, WA, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Web Application Programming Interfaces (APIs) allow heterogeneous software systems to interact over the network [48, 68]. Modern web APIs typically adhere to the REpresentational State Transfer (REST) architectural style, being referred to as *REST APIs* [32]. REST APIs are decomposed into multiple resources (e.g., a *payment* in the VISA API [9]) that clients can manipulate through HTTP interactions. REST APIs have become the de facto standard for software integration, being a key part of the business model of companies such as Amazon, Google, Netflix, or Twitter [48]. The importance and pervasiveness of REST APIs is reflected on the number of APIs hosted by popular API repositories such as RapidAPI (40K) [3].

The critical role of REST APIs in software integration has driven the creation of numerous techniques and tools for the automated detection of faults within these APIs [41, 51]. Most techniques adopt a black-box approach, where test cases are automatically derived from the specification of the API under test, typically in the OpenAPI Specification (OAS) format [2]. These test cases are created by setting values to the input parameters and checking the validity of the returned responses by applying different test oracles, i.e., mechanisms to determine whether a test execution reveals a fault [19]. Despite the capabilities of these tools for detecting faults in industrial APIs [15, 16, 39, 59], they are all limited by the types of failures that they can detect, namely crashes (responses with a 5XX HTTP status code) [14, 15, 44, 50, 57, 75, 78], disconformities with the API specification (e.g., missing output JSON property) [14, 44, 50, 57, 75], regressions [35, 39], and violations of API best practices (e.g., checking that the results of multiple calls to idempotent operations are identical) [16, 18, 74, 82]. As an example, Listing 2 shows a response for the "getAlbumTracks" operation of the Spotify API. The response conforms to the API specification and therefore would be considered as a correct output by existing tools. However, the response could still contain errors that would go unnoticed by current tool support, including incorrect field length or format, and violations of numerical constraints or array properties, among others. Recent surveys [41] and tool comparisons [51, 59] have identified the generation of test oracles as one of the major challenges in the generation of test cases for REST APIs. This is the problem that motivates our work.

The automated generation of test oracles is an active research topic. Existing approaches mostly differ on the inputs from which test oracles are generated, including source code [28, 77], program specifications [34, 47], documentation [20, 40], and previous executions [61, 63], among others. A common approach for test oracle

generation is the detection of likely *invariants*, properties of the program that should always hold, e.g., "*input.var* $\neq$ *null* $\Rightarrow$ *output.array is ordered*". Invariants are often discovered by analyzing previous program inputs and outputs, making this method suitable for programs written in different languages or for cases where the source code is not available, such as REST APIs. This is the strategy leveraged in our work.

In this paper, we present AGORA, a black-box approach for the Automated Generation of Oracles for REST APIs. AGORA relies on the detection of likely invariants. For this purpose, we extended and modified the Daikon [31] system for dynamic invariant detection in two directions. Firstly, we present a novel software tool—a Daikon *instrumenter* called Beet—that converts any OAS specification and a set of API requests and responses into a format processable by Daikon. This makes our approach seamlessly integrable into existing API testing tools supporting OAS. Secondly, we further enhanced the capabilities of Daikon by customizing and expanding its default set of invariants. Currently, AGORA supports the detection of 105 distinct types of invariants in REST APIs.

Evaluation results using 11 operations from 7 industrial APIs showed that a diverse set of just 50 API requests (and their corresponding responses) is sufficient for AGORA to learn hundreds of accurate invariants (test oracles), achieving a precision of 73.2%. This precision improves to 81.2% when learning from 10K API requests. These results surpass those obtained using the default set of invariants in Daikon, with a precision under 52%. We also evaluated the effectiveness of the generated test oracles in detecting failures by automatically seeding 1.1M errors in the outputs of the API operations under test. The test oracles generated by AGORA, learned from only 50 API requests, were able to detect 57.3% of the incorrect outputs, supporting the cost-effectiveness of our approach.

During our evaluation, AGORA generated several invariants that indicated issues within the target APIs. One example was the invariant `return.room.typeEstimated.beds >= 0`, which revealed a bug in the Amadeus API where certain hotel offers included rooms with *zero* beds. This issue was reported and confirmed by Amadeus developers. Overall, AGORA resulted in the detection of 11 faults (4 confirmed, 2 fixed) in 7 operations of 5 industrial APIs, all of which would have passed unnoticed by current test case generators. Our findings also led to updates in the documentation of GitHub. This highlights the value of AGORA not only as a test oracle generation approach, but also as a testing technique on its own.

In summary, after introducing the background and related work on testing REST APIs, test oracle generation, and Daikon (Section 2), this paper presents the following original research and engineering contributions:

- AGORA, a black-box approach for the automated generation of test oracles for REST APIs based on the analysis of the API specification and previous requests and responses (Section 3).
- Beet, a novel Daikon instrumenter for REST APIs readily integrable into existing test case generation tools for REST supporting OAS. Beet is open-source and available on GitHub [5].
- A customized version of Daikon supporting the detection of 105 distinct types of invariants in REST APIs.
- An empirical evaluation of AGORA in terms of precision and failure detection in 11 operations from 7 industrial APIs (Section 4), including reports of 11 real-world bugs (Section 5).
- A publicly available replication package including the source code and the data used in our work, as well as a pre-configured virtual machine to ease reproducibility and replicability [7].

We address the threats to validity in Section 6, and conclude the paper in Section 7.

A very preliminary version of this work (two-page paper) obtained the first prize (graduate category) in the ACM Student Research Competition in ESEC/FSE 2022 [11] and the second prize in the ACM SRC Grand Finals [8].

2 BACKGROUND AND RELATED WORK

This section introduces the key concepts related to automated testing of REST APIs, test oracle generation, and Daikon.

2.1 Automated testing of REST APIs

Modern web APIs are typically compliant with the REpresentational State Transfer (REST) [32] architectural style, being known as REST APIs [68]. REST APIs are usually composed of multiple RESTful web services, with each one of them implementing one or more create, read, update, and delete (CRUD) operations on a resource (e.g., a repository in the GitHub API [36]). These operations are typically invoked by sending HTTP requests to specific Uniform Resource Identifiers (URIs) representing a resource or a collection of resources.

REST APIs are commonly described using the OpenAPI Specification (OAS) [2] format, arguably the industry standard. An OAS document describes the API in terms of the operations supported, as well as their input parameters and responses. As an example, Listing 1 depicts an excerpt of the OAS specification of the "getAlbumTracks" operation of the Spotify API [4]. The document describes the HTTP method and the URI required to call the API operation (lines 1-3), operation ID (line 4), input parameters (lines 5-20), and possible responses (lines 21-61). Listing 2 depicts a response for the "getAlbumTracks" operation conforming to the specification.

The majority of approaches for automated testing of REST APIs adopt a black-box approach [12, 15, 16, 26, 38, 39, 44, 50, 53, 54, 58, 71, 75, 78]. Given an OAS document, these techniques automatically generate pseudo-random test cases (sequences of HTTP requests) and test oracles (assertions on the HTTP responses). Approaches mainly differ in the way they generate API calls (i.e., test inputs) using techniques such as model-based testing [53, 58, 76], property-based testing [44, 50, 69, 71], and constraint-based testing [56, 57], among others. Some methods focus on testing individual API operations and generate single API requests, while others also generate sequences of API calls for stateful testing [15, 26, 44, 75]. White-box approaches require access to the API source code and are far less common than black-box approaches. Most existing techniques leverage search algorithms to maximize fault detection and code coverage [14, 72, 83].

