Test Oracle Generation for REST APIs

JUAN C. ALONSO, SCORE Lab, I3US Institute, Universidad de Sevilla, Spain
MICHAEL D. ERNST, University of Washington, USA
SERGIO SEGURA, SCORE Lab, I3US Institute, Universidad de Sevilla, Spain
ANTONIO RUIZ-CORTÉS, SCORE Lab, I3US Institute, Universidad de Sevilla, Spain

The number and complexity of test case generation tools for REST APIs have significantly increased in recent years. These tools excel in automating input generation but are limited by their test oracles, which can only detect crashes, regressions, and violations of API specifications or design best practices. This article introduces AGORA+, an approach for generating test oracles for REST APIs through the detection of invariants—output properties that should always hold. AGORA+ learns the expected behavior of an API by analyzing API requests and their corresponding responses. We enhanced the Daikon tool for dynamic detection of likely invariants, adding new invariant types and creating a front-end called Beet. Beet translates any OpenAPI specification and a set of API requests and responses into Daikon inputs. AGORA+ can detect 106 different types of invariants in REST APIs. We also developed PostmanAssertify, which converts the invariants identified by AGORA+ into executable JavaScript assertions. AGORA+ achieved a precision of 80% on 25 operations from 20 industrial APIs. It also identified 48% of errors systematically seeded in the outputs of the APIs under test. AGORA+ uncovered 32 bugs in popular APIs, including Amadeus, Deutschebahn, GitHub, Marvel, NYTimesBooks, and YouTube, leading to fixes and documentation updates.

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

Additional Key Words and Phrases: REST APIs, test oracle, invariant detection, automated testing

ACM Reference Format:
Juan C. Alonso, Michael D. Ernst, Sergio Segura, and Antonio Ruiz-Cortés. 2024. Test Oracle Generation for REST APIs. *J. ACM* 37, 4, Article 111 (August 2024), 36 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Web Application Programming Interfaces (APIs) allow heterogeneous software systems to interact over the network [67, 96]. Most modern web APIs are *REST APIs* [51] that adhere to the REpresentational State Transfer (REST) architectural style. REST APIs are decomposed into multiple resources (e.g., a *payment* in the VISA API [25]) that clients can manipulate through HTTP interactions. REST APIs have become the de facto standard for software integration. They are a key part of the business model of companies such as Amazon, Google, and Netflix [67]. The RapidAPI [14] repository hosts over 40K REST APIs.

Authors' addresses: Juan C. Alonso, javalenzuela@us.es, SCORE Lab, I3US Institute, Universidad de Sevilla, Seville, Spain; Michael D. Ernst, mernst@cs.washington.edu, University of Washington, USA; 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.

The importance of REST APIs has driven the creation of numerous techniques and tools for the automated detection of failures within these APIs [59, 75]. 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 [11]. 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., tests of correctness [37]. These tools are all limited in the types of failures that they can detect, namely crashes (responses with a 5XX HTTP status code) [32, 33, 62, 69, 72, 74, 82, 103, 106], disconformities with the API specification (e.g., a missing output JSON property) [32, 62, 69, 82, 103], regressions [54, 57], and violations of API best practices (e.g., checking that the results of multiple calls to idempotent operations are identical) [34, 36, 102, 112]. As an example, Listing 2 shows the response of the "getAlbumTracks" operation of the Spotify API for the request "GET https://api.spotify.com/albums/4Em5W5HgYEvhpc/tracks?limit=1&market=ES". 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 (e.g., the country codes of the `available_markets` response field should have length 2) or format (e.g., the values of the `href` and `uri` response fields should be URLs) and violations of numerical constraints (e.g., the `total` and `duration_ms` response fields cannot be negative) or array properties (e.g., the `total` response field should be greater than or equal to the size of the `items` array), among others. None of these errors would be detected by any existing approach proposed in the literature, since all the responses are syntactically valid and conform to the corresponding API specifications. Recent surveys [59] and tool comparisons [75, 84] 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 [46, 105], program specifications [53, 66], documentation [38, 58], and previous executions [86, 88]. One 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 language-independent and black-box (applicable when the source code is not available, such as REST APIs). This is the strategy we use.

This article presents AGORA+, a black-box approach for the Automated Generation of Oracles for REST APIs through the detection of likely invariants. AGORA+ was created by extending and modifying the Daikon [50] system for dynamic invariant detection in two directions. Firstly, we present a novel software tool—a Daikon *front-end* called Beet—that converts an OAS specification, plus 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 specifications. Secondly, we further enhanced the capabilities of Daikon by customizing and expanding its set of invariant templates. Currently, AGORA+ supports the detection of 106 distinct types of invariants in REST APIs. Finally, to foster its use in practice, we introduce PostmanAssertify. This is a tool that transforms the invariants detected by AGORA+ into executable JavaScript scripts with assertions, written using the Chai library [2], that are compatible with Postman [12], a popular API platform in industry.

Evaluation results using 25 operations from 20 industrial APIs showed that just 50 API requests and responses is sufficient for AGORA+ to learn hundreds of accurate invariants (test oracles), achieving a precision of 68%. This precision improves to 80% when learning from 10K API requests. These results surpass those obtained using the default set of invariants in Daikon, whose precision is less than 40%. We also evaluated the effectiveness of the generated test oracles in detecting failures by automatically seeding 2.5M faults 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 48% of the incorrect outputs, supporting the cost-effectiveness of our approach.

During our evaluation, AGORA+ generated several invariants that, when manually analyzed, indicated issues within the target APIs. One example was the invariant `return.room.typeEstimated.beds >= 0`, which revealed a confirmed bug in the Amadeus API where certain hotel offers included rooms with *zero* beds. Overall, AGORA+ resulted in the detection of 32 faults (17 confirmed, 10 fixed) in 13 operations of 11 industrial APIs, that led to bug fixes in many of them, including updates in the documentation of GitHub. All these bugs would have passed unnoticed by current test case generators. This highlights the value of AGORA+ not only as a test oracle generation approach, but also as a testing technique on its own.

This paper first introduces background and related work on testing REST APIs, test oracle generation, and dynamic invariant detection (Section 2). Then, the 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 (Section 3.1), a Daikon front-end for REST APIs readily integrable into existing test case generation tools for REST APIs supporting OAS specifications. Beet is open-source and available on GitHub [1].
- Several Daikon extensions [3, 4], including reporting invariants in CSV format, support for the detection of 22 new types of invariants (Section 3.2), new heuristics for automatically suppressing common false positive patterns (Section 3.3), as well as a new output format that returns executable assertions compatible with the JavaScript Chai library (Section 3.4).
- PostmanAssertify, a tool that converts the invariants reported by AGORA+ into executable JavaScript test assertions compatible with the popular API platform Postman [12]. PostmanAssertify is open-source and publicly available on GitHub [13].
- An empirical evaluation of AGORA+ in terms of precision and fault detection in 25 operations from 20 industrial APIs (Section 4), including discovery of 32 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 [15].

Section 6 addresses threats to validity, and Section 7 concludes.

A preliminary version of this work appeared in ISSTA 2023 [31] (Distinguished Artifact Award) and in the ACM Student Research Competition [22, 29] (first prize winner in ESEC/FSE 2022, second prize winner in SRC Grand Finals 2023). This paper extends our previous work in several directions. First, we evaluated with 14 new operations from 13 industrial APIs (Section 4). Second, we introduced new heuristics that improve precision by 33% (Section 3.3). Third, we created a new tool, PostmanAssertify, for the automated generation of executable assertions in JavaScript, specifically through the generation of Postman collections, a popular tool among practitioners (Section 3.4).

2 BACKGROUND AND RELATED WORK

This section reviews automated testing of REST APIs, test oracle generation, and dynamic invariant detection.

2.1 Automated Testing of REST APIs

Modern web APIs are typically compliant with the REpresentational State Transfer (REST) [51] architectural style and are known as REST APIs [96]. REST APIs are usually composed of multiple RESTful web services, with each one of them implementing create, read, update, and/or delete (CRUD) operations on a resource (e.g., in the GitHub API [8], each resource is a repository). These operations are typically invoked by sending HTTP requests to a Uniform Resource Locator (URL) that represents a resource or a collection of resources.

REST APIs are commonly described using the OpenAPI Specification (OAS) [11] format (previously known as Swagger), arguably the industry standard. An OAS specification documents operations in the API, including 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 [21]. The specification 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 21–63). Listing 2 shows a response for the "getAlbumTracks" operation conforming to the specification.

Automated testing of REST APIs usually adopts a black-box approach [30, 33, 34, 44, 56, 57, 62, 69–74, 77, 79, 83, 92, 99, 103, 106, 107]. Given an OAS specification, these techniques automatically generate pseudo-random test cases (sequences of HTTP requests) and test oracles (assertions on the HTTP responses). The approaches differ in the way they generate API calls (i.e., test inputs) using techniques such as model-based testing [77, 83, 104], property-based testing [62, 69, 97, 99], and constraint-based testing [81, 82]. Some methods focus on testing individual API operations and generate single API requests, while others generate sequences of API calls for stateful testing [33, 44, 62, 71, 74, 103]. White-box approaches require access to the API source code and are less common than black-box approaches. Most existing techniques utilize search algorithms to maximize failure detection and code coverage [32, 100, 113].

In terms of failure detection, generated test oracles are primarily limited to detecting API crashes (e.g., 500 status codes) and violations of the API specification [32, 62, 69, 82, 103]. Other test oracles focus on detecting regressions [54, 57] or adherence to best design practices [34, 36, 44, 102, 112]. However, all these approaches have limitations in detecting problems that go beyond mere syntax. For example, existing approaches would ignore domain-specific assertions in Listing 2, such as checking that the `linked_from.uri` response field should be a URL that contains the text of the `linked_from.id` field, or that the size of the `items` response field should be less than or equal to the value of the `total` response field. Generating such test oracles is the goal of AGORA+.

2.2 Test Oracle Generation

Automated test oracle generation techniques can be classified based on their inputs and their application domains. Regarding their inputs, test oracles have been derived from source code [46, 63, 85, 105, 109], formal specifications [53, 66], semi-structured documentation [38, 39, 58, 111], previous program executions [41, 42, 47, 65, 68, 86–88, 101], or a combination of them. Application contexts include Java projects [38, 46, 86], machine learning programs [40], databases [47] and cyber-physical systems [35], among others.

Other related techniques include metamorphic testing and regression testing. Metamorphic testing [92, 97–99] 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 [52, 108] relies on previous versions of the software under test to confirm that a change has not affected existing features.

Existing techniques for automated test oracle generation primarily operate at the method level and are tailored to specific programming languages, mainly Java [38, 39, 41, 42, 46, 58, 63, 65, 68, 86–88, 101, 105, 109, 111]. These techniques generate unit test assertions by leveraging information from the method under test, semi-structured procedure specifications (e.g., Javadoc), unit test methods lacking assertions (test prefixes), or a combination of these. Documentation-based approaches [38, 39, 58, 111] use variable names and dataflow analysis to infer test oracles, making them unsuitable for black-box testing. Approaches that infer test oracles from previous program executions require

```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
                    duration_ms:
                      type: integer
                    linked_from: # Nested object
                      type: object
                      properties:
                        id:
                          type: string
                        uri:
                          type: string
```

Listing 1. An excerpt of the getAlbumTracks operation of the Spotify API, expressed in OAS (OpenAPI Specification) format. Listing 2 shows a response.

generating valid and invalid instances of program states [86–88], usually generated through code mutation, or rely on prior static analysis [42], a human in the loop [41, 68], or previously annotated false positives and false negatives [101] to refine automatically generated assertions. Recent LLM-based techniques [46, 63, 105, 109] also focus on specific programming languages and operate at the method level, making them not applicable to the domain of REST APIs.

A common approach for the generation of test oracles is through the detection of invariants. An *invariant* is a property that is always satisfied at one or more points of the execution of a program [49]. For example, given a procedure 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[])`.

```
{
   "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,
         "duration_ms": 234346,
         "linked_from": {
            "id": "98cZPdKywnMGp8fnw2XTYU",
            "uri": "https://spotify.com/artist/98cZPdKywnMGp8fnw2XTYU"
         }
      }
   ]
}
```

Listing 2. Spotify API response in JSON format, for the request GET https://api.spotify.com/albums/4Em5W5HgYEvhpc/tracks?limit=1&market=ES.

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) [45, 55] or dynamically (analyzing the behavior of a program through multiple executions) [49, 50, 61, 76]. Statically detected invariants are usually less numerous and less specific than those detected by dynamic techniques [48, 89, 90]. However, dynamic invariant detection techniques may result in overfitting, especially if the analyzed executions 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. Likely invariants can be categorized based on where in the program execution they are observed. For instance, *likely preconditions* apply to the state before a method or function is executed, while *likely postconditions* describe the state after its completion. Both are specialized forms of likely invariants, scoped to specific program components such as functions or methods. Similarly, *loop invariants* are properties that hold true before and after every iteration of a loop.

The automated detection of likely invariants has shown promising results in contexts such as Java programs [86], relational databases [43], automated program repair [114], WS-BPEL composition testing [91], cyber-physical systems [28], and cloud-based [95] and distributed [60] systems. To the best of our knowledge, this is the first approach for the automated detection of likely invariants in REST APIs. AGORA+ works on a black-box mode, where only the inputs and outputs of the API under test are accessible. This makes our approach compatible with existing REST API testing tools, regardless of how they are implemented, fostering its applicability in practice.

2.3 Daikon

Daikon [50] 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 a *front-end* or instrumenter. This instrumentation produces a declaration file and a data trace file. The *declaration* file describes the variables that exist

```
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):::EXIT1
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.

at each program point. The *data trace* file contains the values in each execution. Front-ends are available for various programming languages and data formats, including Java, Perl, C++, and CSV [6].

Listing 3 shows a Java method that squares an input integer. For this method, a Daikon front-end generates an ENTER program point for method entry and an EXIT program point for method exit. Listing 4 depicts the content of a Daikon declaration file for the EXIT program point. As illustrated, the definition of the program point includes 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`), datatype 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 field `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 since the last time it was observed. After processing these files, assuming a larger data trace file, Daikon would return a set of invariants like the one presented in Listing 6.

3 AGORA+

Figure 1 overviews 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 front-end. 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. Beet returns a declaration file (describing the format of the API operation 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. Optionally, these confirmed invariants can be used as input to PostmanAssertify, which generates a Postman collection with the invariants implemented as executable assertions in JavaScript.

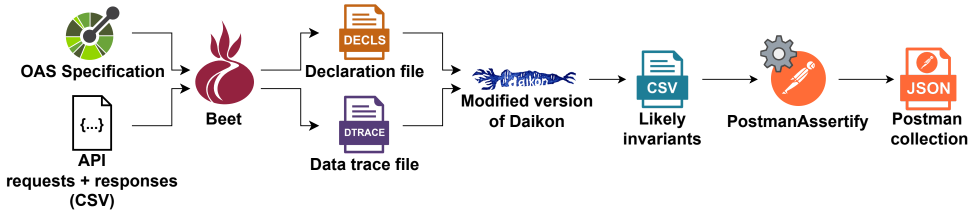


Fig. 1. Workflow of AGORA+.

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 [30, 69, 75, 81–84, 99]. In AGORA+, the detected likely invariants fall into two categories: likely preconditions, which are properties satisfied by the input parameters of an operation, and likely postconditions, which are properties consistently satisfied by the API responses. 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 remains for future work. 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.

We now describe the Beet front-end, the types of invariants currently supported, the heuristics applied by AGORA+ to minimize the number of false positives reported, and the tool PostmanAssertify.

3.1 Beet: A Daikon front-end for REST APIs

This section explains how Beet generates 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/fields 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/fields of the return variable as response fields (`return.<paramName>`). The value of `<primitiveType>` in Table 1 can be java.lang.String, int, double, or boolean. Variables of type object are represented using hash codes.

[1]Daikon is an Asian radish. Previous Daikon front-ends have adopted the name of root vegetables, such as Celeriac, Chicory, and Takuan [6]. We decided to follow this convention.

Table 1. Daikon declaration file format for API Requests and Responses.

API Request	API operation	ppt <operationName>&<statusCode>():::ENTER
		ppt-type enter
		variable input
		var-kind variable
		dec-type <operationName>&Input
		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
	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[]
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
	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
	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[]

```
GET https://api.spotify.com/albums/{id}/tracks?limit={limit}&market={market}
```

(a) API request

```
ppt getAlbumTracks&200():::ENTER
ppt-type enter
variable input
  var-kind variable
  dec-type getAlbumTracks&Input
  rep-type hashcode
variable input.id
  var-kind field id
  enclosing-var input
  dec-type java.lang.String
  rep-type java.lang.String
variable input.market
  var-kind field market
  enclosing-var input
  dec-type java.lang.String
  rep-type java.lang.String
variable input.limit
  var-kind field limit
  enclosing-var input
  dec-type int
  rep-type int
```

(b) Declaration file

Listing 7. ENTER program point of an API operation.

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. Unlike JSON objects, arrays of objects cannot be flattened by creating additional entries in the EXIT with indexed positions (e.g., `return.items[1].name`) because the number of array elements varies across responses, and the declaration file cannot dynamically accommodate a changing number of variable entries. Even if such flattening were feasible, comparing all

```
1  responses:
2  "200":
3    description: "OK"
4    schema:
5      type: object
6      properties:
7        total:
8          type: integer
9        href:
10         type: string
11       items:
12         type: array
13         items:
14           type: object
15           properties:
16             artists:
17               type: array
18               items:
19                 type: object
20                 properties:
21                   id:
22                     type: string
23                   name:
24                     type: string
25             available_markets:
26               type: array
27               items:
28                 type: string
29 ...
```

(a) YAML response schema

```
1  ppt getAlbumTracks&200():::EXIT1
2  ppt-type subexit
3  variable input
4  ... // variables from the entry point are repeated here
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[]
```

(b) Declaration file

Listing 8. EXIT program point of an API operation.

properties across all elements in nested arrays would lead to a combinatorial explosion of reported invariants, many of which would be false positives or irrelevant, while also incurring a substantial performance overhead. To support more informative array-related output invariants, Beet implements a recursive strategy by creating a new 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 (i.e., API request) and EXIT (i.e., API response) program points for the "getAlbumTracks" operation of the Spotify API (Listing 1). The "&" in the program point names has no semantic significance to Daikon. In Listing 7, the definition of the ENTER program point is followed by the definition of the input parameters. 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. The left part of the Listing also includes the corresponding fragment of the response format being represented. The details of the variables of type string (`return.id`, `return.name` and `return.linked_from.uri`), integer (`return.duration_ms`), boolean (`return.explicit`) and array of objects (`return.artists`) have been omitted for brevity.

The use of nested structures improves the relevance of reported invariants and avoids performance overheads. Placing all response fields as variables in a single program point would likely cause a combinatorial explosion of reported

[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.

```
responses:
"200":
  description: "OK"
  schema:
    type: object
    properties:
      total:
        type: integer
      href:
        type: string
      items:
        type: array
        items:
          type: object
          properties:
            artists:
              type: array
              items:
                type: object
                properties:
                  id:
                    type: string
                  name:
                    type: string
            available_markets:
              type: array
              items:
                type: string
            id:
              type: string
            name:
              type: string
            explicit:
              type: boolean
            duration_ms:
              type: integer
            linked_from:
              type: object
              properties:
                id:
                  type: string
                uri:
                  type: string
```

(a) YAML response schema

```
ppt getAlbumTracks&200&items():::EXIT2
ppt-type subexit
variable input
... // variables from the entry point are repeated here
variable return
  var-kind return
  dec-type getAlbumTracks&Output&200&items
  rep-type hashcode
variable return.artists
...
variable return.artists[..]
...
variable return.available_markets
  var-kind field available_markets
  enclosing-var return
  dec-type java.lang.String[]
  rep-type hashcode
variable return.available_markets[..]
  var-kind array
  enclosing-var return.available_markets
  array 1
  dec-type java.lang.String[]
  rep-type java.lang.String[]
variable return.id
...
variable return.name
...
variable return.explicit
...
variable return.duration_ms
...
variable return.linked_from
  var-kind field linked_from
  enclosing-var return
  dec-type getAlbumTracks&Output&200&items&linked_from
  rep-type hashcode
variable return.linked_from.id
  var-kind field id
  enclosing-var return.linked_from
  dec-type java.lang.String
  rep-type java.lang.String
variable return.linked_from.uri
...
```

(b) Declaration file

Listing 9. Second EXIT nesting level.

invariants by comparing variables that are potentially unrelated, leading to false positives and irrelevant results. By using nesting, variables are assigned scopes based on their nesting level, preventing unnecessary comparisons and improving efficiency.

3.1.2 Data Traces. Data trace files contain the actual input and output values observed during the execution of the API. Each data trace record must have the same variables as 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`, along with the API request being represented. Analogously, Listing 11 depicts, along with the fragment of the response in JSON format being represented, 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), omitted for brevity.

Beet is 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 [1].

3.2 Invariant Definition

This section describes the changes made to Daikon for enabling the detection of likely invariants in REST APIs. In order to identify classes of invariants that could be used as effective test oracles, we used a benchmark of 40 APIs

```
GET https://api.spotify.com/albums/4Em5W5HgYEvhpc/tracks?limit=1&market=ES
```

(a) API request

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

(b) Data trace file

Listing 10. ENTER data trace file.