In terms of fault detection, generated test oracles are primarily limited to detecting API crashes (e.g., 500 status codes) and violations of the API specification [14, 44, 50, 57, 75]. Other test oracles focus on detecting regressions [35, 39] or adherence to best design practices [16, 18, 26, 74, 82]. However, all these approaches have limitations in detecting issues that go beyond mere syntax. For example, existing approaches would ignore domain-specific

```yaml
paths:
  '/albums/{id}/tracks':
    get:
      operationId: 'getAlbumTracks'
      parameters:
        - name: id
          description: 'The Spotify ID for the album'
          in: path
          required: true
          type: string
        - name: market
          description: 'An ISO 3166-1 alpha-2 country code'
          in: query
          required: false
          type: string
        - name: limit
          description: 'The maximum number of items to return'
          in: query
          required: false
          type: integer
      responses:
        '200':
          description: 'OK'
          schema:
            type: object
            properties:
              total:
                type: integer
              href:
                type: string
              items: # Array of objects
                type: array
                items:
                  type: object
                  properties:
                    artists: # Array of objects
                      type: array
                      items:
                        type: object
                        properties:
                          id:
                            type: string
                          name:
                            type: string
                    available_markets: # Array of strings
                      type: array
                      items:
                        type: string
                    id:
                      type: string
                    name:
                      type: string
                    explicit:
                      type: boolean
                    linked_from: # Nested object
                      type: object
                      properties:
                        id:
                          type: string
                        uri:
                          type: string
```

Listing 1: OAS excerpt of the Spotify API.

```json
{
   "total": 14,
   "href": "https://api.spotify.com/albums/4Em5W5HgYEvhpc/tracks
      ?limit=1&market=ES",
   "items": [
      {
         "artists": [
            {
               "id": "2CvCyf1gEVhI0mX6aFXmVI",
               "name": "Paul Simon"
            },
            {
               "id": "70cRZdQywnSFp9pnc2WTCE",
               "name": "Arthur Garfunkel"
            }
         ],
         "available_markets": [ "ES", "US", "JP" ],
         "id": "0gFvkiT2afIcJwNxXQ7W51",
         "name": "Mrs. Robinson",
         "explicit": false,
         "linked_from": {
            "id": "98cZPdKywnMGp8fnw2XTYU",
            "uri": "https://spotify.com/artist/98cZPdKywnMGp8fnw2XTYU"
         }
      }
   ]
}
```

Listing 2: Spotify API response in JSON format.

assertions in Listing 2, such as checking that the `linked_from.uri` response field should be a valid URL that contains the value of the `linked_from.id` field, or that the size of the `items` response field should be lower or equal than the value of the `total` response field, among others. Generating such test oracles is the goal of AGORA.

2.2 Test oracle generation

Automated test case generation techniques can be classified based on their inputs, and their application domains. Regarding their inputs, test oracles have been derived from source code [28, 60, 77, 80], formal specifications [34, 47], semi-structured documentation [20, 21, 40, 81], previous program executions [23, 24, 46, 49, 61–63, 73], or a combination of them. Application domains include Java projects [20, 28, 61], machine learning programs [22] and cyber-physical systems [17], among others.

Other related techniques include metamorphic testing and regression testing. Metamorphic testing [70, 71] relies on the manual identification of metamorphic relations among the inputs and outputs of two or more executions of the program under test. Regression testing [33, 79] relies on previous versions of the software under test to confirm that a change has not adversely affected existing features.

A common approach for the generation of test oracles is through the detection of likely invariants. An *invariant* is a property that is always satisfied at one or more points of the execution of a program [30]. For example, given a Java function that receives an array and returns the same array with an additional element, an invariant could specify that the returned array always has a greater size than the array provided as input, i.e., `size(return.array[])>size(input.array[])`. Invariants can serve as test oracles to determine the correctness of a program output. Invariants can be detected either statically (analyzing the source code, without executing it) [27, 37] or dynamically (analyzing the behavior of a program through multiple executions) [30, 31, 43, 52]. Statically detected invariants are usually less numerous and less specific than those detected by dynamic techniques [29, 64, 65]. However, dynamic invariant detection techniques may result in a greater number of false positives, especially if the executions of the program under analysis lack variety. Since it is infeasible to run the program with all possible inputs, dynamically detected invariants are referred to as *likely* invariants, until they are confirmed by a domain expert.

The automated detection of likely invariants has shown promising results in contexts such as Java programs [61], relational databases [25], automated program repair [84], WS-BPEL composition testing [66], cyber-physical systems [10] or Cloud-based [67] and distributed [42] systems. To the best of our knowledge, this is the first approach for the automated detection of likely invariants in REST APIs.

2.3 Daikon

Daikon [31] is an open-source tool that detects likely invariants in programs by monitoring test executions. This monitoring process involves observing the program state at designated *program points*, initially considering all possible invariants as valid. Those invariants that are not violated by any execution are reported as likely invariants. Daikon operates by analyzing an instrumented version of a software execution, generated by an *instrumenter* or front-end. This instrumentation produces a declaration file and a data trace file. The *declaration* file describes the structure of program points in terms of input and output variables. The *data trace* file contains the values assigned to the variables in each execution. Instrumenters

```
public Result computeSquare(int inputValue) {
      return new Result(inputValue*inputValue);
}
```

Listing 3: Java computeSquare function.

```
ppt main.computeSquare(int):::EXIT1
ppt-type subexit
variable inputValue
  var-kind variable
  dec-type int
  rep-type int
variable return
  var-kind return
  dec-type Result
  rep-type hashcode
variable return.square
  var-kind field square
  enclosing-var return
  dec-type int
  rep-type int
```

Listing 4: Daikon declaration file.

```
main.computeSquare(int):::EXIT16
inputValue
10
1
return
1458849419
1
return.square
100
1
```

Listing 5: Daikon dtrace file.

```
=== main.computeSquare(int):::EXIT
return.square >= 0
inputValue <= return.square
```

Listing 6: Likely invariants of computeSquare.

are available for various programming languages and data formats, including Java, Perl, C++, and CSV [1].

Listing 3 shows a sample Java method for computing the square of an input integer. For this method, a Daikon instrumenter should generate an ENTER and an EXIT program point to analyze the evolution in the program state. Listing 4 depicts the content of a Daikon declaration file for the EXIT program point. As illustrated, the definition of the program point is followed by the declaration of the variables representing the input and output parameters to be observed. Each variable definition includes information about its name, datatype in the original program (`dec-type`) and in the data trace file (`rep-type`), and whether the variable is a property of another variable (`enclosing-var`), among others. In the example, Listing 4 contains the int-type variable `inputValue` (input parameter), and the object-type variable `return`, containing the integer property `return.square` (method output).

Listing 5 shows the data trace file of one execution of the EXIT program point. For each variable, the data trace contains its name, the value observed in the program execution, and the modified bit. This bit specifies whether a variable value has been assigned or not. After processing these files, assuming a larger data trace file, Daikon would return a set of invariants as the one presented in Listing 6.

3 AGORA

Figure 1 shows an overview of AGORA, our approach for the automated generation of test oracles for REST APIs. At the core of the approach is Beet[1], a novel Daikon instrumenter. Beet receives three inputs: 1) the OAS specification of the API under test, 2) a set of API requests, and 3) the corresponding API responses. As a result, Beet returns an instrumentation of the API requests consisting on a declaration file—describing the format of the API operations inputs and outputs—and a data trace file—specifying the values assigned to each input parameter and response field in each API call. This instrumentation is then processed by our customized version of Daikon, resulting in a set of likely invariants that, once confirmed by developers, can be used as test oracles.

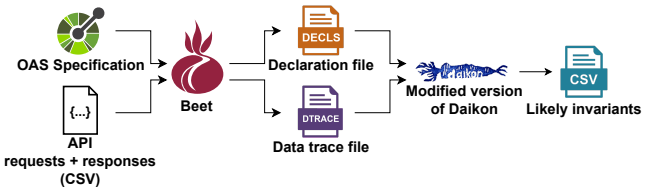

Figure 1: Workflow of AGORA.