```
{
   "total": 14,
   "href": "https://api.spotify.com/albums/4Em...",
   "items": [
      {
         "artists": [
            {
               "id": "2CvCyf1gEVhI0mX6aFXmVI",
               "name": "Paul Simon"
            },
            {
               "id": "70cRZdQywnSFp9pnc2WTCE",
               "name": "Arthur Garfunkel"
            }
         ],
         "available_markets": [ "ES", "US", "JP" ],
         "id": "0gFvkiT2afIcJwNxQ7W51",
...
```

(a) API response in JSON format

```
getAlbumTracks&200():::EXIT1
input
...
return
2043815652
1
return.total
14
1
return.href
"https://api.spotify.com/albums/4Em..."
1
return.items
1534143414
1
return.items[..]
[313805079]
1
```

(b) Data trace file

Listing 11. One instance of an EXIT program point in a data trace file.

(702 operations) from the RapidAPI repository [14] that some of the authors collected systematically for a recent publication [30]. None of the APIs of this benchmark were used for the evaluation of the approach, in order to avoid bias. This dataset contains real-world APIs which spam multiple application domains, such as finance, sports, travel, and visual recognition, among many others.

We followed a systematic procedure to identify the types of invariants present in REST APIs. For each API operation, we began by reviewing its name and description, as well as analyzing its input parameters, to understand its intended behavior and usage. Next, we examined each response field to identify invariant templates to add (i.e., define a new template), enable or disable, as follows:

(1) **Unary invariants.** These involve a single variable, such as the "scalar.LowerBound" invariant, which specifies the minimum allowable value for a numerical value. For each response field, we assessed whether existing Daikon invariants adequately described its characteristics and were sufficiently generic for application across different APIs. When gaps were identified, new invariants were defined. This resulted in the addition of 22 new invariants and disabling 6 invariants.

(2) **Binary invariants.** These involve two variables, such as "twoString.StringEqual", which enforces that two string variables always have the same value. We analyzed potential relationships between the response field under analysis, input parameters and the remaining response fields. This process led to enabling 9 previously disabled

invariants and disabling 27 invariants that were prone to false positives or combinatorial explosions of reported invariants.

(3) **Ternary invariants.** These involve three variables, such as identifying linear relationships among numerical variables. A similar analysis as the one conducted for binary invariants was performed for ternary relationships, but no meaningful logical relationships involving three variables were observed in the APIs. Consequently, existing Daikon ternary invariants were disabled as they were not observed in any of the APIs.

Overall, we implemented 22 new types of invariants, disabled 35 default Daikon invariants, and activated 9 invariants disabled by default in Daikon. Table 2 lists all the added, disabled, and enabled invariants in our modified version of Daikon. In the context of naming invariants, the term "sequence" refers to array variables, while "scalar" refers to numerical variables, booleans (0 or 1) and objects (represented as hashcodes). The new invariants detect string patterns such as URLs, dates, and length constraints. The disabled invariants would most likely (according to our analysis of the input and output format of the benchmark operations) provide irrelevant or misleading information in our context, 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 when comparing array variables (e.g., `x[] is a subsequence of y[]`) and detecting substring relations between string variables (e.g., `input.id is a substring of return.href`). Overall, our customized version of Daikon supports 106 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 (4 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 [1] for a more detailed description of each type of invariant, including examples. 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.

3.3 Minimizing false positives

In the previous version of our approach [31], invariants of the arithmetic comparison category caused of 3 out of every 4 false positives. Most false positives were invariants comparing two unrelated output properties. For example, the invariant `return.duration_ms > size(return.artists[])` states that the duration of a song in milliseconds should always be greater than the number of artists in the song. While this may be true in most cases, it does not characterize the API specification and we consider it a false positive. Finding counterexamples to automatically rule out these false positives is extremely unlikely, and the number of reported false positives can become unbearable if the API has multiple numerical response fields.

To address this, AGORA+ incorporates a novel heuristic to reduce false positives caused by arithmetic comparisons, leveraging the following observation: parameters with different value ranges are unlikely to be related. For example,

```
=== 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
return.duration_ms > size(return.artists[])
return.duration_ms > size(return.available_markets[])
=== getAlbumTracks&200&items&artists():::ENTER
...
=== getAlbumTracks&200&items&artists():::EXIT
LENGTH(return.id)==22
```

Listing 12. Invariants detected in the "getAlbumTracks" operation of the Spotify API.

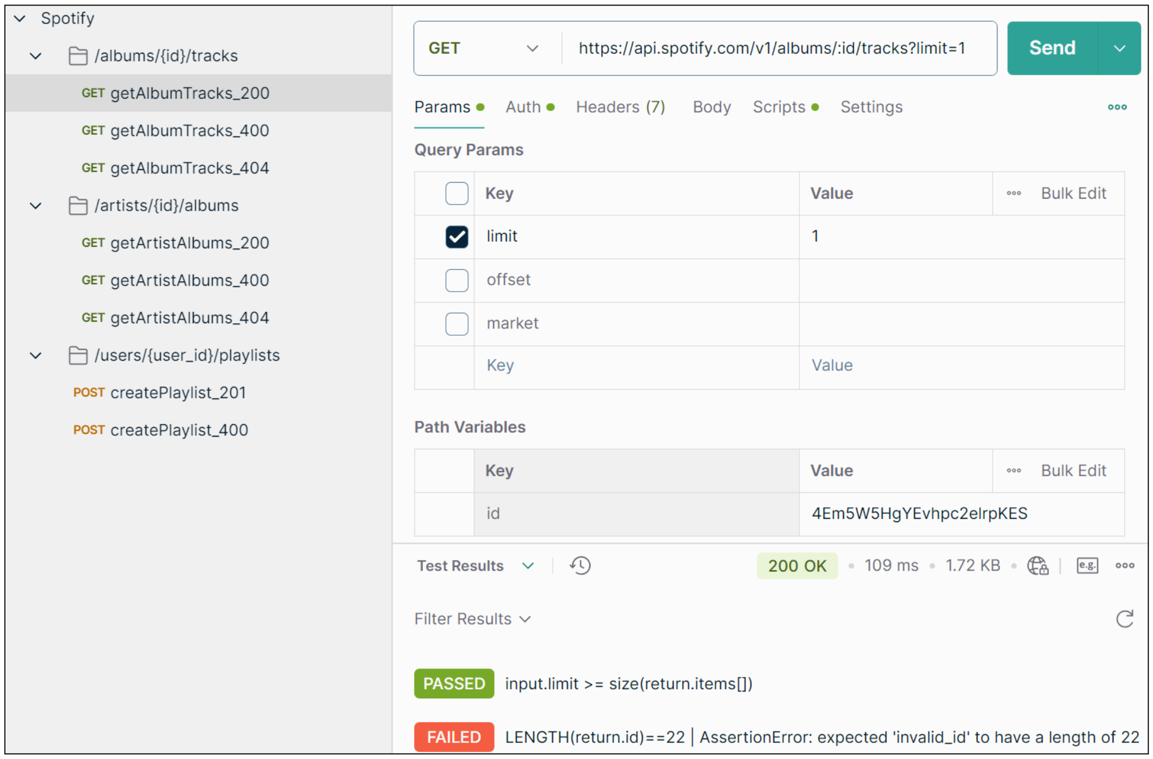
Table 2. Summary of modifications performed on Daikon. The strike-through invariants had to be disabled during our evaluation to avoid a combinatorial explosion of reported false positives.

Added invariants (22)	**Unary (22)**	string.IsUrl	stringsequence.ElementsAreUrl
		string.FixedLengthString	stringsequence.FixedLengthString
		string.IsNumeric	stringsequence.ElementsAreNumeric
		string.IsEmail	stringsequence.ElementsAreEmail
		string.IsDateYYYYMMDD	stringsequence.ElementsAreDateYYYYMMDD
		string.IsDateDDMMYYYY	stringsequence.ElementsAreDateDDMMYYYY
		string.IsDateMMDDYYYY	stringsequence.ElementsAreDateMMDDYYYY
		string.IsTime	stringsequence.ElementsAreTime
		string.IsTimeWithSeconds	stringsequence.ElementsAreTimeWithSeconds
		string.IsTimeAMPM	stringsequence.ElementsAreTimeAMPM
		string.IsTimestamp	stringsequence.ElementsAreTimestamp
Disabled invariants (35)	**Unary (6)**	scalar.NonZero	sequence.EltNonZero
		scalar.NonZeroFloat	sequence.EltNonZeroFloat
		scalar.RangeInt.PowerOfTwo	sequence.EltRangeInt.PowerOfTwo
	Binary (27)	twoScalar.IntNonEqual	~~twoSequence.PairwiseStringGreaterThan~~
		twoScalar.FloatNonEqual	~~twoSequence.PairwiseStringLessEqual~~
		twoScalar.LinearBinary	~~twoSequence.PairwiseStringGreaterEqual~~
		twoScalar.LinearBinaryFloat	twoScalar.NumericInt.Divides
		~~twoString.StringNonEqual~~	twoScalar.NumericInt.Square
		~~twoString.StringLessThan~~	twoScalar.NumericFloat.Divides
		~~twoString.StringGreaterThan~~	twoScalar.NumericFloat.Square
		~~twoString.StringLessEqual~~	twoSequence.PairwiseNumericInt.Divides
		~~twoString.StringGreaterEqual~~	twoSequence.PairwiseNumericInt.Square
		~~twoSequence.SeqSeqStringLessThan~~	twoSequence.PairwiseNumericFloat.Divides
		~~twoSequence.SeqSeqStringGreaterThan~~	twoSequence.PairwiseNumericFloat.Square
		~~twoSequence.SeqSeqStringLessEqual~~	twoSequence.PairwiseLinearBinary
		~~twoSequence.SeqSeqStringGreaterEqual~~	twoSequence.PairwiseLinearBinaryFloat
		~~twoSequence.PairwiseStringLessThan~~	
	Ternary (2)	threeScalar.LinearTernary	LinearTernaryFloat
Enabled invariants (9)	**Binary (9)**	twoString.StdString.SubString	twoSequence.SuperSequence
		twoSequence.SubSequence	twoSequence.SuperSequenceFloat
		twoSequence.SubSequenceFloat	twoSequence.SuperSet
		twoSequence.SubSet	twoSequence.SuperSetFloat
		twoSequence.SubSetFloat	

the duration of a song typically spans thousands of milliseconds (e.g., more than 30K ms), whereas the number of artists on a song may vary from 1 to around 10. Our heuristic evaluates the observed value ranges for variables involved in arithmetic inequalities (e.g., `x<y`, `x<=y`, `x>y`, `x>=y`). Specifically, it discards invariants if the maximum observed value of one variable (e.g., 10 artists) is less than the minimum observed value of the other variable (e.g., 30K ms). For example, the invariant `return.duration_ms > size(return.artists[])` would be discarded if the respective value ranges do not overlap, such as duration of 30K-300K ms versus 1–10 artists. When value ranges overlap, the heuristic allows the invariant to be retained, as the relationship might be meaningful. To implement this heuristic, we modified 32 arithmetic comparison invariants in Daikon, ensuring they are not reported if there is no overlap between the value ranges of the two variables being compared. This heuristic enhances the relevance of the invariants generated by Daikon and has potential for application in the detection of invariants in other domains.

3.4 PostmanAssertify: Automated generation of test assertions

This section presents PostmanAssertify, a tool that converts the invariants reported by AGORA+ into executable assertions in JavaScript that are compatible with Postman. Postman [12] is a widely used API development and testing platform that offers a graphical interface to create, send, and analyze HTTP requests to REST APIs. One of its standout features is the use of collections, which allow developers to group and organize API requests. This enables developers to run entire suites of tests with a single click.



- Spotify
  - /albums/{id}/tracks
    - GET getAlbumTracks_200
    - GET getAlbumTracks_400
    - GET getAlbumTracks_404
  - /artists/{id}/albums
    - GET getArtistAlbums_200
    - GET getArtistAlbums_400
    - GET getArtistAlbums_404
  - /users/{user_id}/playlists
    - POST createPlaylist_201
    - POST createPlaylist_400

GET https://api.spotify.com/v1/albums/:id/tracks?limit=1 Send

Params Auth Headers (7) Body Scripts Settings

Query Params

	Key	Value
✓	limit	1
	offset	
	market	
	Key	Value

Path Variables

Key	Value
id	4Em5W5HgYEvhpc2elrpKES

Test Results 200 OK • 109 ms • 1.72 KB

Filter Results

PASSED input.limit >= size(return.items[])

FAILED LENGTH(return.id)==22 | AssertionError: expected 'invalid_id' to have a length of 22

Fig. 2. Postman collection generated by PostmanAssertify.

```
// Getting value of the id path parameter
input_id = pm.request.url.path[2];
if (input_id != null) {
   input_id = decodeURIComponent(input_id);
}

// Getting value of the market query parameter
input_market = pm.request.url.query.get("market");
if (input_market != null) {
   input_market = decodeURIComponent(input_market);
}

// Getting value of the limit query parameter
input_limit = pm.request.url.query.get("limit");
if (input_limit != null) {
   input_limit = decodeURIComponent(input_limit);
   input_limit = parseInt(input_limit);
}

valuesToConsiderAsNull = [];

// Base nesting level (getAlbumTracks&200 program point)
response = pm.response.json();

pm.test("input.limit >= size(return.items[])", () => {
   // Getting value of variable: return_items_size_array
   return_items_size_array = response["items"];
   if (return_items_size_array != null) {
      return_items_size_array = return_items_size_array.length;
   }

   if ((input_limit != null) && (!valuesToConsiderAsNull.includes(input_limit)) &&
   (return_items_size_array != null) && (!valuesToConsiderAsNull.includes(return_items_size_array))) {
      pm.expect(input_limit).to.be.at.least(return_items_size_array);
   }
})

// Second nesting level (getAlbumTracks&200&items program point)
response_items = response["items"]
if (response_items != null) {
   for (response_items_index in response_items) {
      response_items_element = response_items[response_items_index];

      pm.test("LENGTH(return.id)==22", () => {
         // Getting value of variable: return_id
         return_id = response_items_element["id"];

         if ((return_id != null) && (!valuesToConsiderAsNull.includes(return_id))) {
            pm.expect(return_id).to.have.length(22);
         }
      })