AGORA works at the operation level, that is, it learns invariants from API requests testing individual API operations, as this is the most basic and common testing practice [12, 50, 51, 56–59, 71]. Learning invariants for sequences of API calls (e.g., creating a resource, then updating it, then deleting it) could be implemented in a similar way and remain for future work. Also, AGORA currently supports JSON as the de facto standard data format. Supporting other languages should be straightforward using existing converters, e.g., XML to JSON.

In the next subsections, we describe the Beet instrumenter and the types of invariants currently supported by AGORA.

3.1 Beet: A Daikon instrumenter for REST APIs

This subsection outlines the process followed by Beet to generate a declaration and a data trace file from an OAS specification and a set of API requests and responses.

3.1.1 Declarations. The declaration files provide a description of the inputs and outputs for each API operation. Table 1 summarizes how these files are generated from the information in the API specification. For each operation, an ENTER program point is created, followed by the definition of input parameters, if any. These input parameters are defined as a single `input` variable representing the whole input with as many properties as input parameters (`input.<paramName>`). Similarly, an EXIT program point is created for each operation, including an identical definition of the input parameters, a `return` variable representing the whole output, and as many properties of the return variable as response fields (`return.<paramName>`). The value of `<primitiveType>` in Table 1 can be either java.lang.String, int, double or Boolean. Variables of type object are represented using hashcodes.

Two cases require special consideration: JSON objects and arrays of objects. JSON objects are flattened and each property is treated as a separate parameter. On the other hand, in Daikon, the elements of an array of objects can only be specified using their hashcode, limiting the types of invariants that can be identified to changes in the array. To support more informative array-related output invariants, Beet implements a recursive strategy by creating a new

[1]Existing Daikon instrumenters have adopted the name of vegetables [1]. We decided to follow this convention.

Table 1: Mapping from OAS specification to Daikon declaration file.

API Request	API operation	`ppt <operationName>&<statusCode>():::ENTER`
		`ppt-type enter`
		`variable input`
		`  var-kind variable`
		`  dec-type <operationName>&Input`
		`  rep-type hashcode`
			API Response	API operation	`ppt <operationName>&<statusCode>():::EXIT<exitNumber>`
					`ppt-type subexit`
					`variable input`
					`-- Input variables --`
					`variable return`
					`  var-kind return`
					`  dec-type <ppt-name>&Output&<statusCode>`
					`  rep-type hashcode`

	Input param	`variable input.<parentVariable>.<paramName>`
		`  var-kind field <paramName>`
		`  enclosing-var input.<parentVariable>`
		`  dec-type <primitiveType> | <ppt-name>&Input&<paramName>`
		`  rep-type <primitiveType> | hashcode`
				Response field	`variable return.<parentVariable>.<fieldName>`
					`  var-kind field <fieldName>`
					`  enclosing-var return.<parentVariable>`
					`  dec-type <primitiveType> | <ppt-name>&Output&<fieldName>`
					`  rep-type <primitiveType> | hashcode`

	Input array	`variable input.<parentVariable>.<paramName>`
		`  var-kind field <paramName>`
		`  enclosing-var input.<parentVariable>`
		`  dec-type <primitiveType>[] | <paramName>[]`
		`  rep-type hashcode`
		`variable input.<parentVariable>.<paramName>[..]`
		`  var-kind array`
		`  enclosing-var input.<parentVariable>.<paramName>`
		`  array 1`
		`  dec-type <primitiveType>[] | <paramName>[]`
		`  rep-type <primitiveType>[] | hashcode[]`
				Response array	`variable return.<parentVariable>.<fieldName>`
					`  var-kind field <fieldName>`
					`  enclosing-var return.<parentVariable>`
					`  dec-type <primitiveType>[] | <fieldName>[]`
					`  rep-type hashcode`
					`variable return.<parentVariable>.<fieldName>[..]`
					`  var-kind array`
					`  enclosing-var return.<parentVariable>.<fieldName>`
					`  array 1`
					`  dec-type <primitiveType>[] | <fieldName>[]`
					`  rep-type <primitiveType>[] | hashcode[]`

```
1  ppt getAlbumTracks&200():::ENTER
2  ppt-type enter
3  variable input
4    var-kind variable
5    dec-type getAlbumTracks&Input
6    rep-type hashcode
7  variable input.id
8    var-kind field id
9    enclosing-var input
10   dec-type java.lang.String
11   rep-type java.lang.String
12 variable input.market
13   var-kind field market
14   enclosing-var input
15   dec-type java.lang.String
16   rep-type java.lang.String
17 variable input.limit
18   var-kind field limit
19   enclosing-var input
20   dec-type int
21   rep-type int
```

Listing 7: ENTER program point of an API operation.

```
1  ppt getAlbumTracks&200():::EXIT1
2  ppt-type subexit
3  variable input
4  ...
5  variable return
6    var-kind return
7    dec-type getAlbumTracks&Output&200
8    rep-type hashcode
9  variable return.total
10   var-kind field total
11   enclosing-var return
12   dec-type int
13   rep-type int
14 variable return.href
15   var-kind field href
16   enclosing-var return
17   dec-type java.lang.String
18   rep-type java.lang.String
19 variable return.items
20   var-kind field items
21   enclosing-var return
22   dec-type items[]
23   rep-type hashcode
24 variable return.items[..]
25   var-kind array
26   enclosing-var return.items
27   array 1
28   dec-type items[]
29   rep-type hashcode[]
```

Listing 8: EXIT program point of an API operation.

EXIT[2] program point (that we define as a new nesting level) for each distinct array element, describing its properties as independent response fields.

As an example, Listings 7 and 8 present the declarations of the ENTER and EXIT points for the "getAlbumTracks" operation of the Spotify API (Listing 1). In Listing 7, the definition of the ENTER program point is followed by the definition of the input parameters.

[2]ENTER and EXIT program points must be defined in pairs in Daikon. Each EXIT program point is paired with a renamed copy of the ENTER program point.

```
1  ppt getAlbumTracks&200&items():::EXIT2
2  ppt-type subexit
3  variable input
4  ...
5  variable return
6  ...
7  variable return.artists
8  ...
9  variable return.artists[..]
10 ...
11 variable return.available_markets
12   var-kind field available_markets
13   enclosing-var return
14   dec-type java.lang.String[]
15   rep-type hashcode
16 variable return.available_markets[..]
17   var-kind array
18   enclosing-var return.available_markets
19   array 1
20   dec-type java.lang.String[]
21   rep-type java.lang.String[]
22 variable return.id
23 ...
24 variable return.name
25 ...
26 variable return.track_number
27 ...
28 variable return.explicit
29 ...
30 variable return.linked_from
31   var-kind field linked_from
32   enclosing-var return
33   dec-type getAlbumTracks&Output&200&items&linked_from
34   rep-type hashcode
35 variable return.linked_from.id
36   var-kind field id
37   enclosing-var return.linked_from
38   dec-type java.lang.String
39   rep-type java.lang.String
40 variable return.linked_from.uri
41 ...
```

Listing 9: Second EXIT nesting level.

Specifically, an input variable representing the entire input, which has three properties, each representing a distinct input parameter (`input.id`, `input.market` and `input.limit`). Similarly, in Listing 8, the definition of the EXIT program point is followed by the definition of the input parameters (omitted for brevity), a return variable representing the entire output, and as many properties of the return variable as response fields (e.g., `return.total` and `return.href`). The response includes an array of objects, `items`, including the set of music albums matching the search criteria. This is transformed into two distinct variables, one of type object (hashcode) that represent the whole array (lines 19-23), and another of type array containing the hashcodes of the array elements (lines 24-29). Besides this, an additional EXIT program point is created—a new nesting level—defining the properties of each array item (i.e., Spotify album), as shown in Listing 9.

```
getAlbumTracks&200():::ENTER
input
1242334637
1
input.id
"4Em5W5HgYEvhpc"
1
input.market
"ES"
1
input.limit
1
1
```

Listing 10: ENTER data trace file.

```
getAlbumTracks&200():::EXIT1
input
...
return
2043815652
1
return.total
14
1
return.href
"https://api.spotify.com/albums/4Em5W5HgYEvhpc/tracks?limit=1&market=ES"
1
return.items
1534143414
1
return.items[..]
[313805079]
1
```

Listing 11: EXIT data trace file.