   } // Closing for response_items
} // Closing if response_items
```

Listing 13. Part of the Postman test script generated by PostmanAssertify for the Spotify "getAlbumTracks" operation.

PostmanAssertify receives the API specification and the CSV file containing the set of confirmed invariants generated by AGORA+ as inputs. PostmanAssertify returns a Postman collection, including a sample API request for each status code of the API operations under test and—attached to each request—all the invariants as executable test assertions.

Figure 2 shows an example of a Postman collection generated by PostmanAssertify running on the Postman desktop application. The collection contains a request for each response code of every API operation where AGORA+ has detected invariants. These operations are listed in the directories on the left sidebar of Figure 2. Each request, as shown in the right pane of Figure 2, contains the following:

- The URI of the API operation with the input parameters configured.
- A test script written in JavaScript using the Chai library [2, 26].
- Assertions for all the confirmed invariants implemented in the test script.

Every time this script is executed, Postman generates a report such as the one shown in the bottom pane of Figure 2, in which it indicates whether each assertion passed or failed, with an explanatory message in case of failure. The collection reports can be exported in JSON format for further analysis.

Listing 13 contains a snippet of the test script generated by PostmanAssertify for the Spotify "getAlbumTracks" operation used as running example. For the sake of brevity, the test script only contains the assertions generated for two of the invariants: `input.limit >= size(return.items[])`, reported at the base nesting level; and `LENGTH(return.id)==22`, reported at the `items` nesting level.

PostmanAssertify first generates the code that initializes the variables storing the values of the API request input parameters (lines 1–18), the special values to consider as null (line 20), and the returned API response (line 23). Then, it generates the code containing all the test cases (one per confirmed invariant) of the base nesting level (lines 25–36). Each test case starts by obtaining the values of the variables involved in the assertions (lines 26–30), which can be either input parameters of properties of the response. Then, if none of the variables is null (lines 32–33), the assertion is executed (line 34).

After creating the tests of the base nesting level, PostmanAssertify generates the code for accessing the base variable of the subsequent nesting level (lines 38–42), and all the test cases of this new nesting level, if any (lines 44–51). This process is iteratively repeated for all the subsequent nesting levels.

To implement the Postman assertions (lines 34 and 49), we extended Daikon with a new output format [5] so it reports the invariants supported by AGORA+ as assertions of the Chai JS library. PostmanAssertify has been implemented in Java and is publicly available on GitHub [13].

4 EVALUATION

We aim to answer the following research questions:

RQ1: *How effective is AGORA+ in generating test oracles?* We measure the precision of AGORA+ in generating invariants that properly model the expected API behavior.

RQ2: *How does the size of the input dataset affect the performance of AGORA+?* The precision and the number of detected invariants usually depends on the quality and diversity of the input datasets (i.e., API requests and responses).

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

4.1 Experimental Data

For our experiments, we used a set of 25 operations from 20 industrial APIs (Table 3) tested by previous authors [30, 83, 84, 99]. All these APIs are freely accessible. Our replication package [15] provides detailed information on the specific authentication requirements for each API. 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 (Foursquare, iTunes, Ohsome, OMDb, RESTCountries, and Yelp) or used the version available in APIs.guru [16] (Spotify, Stripe, Vimeo, and YouTube), modifying them according to the latest version of the web docs. Some of the existing 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. This process was carried out using a semi-automated process with RESTest, which leverages the Atlassian OAS syntactic validator [23]. During test case generation, RESTest identified missing response fields, which were then manually added to the OAS in an iterative process that took only a few minutes per API operation.

The Vimeo API offers various authentication scopes [24], including public (exclusively returns public video data), and private (grants access to private video data, provided the user is the video owner). We opted for the public scope due to our limited access to private video metadata. When using the public scope, the API does not return certain response fields that are exclusive to the private scope, and therefore AGORA+ reports that these fields are always null. We removed these fields from the OAS specification to avoid bloating the results.

For each operation, the RESTest [82] framework generated and executed API calls until obtaining 10K valid API calls per operation, at least 9K of which must be non-empty results (i.e., the result is not a zero-length list), generating 250K calls in total. According to REST best practices [96], 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.

4.1.1 Input data generation. The invariants detected by AGORA+ largely depend on the diversity of the API requests provided as input. To foster such diversity, we configured RESTest to use specific data generation strategies for each of the input parameters of the API operations of our study. For each one of the 224 different input parameters of the operations of our analysis, we used one of the following strategies:

- *Manually created data dictionaries (31.3% of the parameters).* Collection of input values created manually. For example, a list of database IDs (such as Spotify Album IDs), country codes or city names, among others.
- *Enum values (19.2% of the parameters).* List of all the possible values accepted for a specific input parameter (usually specified in the documentation). For example, when creating a GitHub repository, the value of the `visibility` parameter can only be either "public" or "private".
- *Numbers (19.2% of the parameters).* Randomly generates a number within minimum and maximum bounds. This strategy is used, for instance, in pagination parameters.
- *English words (17% of the parameters).* Randomly generates a word or sequence of words using the extJWNL (Extended Java WordNet Library) Java library [7]. Used for free text fields, such as when setting the name of a Spotify playlist.
- *Booleans (10.3% of the parameters).* Randomly generates either "true" or "false".
- *Dates (3% of the parameters).* Randomly generates a string following a specific date format. The generated date is within a start and end date.

When manually creating the data dictionaries, we followed current practices and selected values based on an analysis of the API specification and documentation. For instance, for the `location` parameter of the Yelp API, we created a list of cities located in different continents. Our replication package [15] contains the RESTest configuration files and the data dictionaries used for generating the test cases.

4.2 Experiment 1: Test Oracle Generation

This experiment answers RQ1 (precision of the approach) and RQ2 (impact of the dataset size) by evaluating the effectiveness of AGORA+ in generating test oracles.

4.2.1 Experimental Setup. For each API operation, we randomly divided the 10K automatically-generated requests into 10 random subsets of each of the sizes 50, 100, 500, and 1K requests. We report the performance in terms of average precision (i.e., $precision = TP/(TP + FP)$) and average number of reported invariants across the 10 runs. For each run, the inferred invariants were classified as true positives or false positives based on manual analysis. The cost of manually verifying invariants was minimal for most cases, as they could be confirmed within seconds from the name and the description included in the API specification (e.g., a field named `image_url` is clearly expected to be a URL) or by analyzing the API documentation, which typically specifies the expected format or constraints of response fields in natural language. If the documentation was not clear enough, we consulted the API providers about the expected API behavior. For instance, in the RESTCountries API, we noticed that one of the *idds* (international dialing code) of Canada was an empty string (“”), instead of a numerical code. Upon seeking clarification, the API providers confirmed that this behavior was intentional [17]. A true positive invariant describes a property of the output that should always hold; they 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, AGORA+ may report that the visibility of a GitHub repository is always “public” (`return.visibility == “public”`), because it never observes the “private” value, or that the number of stars in a repository is always greater than its number of forks (`return.stargazers_count > return.forks_counts`). Although these were true for all the test cases used to infer the invariants, they do not represent the intended behaviour of the API, and therefore they are false positives. As part of our analysis, we measured the proportion of reported true positives and false positives of each one of the invariant categories presented in Section 3.2.

Daikon also detects invariants among input parameters (i.e., likely preconditions in ENTER program points). These invariants can offer insights about the API behavior regarding input parameter values. For example, they may report that all the supported values of an input parameter have been used in at least one request that returned a valid response (e.g., `input.direction one of { “asc”, “desc” }`) or that the numerical values are within a valid range (e.g., `input.limit >= 1`). A violation of these invariants or an incomplete version of them could reveal either a limitation in the test suite used as input for AGORA+ or a bug in the API implementation. For example, the invariant `input.direction == “asc”` (an incomplete version of `input.direction one of { “asc”, “desc” }`) reveals that there are no API responses for requests using the “desc” value for the `direction` parameter. This could mean either that the test suite is not diverse enough or that there are no valid responses when this input parameter value is used due to a bug in the API implementation. However, invariants about inputs do not provide information about the output and hence we did not use them to calculate precision, focusing only on likely postconditions.

We compared the effectiveness of AGORA+ against the default version of Daikon and the previous version of our approach, AGORA [31]. AGORA supports the same invariants as AGORA+, but it does not apply the heuristic presented in Section 3.3 for reducing false positives. In all cases, our novel front-end, Beet, transformed 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. For a fair comparison, we disabled these problematic invariants in the default version of Daikon used as baseline, as detailed in our supplementary material [15]. These disabled invariants are strike-through in Table 2.

We compared the performance of these three approaches in terms of precision. We do not compare them in terms of recall due to the large number of response fields returned by many of the API operations (up to several hundreds). Generating a ground truth for all the supported invariants would require manually annotating a dataset of up to tens of thousands of invariants per API operation.

The experiment was performed on a laptop equipped with Intel i7-11800H @2.30GHz, 32GB RAM, and 1TB SSD running Windows 11 and Java 17.

4.2.2 *Experimental Results.*

Table 3. 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		AGORA+		Daikon		AGORA		AGORA+		Daikon		AGORA		AGORA+		Daikon		AGORA		AGORA+		Daikon		AGORA		AGORA+	
	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)	I	P (%)
AmadeusCitySearch-GETCities	34	84	34	97	32	100	38	84	37	94	33	100	41	80	38	93	33	99	41	80	38	93	34	98	39	85	37	95	33	100
AmadeusHotel-getMultiHotelOffers	132	17	121	54	120	55	131	18	121	56	120	57	118	21	115	62	114	62	113	23	110	66	110	66	99	26	107	68	107	68
Deutschebahn-getStations	202	22	127	30	91	41	164	28	114	36	86	48	110	45	103	45	82	57	102	49	102	46	81	58	94	53	97	48	76	62
DHL-getLocation-finderV1Find-by-address	54	40	26	50	19	64	52	41	26	52	18	67	49	44	25	55	18	69	48	46	24	58	18	71	45	49	22	64	17	76
FDIC-searchInstitutions	450	14	407	32	235	48	430	16	408	33	226	52	487	17	466	32	231	57	503	17	486	32	229	60	589	16	601	29	249	62
Foursquare-placeSearch	267	13	145	32	85	43	308	13	164	32	95	43	339	13	188	30	94	50	332	14	190	31	94	50	270	17	154	35	90	48
GitHub-createOrganizationRepository	83	94	200	97	194	99	81	96	199	98	193	100	81	96	199	98	193	100	80	96	198	98	192	100	80	96	198	98	192	100
GitHub-getOrganizationRepositories	44	40	149	86	139	92	42	43	146	88	138	93	39	46	147	89	140	94	39	46	149	89	142	92	38	47	148	89	142	93
GitLab-getApiV4ProjectsIdBadges	5	23	6	57	4	92	5	28	6	58	4	85	5	43	6	67	4	100	4	47	6	67	4	100	4	50	6	67	4	100
GitLab-listProjectJobs	142	12	152	32	91	54	136	13	142	36	80	63	119	15	135	38	71	72	116	16	134	39	70	74	113	17	130	41	67	79
iTunes-search	120	12	98	30	54	54	127	12	101	32	57	56	121	14	93	36	54	62	118	14	88	38	50	66	111	17	78	44	44	77
LanguageTool-checkText	123	54	65	67	57	74	116	61	66	71	59	78	120	68	69	75	63	80	120	72	71	76	65	82	120	74	73	78	67	84
Marvel-getComicIndividual	223	24	121	46	103	52	199	28	114	51	96	58	174	34	108	57	92	65	160	38	104	60	89	68	140	46	96	66	88	69
NYTimesBooks-GET_lists-format	44	62	44	68	43	70	42	70	40	77	40	78	35	86	34	89	34	89	34	87	34	90	34	90	34	88	32	91	32	91
Ohsome-elementsAggregation	11	81	18	88	15	99	11	82	18	89	15	100	11	82	17	88	15	100	11	82	17	88	15	100	11	82	17	88	15	100
OMDB-byIdOrTitle	7	55	17	87	16	92	7	56	16	91	16	97	7	54	16	92	15	98	7	54	16	92	15	98	7	57	16	94	15	100
OMDB-bySearch	5	90	5	95	5	97	6	65	6	78	6	92	5	75	6	82	5	98	5	79	6	83	5	100	6	83	7	86	6	100
RESTCountries-v31ListOfCodes	139	27	68	40	60	45	139	31	56	50	52	53	126	39	50	61	47	65	125	39	49	63	46	67	124	40	49	63	46	67
Spotify-createPlaylist	22	100	41	100	41	100	22	100	41	100	41	100	22	100	41	100	41	100	22	100	41	100	41	100	22	100	41	100	41	100
Spotify-getAlbumTracks	48	43	65	83	59	92	47	45	67	86	61	94	43	49	66	88	60	97	42	50	66	88	60	98	41	54	66	89	60	98
Spotify-getArtistAlbums	55	40	53	82	50	89	50	47	53	86	49	92	38	64	51	90	48	96	32	75	50	92	47	98	31	84	52	92	49	98
Stripe-PostProducts	58	67	41	83	31	99	58	67	42	83	31	100	58	67	42	83	32	98	58	67	42	83	31	99	64	61	40	88	31	100
Vimeo-search_videos	548	39	344	55	287	66	546	41	349	56	295	66	524	46	349	59	299	69	508	50	343	61	295	71	514	53	330	66	286	76
Yelp-getBusinesses	68	25	33	36	23	52	59	29	32	37	21	56	49	35	25	48	14	85	45	37	24	51	13	91	41	39	22	50	12	92
YouTube-listVideos	218	33	181	62	122	73	224	34	192	61	118	78	230	36	198	64	121	85	230	36	200	64	121	86	201	42	196	66	123	86
TOTAL	**3100**	**31**	**2561**	**55**	**1975**	**68**	**3042**	**33**	**2557**	**57**	**1951**	**71**	**2953**	**36**	**2586**	**59**	**1921**	**76**	**2897**	**38**	**2587**	**60**	**1903**	**78**	**2838**	**40**	**2615**	**60**	**1892**	**80**

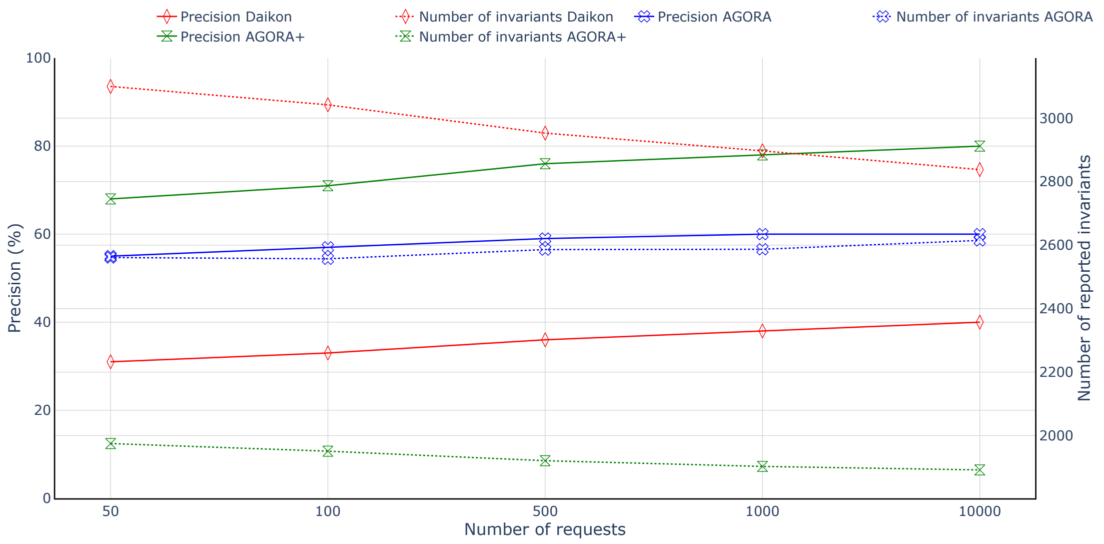


Fig. 3. Precision and number of invariants reported, depending on the size of the input dataset.

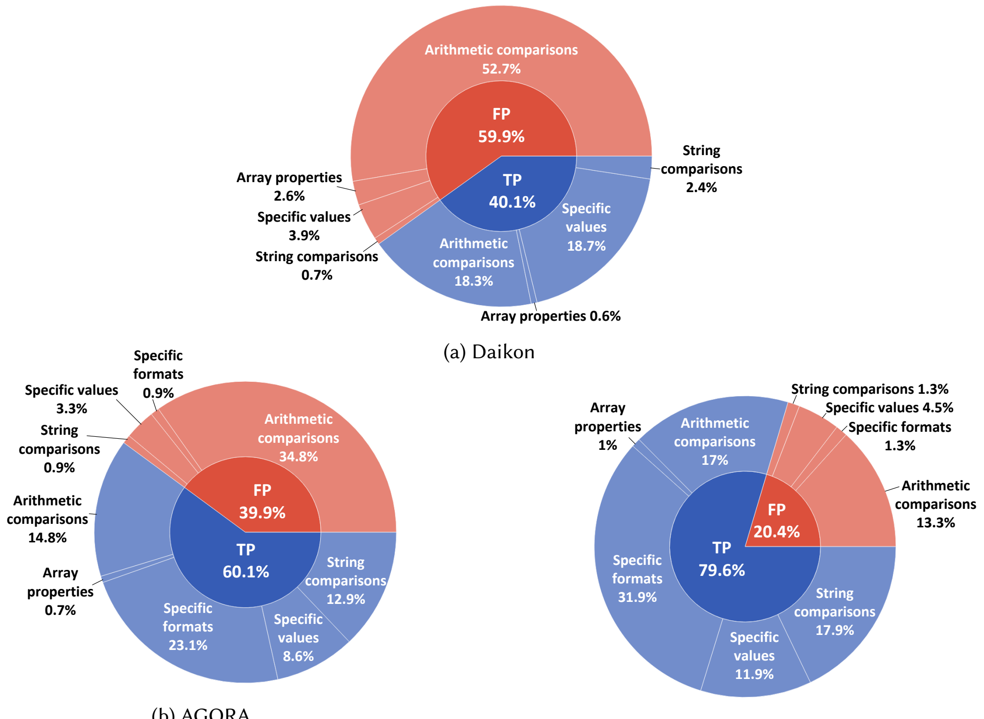
RQ1: Effectiveness of the approach. Table 3 shows the results for each API operation, set of API requests size (50, 100, 500, 1K, 10K), and approach (AGORA+ vs. AGORA vs. default Daikon). The reported values for the random subsets of 50, 100, 500, and 1K requests represent the average precision and the average number of reported invariants across 10 runs. Figure 3 shows the evolution of the precision and the number of invariants reported by each approach when the input test suite size increases.

Next, we analyze the results obtained from the entire dataset of 10K requests (RQ1), shown in the last two columns of the table. The following section compares different sizes of the input dataset (RQ2).

When learning from the whole dataset (10K API requests), AGORA+ obtained a total precision of 80% (1506 out of 1892 invariants are valid test oracles), outperforming the precision of both AGORA (60%) and the default configuration of Daikon (40%) by 33% and 100%, respectively. AGORA+ equaled or outperformed all the baselines in all the API operations.

The precision achieved by AGORA+ ranged between 48% in the Foursquare API and 100% for 8 API operations (AmadeusCitySearch-GETCities, GitHub-createOrganizationRepository, GitLab-getApiV4ProjectsIdBadges, OMDb-byIdOrTitle, OMDb-bySearch, Ohsome, Spotify-createPlaylist, and Stripe). 24 of the 29 false positives in Deutschebahn are related to station opening and closing times; 12 of these FPs state that the opening (or closing) times of all the workdays of the week are the same (e.g., `monday.from_time == tuesday.from_time`). This fact is true for for all the stations returned by the API, but we consider it possible that it would not be true for every station, so we conservatively marked it as a false positive. 12 more FPs are about the length of the station closing time: it is always 5 characters long, as in “18:00”. Because it is conceivable that a station closes before 10:00am, we again marked these as false positives.

The heuristic applied by AGORA+ to reduce the number of false positives (detailed in Section 3.3) led to a precision increase over AGORA of up to 114% (FDIC API), yielding a precision increase in 22 out of 25 operations. Most false positives in the FDIC API arise from arithmetic comparisons involving response fields that serve as numerical flags (values of 0 or 1) with other numerical variables. The heuristics of AGORA+ significantly reduced the number of false



(a) Daikon

(b) AGORA

(c) AGORA+

Fig. 4. Reported invariants by category in Daikon, AGORA, and AGORA+.

positives for this operation, lowering them from 415 to 81. In the Yelp API, 10 out of the 11 false positives reported by AGORA are arithmetic comparisons stating that the value of the numeric parameters `open_at` (an integer representing an Unix time) and `radius` (search radius in meters) is always greater than all the numeric response fields, including the size of all the arrays in the response. The heuristic applied by AGORA+ automatically suppressed all these false positives.

Statistical analysis confirms significant performance differences among the approaches. The Friedman test (*p-value* < 0.05) rejected the null hypothesis of equivalent precision among AGORA+, AGORA, and Daikon. Post-hoc analysis with Holm correction further validated these findings.

The number of invariants reported per operation varied between 4 in the GitLab-getApiV4ProjectsIdBadges operation and 286 in the Vimeo API. We observed a strong positive correlation between the number of response fields and the number of reported invariants, with a Spearman correlation coefficient of 0.92.

Figure 4 breaks down the true positives and false positives of each approach (Daikon, AGORA and AGORA+). In the default version of Daikon, the largest portion of false positives (88%) were found in the arithmetic comparisons category, followed by specific values (7%), array properties (4%), and string comparisons (1%). The invariants reported by AGORA and AGORA+ are the same, except those belonging to the arithmetic comparisons category. Thanks to the heuristics applied in AGORA+ (c.f. Section 3.3), the percentage of false positives in this category decreased from 87% (909 invariants) to 65% (252 invariants), suppressing a total of 657 false positives.

False positives in the arithmetic comparison category typically occur when comparing numerical fields with values in different magnitude orders, such as the duration of a Spotify song in milliseconds and its number of artists. The remaining false positives are either invariants that report an object as always null or that limit a response field to only a specific value or set of values, but the API supports more (e.g., reporting that the visibility of a GitHub repository is always "public"). These invariants indicate either a lack of diversity in the test suite or bugs in the API (c.f. Section 5).

The heuristic applied by AGORA+ to reduce false positives also suppressed 66 true positives. This happened in cases in which the variables involved in the invariant had clearly defined minimum and maximum values that did not overlap (i.e., they guarantee that the invariant always holds and thus it is not a false positive). For example, the automatically suppressed invariant `input.maxWidth > size(return.items[])`, reported in the YouTube API is true because, according to the documentation [27], the minimum value of `input.maxWidth` is 72, whereas the maximum value of `size(return.items[])` is 50.

Overall, arithmetic comparisons remain the most common type of false positives reported by AGORA+ (65% of false positives), followed by specific values (22%), string comparisons (6%) and specific formats (6%), with no false positives of the array properties category.

We now present an analysis of the performance of the sets of added, enabled, and disabled invariants, specified in Table 2. AGORA+ reported 627 invariants belonging to the set of added invariants, spanning 13 invariant types and achieving precision ranges between 91%, and 100% for 11 invariant types. Among enabled invariants, it detected 272 instances of a single type ("twoString.StdString.SubString") with 99% precision. For disabled invariants, the original configuration of Daikon reported 789 instances across 11 invariant types, with precision ranging from 0% (for 7 types) to 3%, except for two high-performing types: "scalar.NonZero" (240 instances, 98% precision) and "sequence.EltNonZero" (74 instances, 99% precision). These invariants specify whether numerical variables or array elements are nonzero but are also used to indicate non-null objects. These invariants were intentionally disabled in AGORA+ to prevent redundancy or bloating the results, as nullability can be effectively specified using the OpenAPI Specification (OAS) nullable property.

Beet took between 1.4 seconds (50 requests) and 67 seconds (10K requests) to generate the instrumentation of the target API operations. Daikon (customized and default version) took between 1.8 seconds (50 requests) and 30 seconds (10K requests) to detect the reported invariants. Overall, AGORA+ took less than 2 minutes to generate the invariants using the complete dataset of 10K requests.

Answer to RQ1: How effective is AGORA+ in generating test oracles?

AGORA+ is effective in generating test oracles, achieving a precision of 80% when learning from 10K API requests. This is a precision improvement of 100% over the default configuration of Daikon.

RQ2: Impact of the size of the input dataset. Figure 5 shows the evolution of the precision of AGORA+ with respect to the number of API requests, as detailed in Table 3. The total precision improved from 68% with 50 API requests (an average of 1345 valid invariants, i.e., test oracles), to 80% with the complete dataset (1506 valid invariants). This means a drop in precision of just 12 percentage points when using the smallest dataset (50 API requests). The greatest precision increase takes place between 100 and 500 requests (5 percentage points). Conversely, the number of reported invariants decreases, mostly due to the suppression of false positives. The suppression of invariants classified as true positives when increasing the size of the input test suite reveals that a counterexample (i.e., failure) has been found (c.f. Section 5).

For instance, in the NYTimesBooks API, the suppression of the confirmed invariant `LENGTH(return.isbn10)==10` revealed the presence of ISBNs with invalid values in the test suite. Figure 6 shows the evolution of the number of reported valid invariants (i.e., true positives) with respect to the number of API requests. When using the 50 requests datasets, AGORA+ detected, on average, 91% of all the true positives reported by the 10K dataset, with a standard deviation of 8 percentage points. The increase in the number of reported true positives ranged between 0 percentage points in 8 API operations (GitHub-createOrganizationRepository, GitLab-getApiV4ProjectsIdBadges, NYTimesBooks-GET_lists-format, Ohsome-elementsAggregation, OMDb-byIdOrTitle, Spotify-createPlaylist, Stripe-PostProducts and Yelp-getBusinesses), and 38 percentage points in the "searchInstitutions" operation of the FDIC API. This shows the applicability of AGORA+ even when provided with a small set of test cases as input.

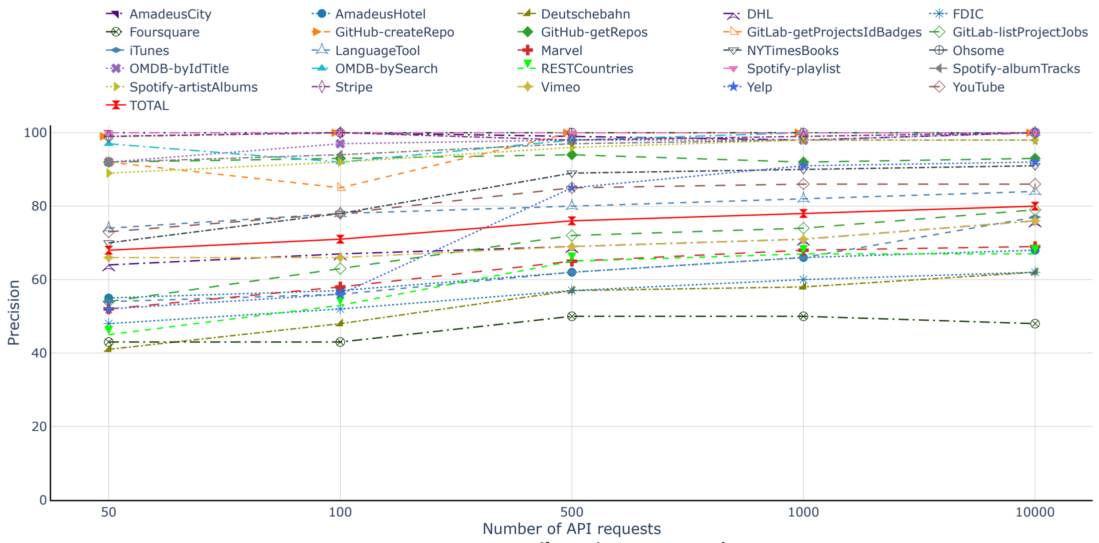


Fig. 5. How test suite size affects the precision of AGORA+.

Figure 7 shows a boxplot with the distribution of the precision achieved by AGORA+ on all the operations across multiple divisions on random subgroups of 50 test cases. For all the API operations, the difference between the median and the first and third quartiles was less than 5 percentage points, with the exceptions of the NYTimesBook API, with a difference of 6 percentage points between the median and the third quartile, and the "getApiV4ProjectsIdBadges" operation of the GitLab API, with a difference of 17 percentage points between the median the first quartile. This significant difference in the GitLab API was due to the suppression, in 8 out of 10 seeds, of 2 valid invariants (out of an average of 4 reported invariants) which revealed the presence of real bugs (Section 5). Overall, these results show that the effectiveness of AGORA+ is stable across different datasets.

Answer to RQ2: How does the size of the input dataset affect the performance of AGORA+?