3.1.2 Data traces. Data trace files specify the actual input and output values observed during the execution of the API. The list of variables in the data trace record must be identical to that in the corresponding declaration. Listing 10 shows the data trace file corresponding to a request to the "getAlbumTracks" operation of the Spotify API with input parameters `id="4Em5W5HgYEvhpc"`, `market="ES"` and `limit=1`. Analogously, Listing 11 depicts the main data trace file of the corresponding response, containing, among other properties, an array of objects. For each array item, Beet generates a new pair of trace files (i.e., an ENTER and an EXIT) with the values of each object (i.e., Spotify Album), not included for brevity.

Beet has been implemented in Java and is open-source. We refer the reader to GitHub for a more exhaustive description of the instrumentation process and additional examples [5].

3.2 Invariant definition

This section details the changes performed on Daikon for detection of likely invariants in REST APIs. In order to identify classes of invariants that could be used as effective test oracles, we resorted to a benchmark of 40 APIs (702 operations) systematically collected from the RapidAPI repository [3], recently used by previous authors in the context of REST API testing [12]. Specifically, we studied the input and output format of each operation trying to identify common types of invariants in REST APIs.

We implemented 22 new types of invariants, suppressed 36 default Daikon invariants, and activated 9 invariants disabled by default in Daikon. The new API-specific invariants aim to detect specific common string patterns such as URLs, dates, or length constraints, among others. Suppressed invariants would most likely provide irrelevant or misleading information in our context and thus they were disabled, such as comparing the scalar value of strings or linear relations between numerical variables. Finally, we activated 9 invariants related to detecting subsets and supersets

```
=== getAlbumTracks&200():::ENTER
LENGTH(input.id)==14
input.limit >= 1
LENGTH(input.market)==2
=== getAlbumTracks&200():::EXIT
return.href is Url
input.limit >= size(return.items[])
return.total >= size(return.items[])
return.total >= 1
input.market is a substring of return.href
input.id is a substring of return.href
=== getAlbumTracks&200&items():::ENTER
...
=== getAlbumTracks&200&items():::EXIT
size(return.artists[]) >= 1
All the elements of return.available_markets[] have LENGTH=2
input.market in return.available_markets[]
LENGTH(return.id)==22
LENGTH(return.linked_from.id)==22
return.linked_from.uri is Url
LENGTH(return.linked_from.uri)==54
return.linked_from.id is a substring of return.linked_from.uri
=== getAlbumTracks&200&items&artists():::ENTER
...
=== getAlbumTracks&200&items&artists():::EXIT
LENGTH(return.id)==22
```

Listing 12: Detected invariants.

when comparing array variables (e.g., `x[] is a subsequence of y[]`) and detecting substrings relations between string variables (e.g., `input.id is a substring of return.href`). Overall, our customized version of Daikon supports a total of 105 distinct types of invariants for REST APIs, classified into five categories:

- *Arithmetic comparisons (48 invariants).* Specify numerical bounds (e.g., `size(return.artists[]) >= 1`) and relations between numerical fields (e.g., `input.limit >= size(return.items[])`).
- *Array properties (23 invariants).* Represent comparisons between arrays, such as subsets, supersets, or fields that are always member of an array (e.g., `return.hotel.hotelId in input.hotelIds[]`).
- *Specific formats (22 invariants).* Specify restrictions regarding the expected format (e.g., `return.href is Url`) or length (e.g., `LENGTH(return.id)==22`) of string fields.
- *Specific values (9 invariants).* Restrict the possible values of fields (e.g., `return.visibility one of {"public", "private"}`).
- *String comparisons (3 invariants).* Specify relations between string fields, such as equality (e.g., `input.name == return.name`) or substrings (e.g., `input.id is a substring of return.href`).

We refer the reader to the AGORA GitHub repository [5] for a more detailed description of each type of invariant, including examples. The set of invariants is not exhaustive and more types of invariants could be considered in the future.

Listing 12 shows some of the likely invariants inferred by Daikon for the "getAlbumTracks" operation of the Spotify API used as running example.

4 EVALUATION

We aim to answer the following research questions:

RQ1: *How effective is AGORA in generating test oracles?* We aim to measure the precision of AGORA in generating invariants that result in valid test oracles (i.e., they properly model the expected API behavior) using the default configuration of Daikon as a baseline.

RQ2: *What is the impact of the size of the input dataset on the precision of AGORA?* The precision of the detected invariants usually depends on the quality and diversity of the input datasets (i.e., API requests and responses). Hence, we aim to study the impact of dataset size on the effectiveness of AGORA.

RQ3: *How effective are the generated test oracles in detecting failures?* The final goal is generating test oracles that can be used during testing to identify erroneous responses caused by faults. Thus, we aim to investigate the effectiveness of the generated oracles for detecting non-trivial failures in REST APIs.

4.1 Experimental data

For our experiments, we resorted to a set of 11 operations from 7 industrial APIs (Table 2) tested by previous authors [12, 58, 59, 71]. The OAS specification of the APIs were obtained from their official websites. For those APIs that do not provide access to the official specification, we either generated them manually (OMDb and Yelp) or used the version available in APIs.guru [13] (Spotify and YouTube), modifying them according to the latest version of the web docs. Some of the specifications were either incorrect (parameters of type array defined as strings) or incomplete (missing response fields). We manually fixed those specifications to ensure that Beet could process them.

For each operation, we automatically generated and executed API calls using the RESTest [57] framework until obtaining 10K valid API calls per operation (110K calls in total). According to REST best practices [68], we consider an API response as valid if it is labeled with a 2XX HTTP status code. The OMDb API does not adhere to REST best practices, returning a 200 response containing the `error` response field when the user provides invalid data. For this API, we consider as valid those API responses that do not contain this field.

Some of the search operations support filtering parameters that can be highly restrictive, resulting in a significant number of responses containing zero results, which provide almost no information about the API behavior. To avoid this, when generating the API requests, we ensured that at least 90% of the responses contain results.

The invariants detected by AGORA largely depend on the diversity of the API requests provided as input. To foster such diversity, we followed current practices and manually created a data dictionary for each non-trivial input parameter based on the analysis of the API specification and the API documentation. For instance, for the `location` parameter of the Yelp API, we created a list of cities located in different continents. Numerical values and dates were configured using RESTest generators, that provide a random value in a specified range. Our replication package [7] contains the RESTest configuration files and the data dictionaries used for generating the test cases.

4.2 Experiment 1: Test oracle generation

In this experiment, we aim to answer RQ1 and RQ2 by evaluating the effectiveness of AGORA in generating test oracles for the target API operations.

4.2.1 Experimental setup. For each API operation, we randomly divided the set of automatically-generated requests (10K) into subsets of 50, 100, 500, 1K and 10K requests. Then, we ran AGORA—Beet instrumentation plus customized Daikon execution—using each subset as input and computed precision by manually classifying the inferred invariants as true or false positives. True positive invariants describe properties of the output that should always hold and therefore are valid test oracles. A false positive reflects a pattern that has been observed in all the API requests and responses provided as input but does not represent the expected behavior of the API. For example, one of the likely invariants inferred in Spotify states that the duration in milliseconds of a song should always be greater than the number of artists in the song: (`return.duration_ms>size(return.artists[])`). While this may be true in most cases, it is clearly not the intended behavior of the API, and therefore it is considered a false positive.

Daikon also detects invariants among input parameters (i.e., ENTER program points). While these invariants can offer insights into the API behavior, they are not used to calculate precision as they do not provide information about the output. Labeling them as true positives would result in inflated results.