Increasing the size of the test suite causes an increase in the precision achieved and a decrease in the number of invariants reported. The suppressed invariants are mostly false positives, with only a small increase in the number of valid invariants (i.e., true positives). With a small but diverse set of 50 API requests, AGORA+ achieves a precision of 68% (12 percentage points less than when using a dataset 200 times bigger), and detects 91% of the valid invariants detected when using the 10K dataset.

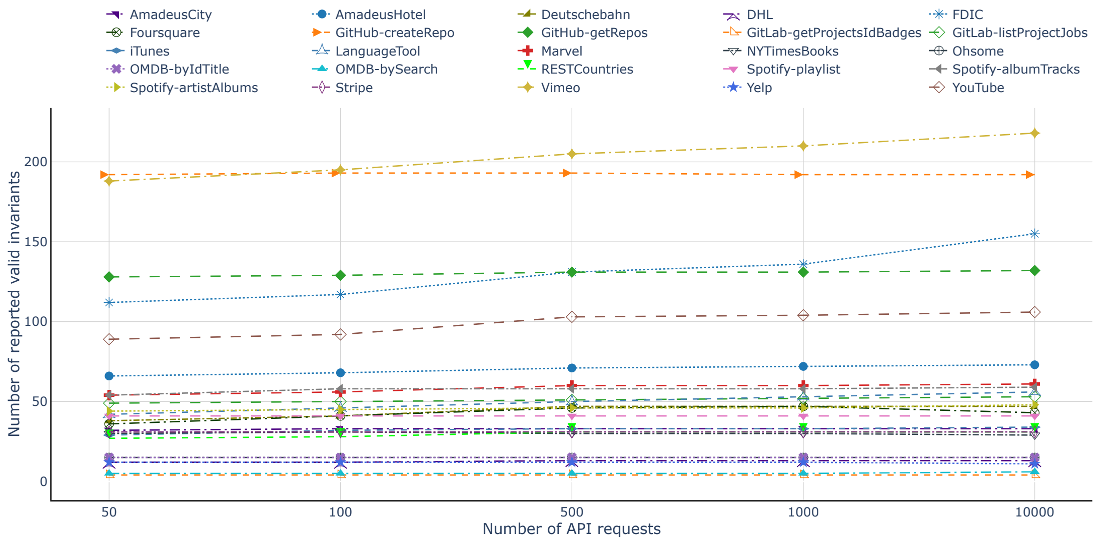


Fig. 6. How test suite size affects the number of valid invariants reported by AGORA+.

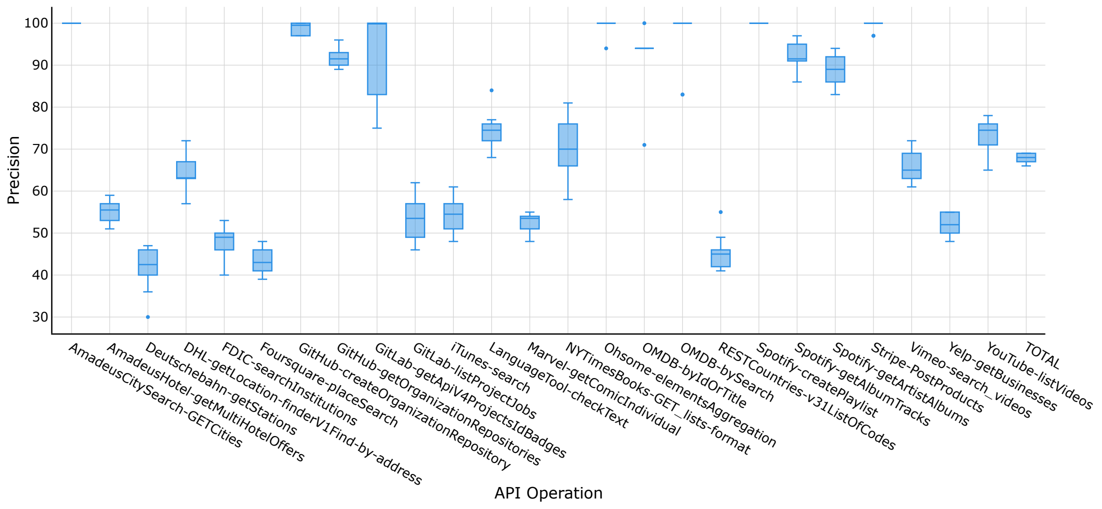


Fig. 7. Distribution of the precision of AGORA+ in the 50 test cases subset across 10 executions.

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 seeded *errors* in API responses using JSONMutator [10], 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 *defects* are seeded in the source

Table 4. Failure Detection Ratio per API operation.

API - Operation	Assertions (test oracles)	FDR (%)
AmadeusCitySearch-GETCities	33	51
AmadeusHotel-getMultiHotelOffers	69	60
Deutschebahn-getStations	38	21
DHL-getLocation-finderV1Find-by-address	12	48
FDIC-searchInstitutions	119	47
Foursquare-placeSearch	40	23
GitHub-createOrganizationRepository	193	93
GitHub-getOrganizationRepositories	130	65
GitLab-getApiV4ProjectsIdBadges	3	30
GitLab-listProjectJobs	53	37
iTunes-search	32	38
LanguageTool-checkText	42	29
Marvel-getComicIndividual	54	37
NYTimesBooks-GET_lists-format	31	50
Ohsome-elementsAggregation	15	74
OMDB-byIdOrTitle	15	36
OMDB-bySearch	5	21
RESTCountries-v31ListOfCodes	26	15
Spotify-createPlaylist	41	93
Spotify-getAlbumTracks	43	70
Spotify-getArtistAlbums	43	75
Stripe-PostProducts	30	58
Vimeo-search_videos	199	47
Yelp-getBusinesses	12	24
YouTube-listVideos	91	67
TOTAL	**1369**	**48**

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 [75, 80]. 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. This simulates a regression testing scenario where previously inferred test oracles are reused to ensure that code changes do not unintentionally impact existing functionalities. In case of changes in the API implementation or functional behavior, it would be necessary to run AGORA+ again to generate updated test oracles that align with the revised API behavior.

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 conservative input dataset. Test oracles were transformed into executable assertions (1369 in total) using PostmanAssertify (Table 4). 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 2.5M seeded errors: 25 operations x 1,000 API responses x 100 repetitions. For each mutated test suite, we used PostmanAssertify to generate a Postman collection in which the mutated outputs were used as response mocks.

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 that would result in violations of the API specification (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 [78, 93].

For string response fields, the "boundary" operator replaces the original string with boundary values, such as an empty string, an all-uppercase or all-lowercase string, or strings of minimum and maximum length (e.g., 1 or 10 characters). The "mutate" operator modifies strings by adding, removing, or replacing a single character, while the "addSpecialCharacters" operator introduces special characters like "/" or "*" into the string. Lastly, the "replace" operator completely substitutes the string with a random value.

For numerical response fields (doubles, longs, and integers), the "mutate" operator alters values by adding or subtracting a small delta, which is 0.5 for decimal numbers and 1 for integers. The "replace" operator, on the other hand, substitutes the number with a completely random value.

For arrays, the "disorderElements" operator changes the order of the elements within the array, while the "removeElement" operator randomly removes one or more elements. The "empty" operator clears the entire array, leaving it with no elements.

For Boolean fields, the "mutate" operator simply flips the Boolean value, changing true to false or vice versa.

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.

4.3.2 Experimental Results. Table 4 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 48% of the failures. This percentage ranged between 15% in the RESTCountries API and 93% in the operations "createOrganizationRepository" of the GitHub API and "createPlaylist" of the Spotify API.

Figure 8 illustrates the percentage of mutants generated by each mutation operator killed by AGORA+. AGORA+ showed its lowest performance with the "disorderElements" mutation operator (6% detection rate), as only 3 out of 25 API operations had test oracles related to array ordering. Conversely, string response fields are the most common in the API responses, accounting for 72% of mutations (1.8M out of 2.5M). AGORA+ performed best on string-type response fields, detecting 54% of mutations. Overall, AGORA+ proves effective in detecting failures related to incorrect formats (e.g., URLs or dates), predefined values, or relational patterns (e.g., equality or substring). However, it struggles to identify incorrect values that depend solely on the semantics of the application or underlying data. For instance, the primary source of undetected errors was modifying unique string values (e.g., for movie titles, "title=Alien" $\rightarrow$ "title=Aliens") introduced by the string-replace mutation operator. Detecting such semantic bugs is extremely challenging and falls outside the scope of this work.

In view of these results, we can answer RQ3 as follows:

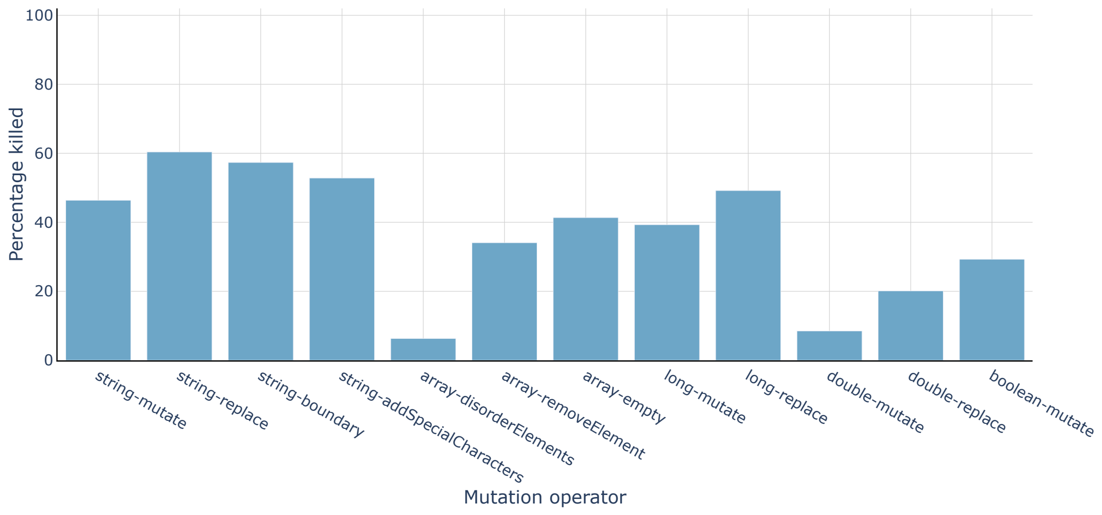


Fig. 8. AGORA+ performance detecting each mutation operator.

Answer to RQ3: How effective are the generated test oracles in revealing failures?

The test oracles generated by AGORA+ are effective in detecting failures, catching 5 out of every 10 errors systematically seeded in API responses.

5 DETECTED FAILURES

The invariants detected by AGORA+ revealed real-world bugs in some of the APIs, 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 in the Amadeus Hotel API. 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 failure detection: invalid invariants reveal failures observed during the invariant detection process, whereas violated valid invariants indicate failures observed in production. Invalid variants require manual inspection, whereas the violation of confirmed invariants can be automatically detected during testing. Overall, AGORA+ detected 32 domain-specific bugs (17 confirmed) in 13 operations from 11 APIs with millions of users worldwide, namely Amadeus Hotel, Deutschebahn, FDIC, Foursquare, GitHub, GitLab, Marvel, NYTimesBooks, OMDb, RESTCountries, 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 [15]. 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. Also, according to the description provided in the OAS specification, the property `code` of the `taxes` response field of this operation should be a two-letter country code, but the API always returns values such as “TOTAL_TAX” or “SERVICE_CHANGE”. This bug has not been confirmed yet.

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 was created with an incorrect license. More specifically, in some cases, when creating a repository with the "ncsa" or "postgresql" license, GitHub created a repository with the "other" 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.

GitLab. The suppression of two confirmed invariants (`return.image_url is Url` and `return.rendered_image_url is Url`) of the "getApiV4ProjectsIdBadges" operation uncovered badges with URLs containing whitespaces, rendering them invalid. These bugs were confirmed by the API providers, who created an issue to address them [9].

Foursquare. AGORA+ identified 9 bugs in this API, all of which have been confirmed by the API providers. The suppression of the confirmed invariants `return.menu is URL` and `return.website is URL` exposed locations with invalid URLs for menus and websites, such as "http://gh". These bugs have already been fixed. The suppression of the valid invariant `input.radius > return.distance` revealed cases where the API returned places outside the user-defined search radius. This was traced back to incorrect user-entered coordinates and has been corrected. Similarly, the suppression of `input.radius >= return.context.geo_bounds.circle.radius` exposed a similar issue, this bug has been fixed too. The suppression of `input.min_price <= return.price` and `input.max_price >= return.price` uncovered test cases where the API returned places outside the specified price range, these bugs have not been fixed yet. Additionally, the suppression of `return.stats.total_photos >= 0` revealed a flaw where negative values were assigned to the `total_photos` field when photos were demoted by the system or the Placemarker community. The API providers confirmed this as a bug and are working on a fix. Similarly, the suppression of `return.stats.total_photos >= size(return.photos[])` and `return.stats.total_tips >= size(return.tips[])` exposed test cases where the reported totals for photos and tips were less than the actual number of items in the corresponding arrays. These two bugs have been resolved.

FDIC. The API documentation specifies that the values of the `CONSERVE` and `LAW_SASSER_FLG` response fields should be a numerical flag (0 or 1), but the invariants `return.data.CONSERVE one of {"N","Y"}` and `return.data.LAW_SASSER_FLG one of {"N","Y"}` revealed that the documentation was inconsistent. The same happens with the TRUST response field, AGORA+ reported the invariant `return.data.TRUST one of {"0", "1"}`, whereas the documentation describes "00", "10" and "11" as the expected values. None of these reports have been confirmed yet.

RESTCountries. AGORA+ detected that the area in square kilometers of the Norwegian territories Svalbard and Jan Mayen was indicated as -1.0 (invariant `return.area>=-1.0`). This bug has been confirmed and fixed by the API providers [18, 19]. Also, the response fields documentation [20], indicates that the value of the `startOfWeek` response field can be either Sunday or Monday. However, the invariant `return.startOfWeek one of {"monday","saturday","sunday"}` identified "Saturday" as the value for Iran. This report has 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.

Deutschebahn. The violation of the confirmed invariant `return.localServiceStaff.availability.sunday.toTime is Time: 24-hour format` led to the detection of a train station (Coburg) with a value following an incorrect format for the end time of the availability of the service staff on Sundays (recorded as 18.30, instead of 18:30). This bug has been confirmed and fixed by the API providers.

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. The OAS specification states that the value of the `data.results.stories.items.type` response field should be either "interior" or "cover", but the invariant `return.type one of {"cover","interiorStory"}` revealed that this description is imprecise. Further analysis of the remaining API requests uncovered 32 different values for this response field. These reports have not been confirmed yet.

NYTimesBooks. The `isbn10` response field of this API is expected to store a string of 10 characters. However, the violation of the `LENGTH(return.isbn10)==10` invariant revealed two different failures: a book with an isbn10 of 9 characters, and a book with the string "isbn10" as value for the `isbn10` response field. These reports have not been confirmed yet.

OMDb. The `type` parameter of the OMDb API operations allows users to filter results by media type—namely "movie", "series" or "episode"— as specified in 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", despite being specified as a valid filtering value in the documentation. API developers have not confirmed these issues yet.

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 used the publicly available API specifications. However, the specifications of the Foursquare, iTunes, Ohsome, OMDb, RESTCountries, 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, we used the state-of-the-art Atlassian OAS syntactic validator [23] to ensure the OAS was consistent with the returned API responses. The updated OAS specifications 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 [64]) 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. If the documentation was not clear enough, we consulted the API providers about the expected API behavior.

Finally, the division of the dataset into random subgroups may have also affected the results. To mitigate this threat, we performed this division into subsets 10 times, computing the average performance between runs.

External validity. *To what extent can we generalize the findings of our investigation?* We evaluated AGORA+ on a set of 25 operations from 20 different APIs, and therefore our conclusions might 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 completeness of the OAS specification has a direct impact in the quality of the reported invariants. An incomplete OAS (e.g., with missing response fields) leads to an incomplete set of invariants. However, most existing tools for automated API testing provide features that ensure the syntactic validity of the OAS with respect to the returned API responses.

AGORA+ relies on test suite diversity to ensure accurate invariant detection. A lack of diversity in the test suite can lead to invalid or incomplete invariants that reflect only the API behavior within the test suite, without generalizing to the expected functionality. However, test suite diversity is a fundamental requirement in testing and can typically be achieved by applying standard test case design techniques.

The novel types of invariant proposed might not generalize beyond the selected APIs. To mitigate this threat, these invariants were created based on an analysis of a systematically collected dataset of 40 realistic APIs (702 operations) belonging to different domains [30]. 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

This paper introduces AGORA+, an 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 created Beet, a novel Daikon front-end for REST APIs described using OAS, and a customized version of Daikon supporting the detection of 106 distinct types of invariants in REST APIs. AGORA+ also integrates PostmanAssertify, a tool that converts the reported invariants into executable Postman assertions, making our approach readily applicable. Evaluation results on a set of 25 operations from 20 industrial APIs show that AGORA+ can generate hundreds of effective test oracles with just 50 requests in seconds, outperforming the default version of Daikon and AGORA—a previous version of our approach—by a margin of 100% and 33%, respectively. In addition, AGORA+ helped identify 32 issues in industrial APIs with millions of users, contributing to fixes and documentation updates, demonstrating its potential as a standalone testing technique. Other potential lines of work include developing invariant prioritization strategies [94], and enabling users to focus test oracle generation on specific variables [110]. Since it operates in a black-box mode, AGORA+ can be easily integrated into existing API testing tools supporting the OAS specification format.

DATA AVAILABILITY STATEMENT

Supplementary material includes the source code of the scripts and projects developed, the data generated in our experiments, and instructions on how to reproduce our evaluation. The artifact can be downloaded at [15] https://doi.org/10.5281/zenodo.12506791.

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] 2025. Beet repository. https://github.com/isa-group/Beet Accessed January 2025.

[2] 2025. Chai Assertion Library. https://www.chaijs.com/api/bdd/ Accessed January 2025.

[3] 2025. Daikon AGORA Docker image. https://hub.docker.com/r/javalenzuela/daikon_agora Accessed January 2025.

[4] 2025. Daikon AGORA repository. https://github.com/JuanCarlosAlonsoValenzuela/daikon_modified Accessed January 2025.

[5] 2025. Daikon developer manual. New formatting for invariants. https://plse.cs.washington.edu/daikon/download/doc/developer.html#New-formatting-for-invariants Accessed January 2025.

[6] 2025. Daikon front ends and instrumentation. https://plse.cs.washington.edu/daikon/download/doc/daikon.html#Front-ends-and-instrumentation Accessed January 2025.

[7] 2025. extJWNL (Extended Java WordNet Library) repository. https://github.com/extjwnl/extjwnl Accessed January 2025.

[8] 2025. GitHub API. https://developer.github.com/v3/ Accessed January 2025.

[9] 2025. GitLab issue. Invalid URLs in the getApiV4ProjectsIdBadges operation. https://gitlab.com/gitlab-org/gitlab/-/issues/473603 Accessed January 2025.

[10] 2025. JSONMutator. https://github.com/isa-group/JSONmutator Accessed January 2025.

[11] 2025. OpenAPI Specification. https://www.openapis.org Accessed January 2025.

[12] 2025. Postman API Platform. https://www.postman.com Accessed January 2025.

[13] 2025. PostmanAssertify repository. https://github.com/JuanCarlosAlonsoValenzuela/PostmanAssertify Accessed January 2025.

[14] 2025. RapidAPI API directory. https://rapidapi.com/products/api-hub/ Accessed January 2025.

[15] 2025. Replication package. https://doi.org/10.5281/zenodo.12506791 Accessed January 2025.

[16] 2025. Repository APIs.guru. https://apis.guru Accessed January 2025.

[17] 2025. RESTCountries API issue. Empty idd value. https://gitlab.com/restcountries/restcountries/-/issues/220#note_1726639875 Accessed January 2025.

[18] 2025. RESTCountries commit fixing country with negative area bug. https://gitlab.com/restcountries/restcountries/-/commit/ee498c74ad21c93b66a577d63d2c8eacefc58d42 Accessed January 2025.

[19] 2025. RESTCountries GitLab issue. Country with negative area. https://gitlab.com/restcountries/restcountries/-/issues/219 Accessed January 2025.

[20] 2025. RESTCountries response fields. https://gitlab.com/restcountries/restcountries/-/blob/master/FIELDS.md Accessed January 2025.

[21] 2025. Spotify Web API. https://developer.spotify.com/web-api/ Accessed January 2025.

[22] 2025. SRC Grand Finalists 2023. https://src.acm.org/grand-finalists/2023 Accessed January 2025.

[23] 2025. Swagger Request Validator. https://mvnrepository.com/artifact/com.atlassian.oai/swagger-request-validat Accessed January 2025.

[24] 2025. Vimeo API. Working with Authentication. https://developer.vimeo.com/api/authentication Accessed January 2025.

[25] 2025. Visa Developer Center. https://developer.visa.com Accessed January 2025.

[26] 2025. Write API test scripts in Postman. https://learning.postman.com/docs/writing-scripts/test-scripts/ Accessed January 2025.

[27] 2025. YouTube API. List videos operation. https://developers.google.com/youtube/v3/docs/videos/list Accessed January 2025.

[28] 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

[29] 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

[30] 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

[31] Juan C. Alonso, Sergio Segura, and Antonio Ruiz-Cortés. 2023. AGORA: Automated Generation of Test Oracles for REST APIs. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1018–1030. https://doi.org/10.1145/3597926.3598114

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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.

[39] 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.

[40] 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

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, Hua Zhong, and Tao Huang. 2023. Detecting Isolation Bugs via Transaction Oracle Construction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1123–1135. https://doi.org/10.1109/ICSE48619.2023.00101

[48] Michael D. Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *WODA 2003: Workshop on Dynamic Analysis*. Portland, OR, USA, 24–27.

[49] 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.

[50] 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.

[51] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph. D. Dissertation. University of California, Irvine.

[52] 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

[53] 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

[54] 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

[55] 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

[56] 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

[57] 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

[58] 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.

[59] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 27 (Nov. 2023), 41 pages. https://doi.org/10.1145/3617175

[60] 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

[61] 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

[62] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving Semantics-Aware Fuzzers from Web API Schemas. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 345–346. https://doi.org/10.1145/3510454.3528637

[63] Soneya Binta Hossain and Matthew Dwyer. 2024. TOGLL: Correct and Strong Test Oracle Generation with LLMs. arXiv:2405.03786 [cs.SE] https://arxiv.org/abs/2405.03786

[64] 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

[65] 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

[66] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[67] Daniel Jacobson, Greg Brail, and Dan Woods. 2011. *APIs: A Strategy Guide*. O'Reilly Media, Inc.

[68] 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

[69] 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.

[70] Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. 2023. Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1232–1243. https://doi.org/10.1145/3597926.3598131

[71] M. Kim, S. Sinha, and A. Orso. 2023. Adaptive REST API Testing with Reinforcement Learning. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 446–458. https://doi.org/10.1109/ASE56229.2023.00218

[72] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2025. LlamaRestTest: Effective REST API Testing with Small Language Models. arXiv:2501.08598 [cs.SE] https://arxiv.org/abs/2501.08598

[73] Myeongsoo Kim, Tyler Stennett, Dhruv Shah, Saurabh Sinha, and Alessandro Orso. 2024. Leveraging Large Language Models to Improve REST API Testing. arXiv:2312.00894 [cs.SE]

[74] Myeongsoo Kim, Tyler Stennett, Saurabh Sinha, and Alessandro Orso. 2024. A Multi-Agent Approach for REST API Testing with Semantic Graphs and LLM-Driven Inputs. arXiv:2411.07098 [cs.SE] https://arxiv.org/abs/2411.07098

[75] 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

[76] 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

[77] 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

[78] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Józala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering* 40, 1 (2014), 23–42. https://doi.org/10.1109/TSE.2013.44

[79] 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

[80] 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*.

[81] 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

[82] 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.

[83] 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*.

[84] 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

[85] 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

[86] 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

[87] 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

[88] 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

[89] 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.

[90] 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.

[91] 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.

[92] Lianglu Pan, Shaanan Cohney, Toby Murray, and Van-Thuan Pham. 2024. EDEFuzz: A Web API Fuzzer for Excessive Data Exposures. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 45, 12 pages. https://doi.org/10.1145/3597503.3608133

[93] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. Advances in Computers, Vol. 112. Elsevier, 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

[94] Fabrizio Pastore and Leonardo Mariani. 2015. ZoomIn: Discovering Failures by Detecting Wrong Assertions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 66–76. https://doi.org/10.1109/ICSE.2015.29

[95] 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

[96] Leonard Richardson, Mike Amundsen, and Sam Ruby. 2013. *RESTful Web APIs*. O'Reilly Media, Inc.

[97] 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

[98] 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

[99] 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

[100] 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

[101] 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

[102] 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

[103] 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*.

[104] 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

[105] 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

[106] 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

[107] Rahulkrishna Yandrapally, Saurabh Sinha, Rachel Tzoref-Brill, and Ali Mesbah. 2023. Carving UI Tests to Generate API Tests and API Specification. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1971–1982. https://doi.org/10.1109/ICSE48619.2023.00167

[108] 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

[109] 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

[110] Lucas Zamprogno, Braxton Hall, Reid Holmes, and Joanne M. Atlee. 2023. Dynamic Human-in-the-Loop Assertion Generation. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2337–2351. https://doi.org/10.1109/TSE.2022.3217544

[111] 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

[112] 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

[113] 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

[114] 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

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009