We could not find any comparable approach to be used as a baseline. Therefore, we compared the effectiveness of AGORA against the default version of Daikon in identifying likely invariants for REST APIs. In both cases, our novel instrumenter, Beet, was used to transform API specifications, requests, and responses into inputs for Daikon. Our preliminary experiments revealed that 13 of Daikon default invariants resulted in a combinatorial explosion of string comparisons and a high number of false positives. To make the comparison feasible, we disabled these problematic invariants, detailed in our supplementary material [7].

The experiment was performed on a laptop equipped with Intel i7-11800H @2.30GHz, 32GB RAM, and 1TB SSD running Windows 11 and Java 8.

4.2.2 Experimental results.

RQ1: Effectiveness of the approach. Table 2 shows the results for each API operation, set of API requests (50, 100, 500, 1K, 10K) and approach (AGORA vs default Daikon). The columns labeled with "I" present the number of likely invariants detected, whereas the columns labeled with "P" present the total precision, that is, the percentage of true positives, i.e., valid test oracles. Next, we analyze the results obtained from the entire dataset of 10K requests (RQ1), shown in the last two columns of the table. The results with different sizes of the input dataset (RQ2) are analyzed in the next section.

When learning from the whole dataset (10K API requests), AGORA obtained a total precision of 81.2% (770 out of 948 invariants are valid oracles), whereas the original configuration of Daikon achieved 51.4% (363 out of 706). AGORA outperformed the default configuration of Daikon in all the API operations. Precision ranged between 50% in the Yelp API and 100% in the "createPlaylist" operation of the Spotify API. The number of invariants reported per operation oscillated between 7 in the "bySearch" operation of the OMDb API and 198 in the "createOrganizationRepository" of the GitHub API. We observed a correlation between the number of response fields and the number of reported invariants. This was confirmed by a correlation study with a Spearman coefficient of

0.9, indicating that the number of invariants tends to increase with the response size.

The sunburst in Figure 2 shows a breakdown of the reported invariant categories for each classification: true positives or false positives. The largest portion of false positives (75.8%) were found in the arithmetic comparison category, followed by specific values (18.5%), specific formats (2.8%) and string comparisons (2.8%), with no false positives of the array properties category. It is noteworthy that the precision of AGORA increases to 93.6% when suppressing the invariants of the arithmetic comparison category.

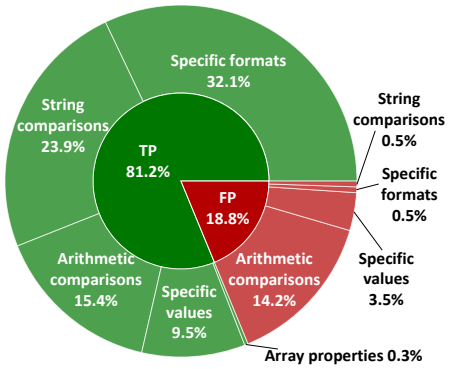

Figure 2: Breakdown by invariant categories.

False positives in the arithmetic comparison category typically occur when comparing numerical fields with values in different orders of magnitude, such as comparing the duration of a Spotify song in milliseconds with its number of artists. In many cases, it may be difficult to find a counterexample that refutes these invariants. The remaining false positives are either invariants that report an object as always null or invariants that limit a response field to only a specific value or set of values, but the API supports more. These false positives mainly occur when the API has not returned all possible values for an enum field (e.g., if the GitHub repository visibility is always "public"). These invariants indicate either a lack of diversity in the test suite or bugs in the API (c.f., Section 5).

Beet took between 0.3 seconds (50 requests) and 49.9 seconds (10K requests) to generate the instrumentation of the target API operations. Daikon (customized and default version) took between 0.2 seconds (50 requests) and 15.6 seconds (10K request) to detect the reported invariants. Overall, AGORA took around 1 minute to generate the invariants using the complete dataset of 10K requests.

Response to RQ1

AGORA is effective in generating test oracles obtaining a total precision of 81.2% when learning from 10K API requests. This means an improvement of 29.8% over the default configuration of Daikon. The precision of AGORA raises to 93.6% when suppressing the invariants in the arithmetic comparison category, which are the cause of 3 out of every 4 false positives.

RQ2: Impact of the size of the input dataset. Figure 3 shows the evolution of the precision of AGORA with respect to the number of input API requests, as detailed in Table 2. The total precision improved from 73.2% with 50 API requests (724 valid invariants, i.e, test oracles), to 81.2% with the complete dataset (770 valid invariants). This means a drop in precision of just 8% when using the smallest dataset (50 API requests) against the complete one (10K API requests). As in the previous section, it is worth highlighting that the precision of AGORA for the 50 API requests sets increases to 87.6% when excluding arithmetic comparisons, where the largest number of false positives was found. This means a precision difference of only 6% with respect to the precision achieved with the complete dataset suppressing arithmetic comparisons (93.6%).

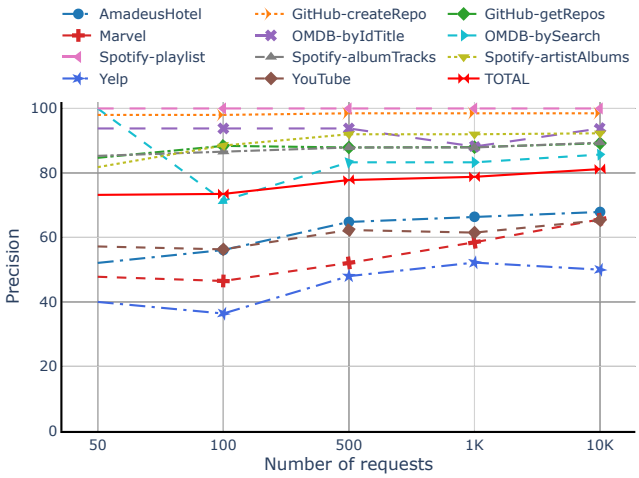

Figure 3: Evolution of the precision of AGORA.

The test suites of 50 requests seem to offer the best trade-off between effectiveness and test generation and execution costs. When comparing the smallest dataset with those with 100 requests or more, there is no increment in the precision value for 2 out of 11 operations, and the increase is less than 10% in 7 of them. This is explained by false positives, most of them arithmetic comparisons, for which it is difficult to find a response that rules them out.

Response to RQ2

The number of input API requests has a very limited impact on the effectiveness of AGORA. A small and diverse set of only 50 API requests suffixes to achieve a precision of 73.2% (87.6% excluding arithmetic comparisons), minimizing the effort required to generate and execute test cases.

4.3 Experiment 2: Failure detection

This experiment aims to answer RQ3 by analyzing the effectiveness of the test oracles generated by AGORA in detecting failures.

4.3.1 Experimental setup. To address RQ3, we evaluated the effectiveness of the generated test oracles in detecting failures (i.e., erroneous outputs) in the APIs under test. To this end, we systematically

Table 2: Test oracle generation. I="Number of likely invariants", P="Precision (% valid test oracles)"

API - Operation	50 API calls				100 API calls				500 API calls				1K API calls				10K API calls			
	Daikon		AGORA		Daikon		AGORA		Daikon		AGORA		Daikon		AGORA		Daikon		AGORA	
	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)
AmadeusHotel-getMultiHotelOffers	109	21.1	117	52.1	136	16.9	114	56.1	116	22.4	108	64.8	107	24.3	107	66.4	99	26.3	106	67.9
GitHub-createOrganizationRepository	82	95.1	198	98	82	95.1	198	98	80	96.2	198	98.5	80	96.2	198	98.5	80	96.2	198	98.5
GitHub-getOrganizationRepositories	45	40	150	84.7	40	45	147	88.4	39	46.2	149	87.9	39	46.2	150	88	38	47.4	148	89.2
Marvel-getComicById	178	29.8	115	47.8	194	28.9	127	46.5	178	33.7	119	52.1	167	35.9	106	58.5	140	45.7	96	65.6
OMDB-byIdOrTitle	7	57.1	16	93.8	7	57.1	16	93.8	7	57.1	16	93.8	8	50	17	88.2	7	57.1	16	93.8
OMDB-bySearch	4	100	5	100	7	57.1	7	71.4	5	80	6	83.3	5	80	6	83.3	6	83.3	7	85.7
Spotify-createPlaylist	22	100	41	100	22	100	41	100	22	100	41	100	22	100	41	100	22	100	41	100
Spotify-getAlbumTracks	46	45.7	68	85.3	45	46.7	67	86.6	42	50	66	87.9	42	50	66	87.9	41	53.7	66	89.4
Spotify-getArtistAlbums	53	43.4	55	81.8	49	49	52	88.5	35	68.6	50	92	32	75	50	92	31	83.9	52	92.3
Yelp-getBusinesses	60	28.3	30	40	55	30.9	33	36.4	46	37	25	48	45	37.8	23	52.2	41	39	22	50
YouTube-listVideos	228	31.6	194	57.2	227	32.2	199	56.3	218	35.8	191	62.3	225	36	200	61.5	201	41.3	196	65.3
TOTAL	**834**	**40.2**	**989**	**73.2**	**864**	**39.4**	**1001**	**73.5**	**788**	**44.5**	**969**	**77.8**	**772**	**45.9**	**964**	**78.8**	**706**	**51.4**	**948**	**81.2**

seeded *errors* in API responses using JSONMutator [6], an open-source mutation tool that applies different mutation operators on JSON data, e.g., removing an array item. This approach differs from traditional mutation testing, where *faults* are seeded in the source code of the program under test. The motivation behind our strategy is to assess the failure detection capabilities of the generated test oracles on large-scale industrial APIs, for which source code is not available. Although open-source APIs exist, they are generally less complex compared to the APIs used in our study [51, 55]. Also, we argue that this strategy—introducing errors in API responses—is appropriate since our goal is assessing the effectiveness of the test oracles, not the test inputs, which has already been thoroughly investigated in previous studies.

For each API operation, we selected the test oracles derived from the set of 50 test cases since, as revealed in our previous experiment, this was the most cost-effective input dataset. Test oracles were transformed into executable assertions in Java, 724 in total (Table 3). Then, for each API operation, we randomly selected 1K API responses from the set of 10K test cases generated by RESTest meeting the following constraints: (1) they were not part of the 50-requests set used for detecting the invariants, (2) they contained at least one result item (we cannot apply mutation operators on empty arrays), and (3) they revealed no failures (c.f. Section 5).

We used JSONMutator to introduce a single error on each API response simulating a failure. Then, we ran the assertions and marked the failure as detected if at least one of the test assertions (i.e., test oracles) was violated. We repeated this process 100 times per operation to minimize the effect of randomness computing the average percentage of failures detected. In total, the results are based on 1.1M seeded errors: 11 operations x 1,000 API responses x 100 repetitions.

For our experiments, we configured JSONMutator to apply mutation operators that resulted in syntactically valid mutants, i.e., conform to the API specification. Syntactically invalid mutants (e.g., adding a new property to a JSON object) can be detected by existing approaches and therefore are out of the scope of AGORA. Specifically, we enabled the mutation operators that consist of changing Boolean, double, long and string values (e.g., adding or removing characters) and altering array values (e.g., removing and disordering elements), using a total of 12 mutation operators. All the mutations resulted in a distinguishable change in the API response and therefore there were no equivalent mutants. We disabled the mutation operators that produced mutants non-conformant with the OAS specification. Also, we disabled operators that converted response fields into null values, since null values are easily detected as a violations of the `nullable` property of OAS. Our supplementary material contains a detailed list of all the mutation operators applied [7].

Table 3: Failure Detection Ratio per API operation.

API - Operation	Assertions (test oracles)	FDR (%)
AmadeusHotel-getMultiHotelOffers	61	60
GitHub-createOrganizationRepository	194	92.3
GitHub-getOrganizationRepositories	127	63.9
Marvel-getComicById	55	37
OMDB-byIdOrTitle	15	36.2
OMDB-bySearch	5	20.8
Spotify-createPlaylist	41	84.7
Spotify-getAlbumTracks	58	70.2
Spotify-getArtistAlbums	45	76.6
Yelp-getBusinesses	12	23.2
YouTube-listVideos	111	65.4
TOTAL	**724**	**57.3**

4.3.2 Experimental results. Table 3 shows the number of test assertions (i.e., test oracles) and the percentage of detected failures for each API operation. Overall, test oracles generated by AGORA identified 57.3% of the failures. This percentage ranged between 20.8% in the "bySearch" operation of the OMDb API and 92.3% in the "createOrganizationRepository" of the GitHub API. One of the main causes behind undetected errors was introducing changes in unique string values (e.g., "title=Taxi Driver" -> "title=TaAxi Driver") for which inferring test oracles is challenging.

Response to RQ3

The test oracles generated by AGORA are effective in detecting failures, catching 6 out of every 10 errors systematically seeded in API responses.

5 DETECTED FAULTS

The invariants detected by AGORA allowed us to detect bugs in some of the APIs under test, showing the potential of the approach as a testing technique on its own. Some of the invariants revealed

inconsistent behavior, e.g., hotel rooms with *zero* beds. We also found cases where a confirmed invariant (i.e., test oracle) was discarded when increasing the size of the input dataset, meaning that a counterexample (i.e., failure) had been detected. Therefore, the invariants reported by AGORA play a dual role in fault detection: invalid invariants (which require manual inspection) reveal faults observed during the invariant detection process, whereas violated valid invariants (which can be automatically detected) indicate faults observed in production. Overall, AGORA detected 11 domain-specific bugs in 7 operations from 5 APIs with millions of users worldwide, namely Amadeus, GitHub, Marvel, OMDb and YouTube. Our supplementary material contains videos showing the replication process of these bugs, as well as anonymized screenshots of our reports and the received responses [7]. Next, we detail the detected bugs.

Amadeus Hotel. During our initial experiments, one of the detected invariants in the Amadeus Hotel API led to the identification of 55 hotel offers in which the offered room had zero beds (`return.room.typeEstimated.beds>=0`). This bug has been confirmed and fixed by Amadeus developers.

GitHub. In the "createOrganizationRepository" operation, the violation of the confirmed invariant `input.license_template==return.license.key` revealed 15 test cases in which the repository is created with an incorrect license. This bug has been confirmed by the API providers. Also, contrary to what is stated in the API specification and the documentation, AGORA detected that the field `template_directory` was never included in the responses of the "getOrganizationRepositories" operation (`return.template_repository==null`). Developers confirmed the issue and updated the documentation of GitHub accordingly.

Marvel. In the "getComicById" operation, AGORA detected +3.1K comics with 0 pages (`return.pageCount>=0`), invalid date formats, comics with an invalid Diamond code (violations of the invariant `LENGTH(return.diamondCode)==9`), and invalid values for the EAN code (violations of the invariant `LENGTH(return.ean)==20`). For example, we found a case where the EAN code had the value of the Diamond code. These reports have not been confirmed yet.

OMDb. The `type` parameter of the OMDb API operations is used to filter the obtained results to one media type: "movie", "series" or "episode" (according to the documentation). However, one of the invariants (`return.Type one of {"game","movie","series"}`) revealed a new value for this parameter that was not specified in the documentation: "game". Moreover, we detected that the operations "byIdOrTitle" and "bySearch" do not support filtering by "episode". These reports have not been confirmed yet.

YouTube. When performing a search using the `regionCode` input parameter, the returned videos must be available in the provided region. However, a violation of the confirmed invariant `input.regionCode in return.contentDetails.regionRestriction.allowed[]`, led us to detect 81 cases in which the API returned videos that were not available in the provided region. This error has been confirmed by YouTube developers.

6 THREATS TO VALIDITY

In this section, we discuss the potential validity threats that may have influenced our work, and how these were mitigated.

Internal validity. *Are there factors that might affect the results of our evaluation?* For our experiments, we used the OAS specification of the APIs under test. When possible, we resorted to the publicly available API specifications. However, the specifications of the OMDb and Yelp APIs were unavailable, so we generated them manually based on an analysis of the web documentation. Therefore, it is possible that these specifications have errors and deviate from the API documentation. To mitigate this threat, the specifications files were thoroughly reviewed by at least two authors.

The effectiveness of our approach largely depends on the diversity of the input API requests and responses. To maximize input diversity, we manually selected a set of varied test inputs for each parameter based on an analysis of the documentation. This may be considered a naive and conservative approach. Using more systematic or automated means (e.g., adaptive random testing [45]) could probably yield even better results.

The classification of the reported invariants as true positives or false positives may be affected by human biases or errors. To mitigate this threat, each invariant was checked by at least two authors, analyzing the API documentation, or consulting the API developers in case of discrepancy.

Finally, the division of the dataset into random subgroups may have also affected the results. We did not apply multiple executions of this experiment given the manual work required to classify the invariants reported in each execution. To mitigate this threat, the experiment was performed with 11 API operations belonging to different application domains.

External validity. *To what extent can we generalize the findings of our investigation?* We evaluated AGORA on a set of 11 operations from 7 different APIs, and therefore our conclusions could not generalize beyond that. To mitigate this threat, we evaluated the approach with a set of popular industrial APIs of different domains and various sizes used in related papers.

The novel types of invariant proposed could not generalize beyond the selected APIs. To mitigate this threat, these invariants were created based on an analysis of a systematically collected dataset of realistic APIs belonging to different domains [12]. We remark, however, that this set of invariants is not intended to be complete and new invariant types could be proposed in the future.

7 CONCLUSIONS AND FUTURE WORK

This paper introduces AGORA, a novel approach for generating test oracles for REST APIs through the detection of likely invariants. Invariants are detected by analyzing the API specification and a set of API requests with their corresponding responses. The approach is implemented using Daikon, an open-source tool for dynamic invariant detection. In particular, we propose Beet, a novel Daikon instrumenter for REST APIs described using OAS, and a customized version of Daikon supporting the detection of 105 distinct types of invariants in REST APIs. Evaluation results on a set of 11 operations from 7 industrial APIs show that AGORA can generate hundreds of effective test oracles with just 50 requests in seconds. In addition,

AGORA helped identify 11 faults in industrial APIs with millions of users, contributing to fixes and documentation updates, demonstrating its potential as a standalone testing technique. AGORA operates in a black-box mode and can be easily integrated into existing API testing tools supporting the OAS specification format.

Future lines of work include the automated generation of assertions from the reported invariants, and the deployment of AGORA as a Web API to ease its integration into others applications.

ACKNOWLEDGMENTS

This work has been partially supported by grants PID2021-126227NB-C22 and PID2021-126227NB-C21, funded by MCIN/AEI /10.13039/501100011033/FEDER, UE; and grant TED2021-131023B-C21, funded by MCIN/AEI/10.13039/501100011033 and by European Union "NextGenerationEU"/PRTR».

REFERENCES

[1] 2022. DAIKON instrumenters. https://plse.cs.washington.edu/daikon/download/doc/daikon.html#Front-ends-_0028instrumentation_0029. Accessed September 2022.
[2] 2022. OpenAPI Specification. https://www.openapis.org. Accessed September 2022.
[3] 2022. RapidAPI API directory. https://rapidapi.com/marketplace. Accessed November 2022.
[4] 2022. Spotify Web API. https://developer.spotify.com/web-api/. accessed September 2022.
[5] 2023. Beet repository. https://github.com/isa-group/Beet
[6] 2023. JSONMutator. https://github.com/isa-group/JSONmutator. Accessed January 2023.
[7] 2023. Replication package. https://doi.org/10.5281/zenodo.7970822
[8] 2023. SRC Grand Finalists 2023. https://src.acm.org/grand-finalists/2023
[9] 2023. Visa Developer Center. https://developer.visa.com. accessed January 2023.
[10] Afsoon Afzal, Claire Le Goues, and Christopher Steven Timperley. 2021. Mithra: Anomaly Detection as an Oracle for Cyberphysical Systems. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3120680
[11] Juan C. Alonso. 2022. Automated Generation of Test Oracles for RESTful APIs. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1808–1810. https://doi.org/10.1145/3540250.3559080
[12] Juan C. Alonso, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. 2022. ARTE: Automated Generation of Realistic Test Inputs for Web APIs. *IEEE Transactions on Software Engineering* (2022). https://doi.org/10.1109/TSE.2022.3150618
[13] apisguru 2022. APIs.guru. https://apis.guru. accessed October 2022.
[14] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology* 28, 1 (2019), 1–37. https://doi.org/10.1145/3293455
[15] V. Atlidakis, P. Godefroid, and M. Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 748–758. https://doi.org/10.1109/ICSE.2019.00083
[16] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020. Checking Security Properties of Cloud Services REST APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 387–397. https://doi.org/10.1109/ICST46399.2020.00046
[17] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. 2021. Generating Metamorphic Relations for Cyber-Physical Systems with Genetic Programming: An Industrial Case Study. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1264–1274. https://doi.org/10.1145/3468264.3473920
[18] Efe Barlas, Xin Du, and James C. Davis. 2022. Exploiting Input Sanitization for Regex Denial of Service. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 883–895. https://doi.org/10.1145/3510003.3510047
[19] 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. https://doi.org/10.1109/TSE.2014.2372785
[20] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *ISSTA 2018, Proceedings of the 2018 International Symposium on Software Testing and Analysis*. Amsterdam, Netherlands, 242–253.
[21] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *Journal of Systems and Software* 181 (Nov. 2021), 111041:1–13.
[22] Houssem Ben Braiek and Foutse Khomh. 2020. On testing machine learning programs. *Journal of Systems and Software* 164 (2020), 110542. https://doi.org/10.1016/j.jss.2020.110542
[23] Marcel Böhme, Charaka Geethal, and Van-Thuan Pham. 2020. Human-In-The-Loop Automatic Program Repair. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 274–285. https://doi.org/10.1109/ICST46399.2020.00036
[24] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting Static Analysis Accuracy with Instrumented Test Executions. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1154–1165. https://doi.org/10.1145/3468264.3468626
[25] Jake Cobb, James A. Jones, Gregory M. Kapfhammer, and Mary Jean Harrold. 2011. Dynamic Invariant Detection for Relational Databases. In *Proceedings of the Ninth International Workshop on Dynamic Analysis* (Toronto, Ontario, Canada) *(WODA '11)*. Association for Computing Machinery, New York, NY, USA, 12–17. https://doi.org/10.1145/2002951.2002955
[26] Davide Corradini, Michele Pasqua, and Mariano Ceccato. 2023. Automated Black-box Testing of Mass Assignment Vulnerabilities in RESTful APIs. https://doi.org/10.48550/ARXIV.2301.01261
[27] Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *Journal of Logic and Computation* 2 (08 1992). https://doi.org/10.1093/logcom/2.4.511
[28] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2130–2141. https://doi.org/10.1145/3510003.3510141
[29] Michael D. Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *WODA 2003: Workshop on Dynamic Analysis*. Portland, OR, USA, 24–27.
[30] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (Feb. 2001), 99–123.
[31] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015 Special issue on Experimental Software and Toolkits.
[32] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph. D. Dissertation. University of California, Irvine.
[33] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291. https://doi.org/10.1109/TSE.2012.14
[34] Gregory Gay, Sanjai Rayadurgam, and Mats P.E. Heimdahl. 2014. Improving the Accuracy of Oracle Verdicts through Automated Model Steering. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 527–538. https://doi.org/10.1145/2642937.2642989
[35] Luca Gazzola, Maayan Goldstein, Leonardo Mariani, Itai Segall, and Luca Ussi. 2020. Automatic Ex-Vivo Regression Testing of Microservices. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test* (Seoul, Republic of Korea) *(AST '20)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/3387903.3389309
[36] github-api 2022. GitHub API. https://developer.github.com/v3/ accessed September 2022.
[37] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum. 2013. Practical automated vulnerability monitoring using program state invariants. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12. https://doi.org/10.1109/DSN.2013.6575318
[38] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API Data Fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 725–736. https://doi.org/10.1145/3368089.3409719
[39] 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* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 312–323.

https://doi.org/10.1145/3395363.3397374
[40] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *ISSTA 2016, Proceedings of the 2016 International Symposium on Software Testing and Analysis*. Saarbrücken, Genmany, 213–224.
[41] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2022. Testing RESTful APIs: A Survey. https://doi.org/10.48550/ARXIV.2212.14604
[42] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. 2018. Inferring and Asserting Distributed System Invariants. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1149–1159. https://doi.org/10.1145/3180155.3180199
[43] J. Haltermann and H. Wehrheim. 2022. Machine Learning Based Invariant Generation: A Framework and Reproducibility Study. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 12–23. https://doi.org/10.1109/ICST53961.2022.00012
[44] Zac Hatfield-Dodds and Dmitry Dygalo. 2021. Deriving Semantics-Aware Fuzzers from Web API Schemas. *arXiv preprint arXiv:2112.10328* (2021).
[45] Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, and Xin Xia. 2021. A Survey on Adaptive Random Testing. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2052–2083. https://doi.org/10.1109/TSE.2019.2942921
[46] Ali Reza Ibrahimzada, Yigit Varli, Dilara Tekinoglu, and Reyhaneh Jabbarvand. 2022. Perfect is the Enemy of Test Oracle. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 70–81. https://doi.org/10.1145/3540250.3549086
[47] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.
[48] Daniel Jacobson, Greg Brail, and Dan Woods. 2011. *APIs: A Strategy Guide*. O'Reilly Media, Inc.
[49] Charaka Geethal Kapugama, Van-Thuan Pham, Aldeida Aleti, and Marcel Böhme. 2022. Human-in-the-Loop Oracle Learning for Semantic Bugs in String Processing Programs. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/3533767.3534406
[50] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs. In *International Conference on Software Testing, Validation and Verification*. 131–141.
[51] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 289–301. https://doi.org/10.1145/3533767.3534401
[52] Sumit Lahiri and Subhajit Roy. 2022. Almost Correct Invariants: Synthesizing Inductive Invariants by Fuzzing Proofs. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 352–364. https://doi.org/10.1145/3533767.3534381
[53] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: Model-Based RESTful API Testing with Execution Feedback. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1406–1417. https://doi.org/10.1145/3510003.3510133
[54] R. Mahmood, J. Pennington, D. Tsang, T. Tran, and A. Bogle. 2022. A Framework for Automated API Fuzzing at Enterprise Scale. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 377–388. https://doi.org/10.1109/ICST53961.2022.00018
[55] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. 2021. Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?. In *International Symposium on Software Reliability Engineering*.
[56] Alberto Martin-Lopez, Sergio Segura, Carlos Müller, and Antonio Ruiz-Cortés. 2021. Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. *IEEE Transactions on Services Computing* (2021). https://doi.org/10.1109/TSC.2021.3050610
[57] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In *International Conference on Service-Oriented Computing*. 459–475.
[58] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: Automated Black-Box Testing of RESTful Web APIs. In *International Symposium on Software Testing and Analysis*.
[59] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2022. Online Testing of RESTful APIs: Promises and Challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 408–420. https://doi.org/10.1145/3540250.3549144
[60] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 336–347. https://doi.org/10.1109/ICSE43902.2021.00041
[61] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2022. Fuzzing Class Specifications. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1008–1020. https://doi.org/10.1145/3510003.3510120
[62] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo Frias. 2019. Training Binary Classifiers as Data Structure Invariants. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 759–770. https://doi.org/10.1109/ICSE.2019.00084
[63] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1223–1235. https://doi.org/10.1109/ICSE43902.2021.00112
[64] Jeremy W. Nimmer and Michael D. Ernst. 2001. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *RV 2001: Proceedings of the First Workshop on Runtime Verification*. Paris, France.
[65] Jeremy W. Nimmer and Michael D. Ernst. 2002. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*. Rome, Italy, 232–242.
[66] Manuel Palomo-Duarte, Antonio García-Domínguez, Inmaculada Medina-Bulo, Alejandro Alvarez-Ayllón, and Javier Santacruz. 2010. Takuan: A Tool for WS-BPEL Composition Testing Using Dynamic Invariant Generation. In *Web Engineering*, Boualem Benatallah, Fabio Casati, Gerti Kappel, and Gustavo Rossi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 531–534.
[67] Antonio Pecchia, Stefano Russo, and Santonu Sarkar. 2020. Assessing Invariant Mining Techniques for Cloud-Based Utility Computing Systems. *IEEE Transactions on Services Computing* 13, 1 (2020), 44–58. https://doi.org/10.1109/TSC.2017.2679715
[68] Leonard Richardson, Mike Amundsen, and Sam Ruby. 2013. *RESTful Web APIs*. O'Reilly Media, Inc.
[69] Sergio Segura, Juan C. Alonso, Alberto Martin-Lopez, Amador Durán, Javier Troya, and Antonio Ruiz-Cortés. 2022. Automated Generation of Metamorphic Relations for Query-Based Systems. In *2022 IEEE/ACM 7th International Workshop on Metamorphic Testing (MET)*. 48–55. https://doi.org/10.1145/3524846.3527338
[70] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875
[71] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2018. Metamorphic Testing of RESTful Web APIs. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1083–1099. https://doi.org/10.1109/TSE.2017.2764464
[72] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2022. Improving Test Case Generation for REST APIs through Hierarchical Clustering. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 117–128. https://doi.org/10.1109/ASE51524.2021.9678586
[73] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary Improvement of Assertion Oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1178–1189. https://doi.org/10.1145/3368089.3409758
[74] Theofanis Vassiliou-Gioles. 2020. A simple, lightweight framework for testing RESTful services with TTCN-3. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 498–505. https://doi.org/10.1109/QRS-C51114.2020.00089
[75] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RestTestGen: Automated Black-Box Testing of RESTful APIs. In *International Conference on Software Testing, Verification and Validation*.
[76] Henry Vu, Tobias Fertig, and Peter Braun. 2018. Verification of Hypermedia Characteristic of RESTful Finite-State Machines. In *Companion Proceedings of the The Web Conference 2018* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1881–1886. https://doi.org/10.1145/3184558.3191656
[77] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On Learning Meaningful Assert Statements for Unit Test Cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. https://doi.org/10.1145/3377811.3380429
[78] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial Testing of RESTful APIs. In *Proceedings of the 44th International Conference*

on Software Engineering (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 426–437. https://doi.org/10.1145/3510003.3510151

[79] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.* 22, 2 (mar 2012), 67–120. https://doi.org/10.1002/stv.430

[80] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. 2022. Automated Assertion Generation via Information Retrieval and Its Integration with Deep Learning. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/3510003.3510149

[81] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: Translating Natural Language Comments to Formal Program Specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 25–37. https://doi.org/10.1145/3368089.3409716

[82] Man Zhang and Andrea Arcuri. 2021. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 2 (sep 2021), 52 pages. https://doi.org/10.1145/3464940

[83] M. Zhang, A. Belhadi, and A. Arcuri. 2022. JavaScript Instrumentation for Search-Based Software Testing: A Study with RESTful APIs. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 105–115. https://doi.org/10.1109/ICST53961.2022.00022

[84] Yuntong Zhang, Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. 2022. Program Vulnerability Repair via Inductive Inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 691–702. https://doi.org/10.1145/3533767.3534387