SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **00**:1–31
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Automated Black-Box Testing of Nominal and Error Scenarios in RESTful APIs

Davide Corradini[1], Amedeo Zampieri[1], Michele Pasqua[1],
Emanuele Viglianisi[2], Michael Dallago[3], Mariano Ceccato[1]

[1] *University of Verona. Verona, Italy*
[2] *Fondazione Bruno Kessler. Trento, Italy*
[3] *University of Trento. Trento, Italy*

SUMMARY

RESTful APIs (or REST APIs for short) represent a mainstream approach to design and develop Web APIs using the REpresentational State Transfer architectural style. Black-box testing, which assumes only the access to the system under test with a specific interface, is the only viable option when white-box testing is impracticable. This is the case for REST APIs: their source code is usually not (or just partially) available; or a white-box analysis across many dynamically allocated distributed components (typical of a micro-services architecture) is computationally challenging.
This paper presents RESTTESTGEN, a novel black-box approach to automatically generate test cases for REST APIs, based on their interface definition (an OpenAPI specification). Input values and requests are generated for each operation of the API under test with the twofold objective of testing nominal execution scenarios and error scenarios. Two distinct oracles are deployed to detect when test cases reveal implementation defects. While this approach is mainly targeting the research community, it is also of interest to developers because, as a black-box approach, it is universally applicable across different programming languages, or in the case external (compiled only) libraries are used in a REST API.
The validation of our approach has been performed on more than one hundred of real-world REST APIs, highlighting the effectiveness of the approach in revealing actual faults in already deployed services.

Received . . .

KEY WORDS: Black-box testing, REST APIs, Automatic test case generation, Oracle.

1. INTRODUCTION

REST APIs are the de-facto standard to implement and grant remote access to Web APIs. They are so largely accepted and adopted that the *Berlin Group Initiative*[1] elaborated a standard based on REST APIs for unifying the European Banking APIs. This initiative was meant to address the PSD2 European Union directive[2], that requested banks to open their customer data to authorized third-party service providers. Moreover, reference implementations of PSD2 compliant banking APIs are mostly available in the form of REST APIs[3].

REST APIs are often components of micro-services architectures [35], according to which each component should be small and assigned just one (or very few) responsibilities, resulting in a high

[1] https://www.berlin-group.org/
[2] https://ec.europa.eu/info/law/payment-services-psd-2-directive-eu-2015-2366_en
[3] https://www.openbankproject.com/

number of simple components. Distinct components are usually deployed in different containers that can be dynamically allocated and deallocated, possibly across different hosts (for load balancing).

Such a dynamic usage of REST APIs, that spread across many independent containers, poses peculiar challenges to automated analysis and testing with white-box approaches. A black-box approach, instead, gives up on the possibility to exploit program information available in the source code. In fact, it only relies on a well-defined interface to access the REST API and does not need to cope with the very complex internal details of how components are deployed and run.

Moreover, the implementation of REST APIs might include commercial third-party libraries or frameworks, that come in the form of compiled code. When source code is not (or only partially) available, a black-box perspective in automated testing of REST APIs is a natural option.

In this paper we present RESTTESTGEN, a novel approach to automatically generate test cases for REST APIs. This approach relies on the definition of the interface to interact with a REST API (an OpenAPI specification), including the list of operations available and the format of the input/output data of their requests and responses. We propose the *Operation Dependency Graph* (ODG) as a way to explicitly model data dependencies among operations that can be inferred from the REST API interface. This graph is updated during the test cases generation to dynamically decide when a new operation is ready to be tested, i.e., when the values of all its required input parameters could be guessed.

The dynamic testing order based on the ODG is just an approximation of the (unknown) optimal testing order. In fact, the list of the *data dependencies* that can be inferred from the OpenAPI specification might be incomplete . With access to source code and documentation, a more comprehensive list of dependencies could have been filled. For instance, precise data dependencies between input and output parameters of API operations could be inferred by analyzing what parameters access the same table and column in the database. The ODG is not meant to fully replace source code access, but it provides an approximation of it. The limitation of this approximation can still be acceptable by developers, when it is paired with the invaluable benefit of a fully automated testing approach that can be applied to any programming language, even when using third parties closed source components.

RESTTESTGEN aims at testing REST APIs according to two perspectives. *Nominal execution scenarios*, meant to test the system using input data as documented in the interface; and *error execution scenarios*, which exploit input data violating the interface to expose implementation defects and unhandled exceptional flows.

An extensive empirical validation has been conducted, involving 116 real-world REST APIs for which only black-box access is available. RESTTESTGEN was able to test these case studies to a large extent, revealing a considerable number of implementation defects.

This work builds on top of a companion conference paper [44]. The novel contribution consists of the following parts.

- A major extension of RESTTESTGEN, that now offers novel features to test complex APIs, such as the support for authentication (presented in Section 5.2) and a better management of the testing time budget (introduced in Section 5.1 and in Listing 2), that allows a more flexible empirical investigation.

- The empirical assessment has been extended by including novel case studies, including also some from mainstream vendors that require authentication and enforce access control (experiment presented in Section 7.3).

- Presentation of a few actual defects that RESTTESTGEN could detect in the considered case studies, showing practically how real-world industrial REST APIs might fail. Based on these defects, a list of considerations are formulated about our approach and about automated testing of REST APIs (Section 8.1).

- Assessment of the contribution of the ODG to test case generation, by comparing test cases generated by RESTTESTGEN when the order of the operations to test is based on the ODG with test cases generated with a random operation ordering. (Section 7.4)

DOI: 10.1002/stvr

The rest of the paper is organized as follows. Section 2 covers the background on REST APIs. Section 3 proposes an overview of the RESTTESTGEN modules, that are presented in detail in the subsequent sections. Section 4 defines the Operation Dependency Graph, while Section 5 and Section 6 describe, respectively, the *Nominal Tester* and the *Error Tester* modules. In Section 7, the empirical assessment of RESTTESTGEN is presented. Finally, after discussing related work in Section 9, Section 10 closes the paper.

2. BACKGROUND

2.1. REST APIs

A RESTful API (or REST API for short) is an API that respects the REST (REpresentational State Transfer) architectural style [13]. Nowadays, most of the APIs use a RESTful architecture over the HTTP protocol to manage resources, allowing clients to access and manipulate them using a set of stateless operations.

REST APIs provide a uniform interface to create, read, update and delete (CRUD) a resource. A resource is generally identified by an HTTP URI, and CRUD operations are usually mapped to the HTTP methods POST, GET, PUT and DELETE to the resource URI.

For example, consider a REST API managing a collection of pets. A possible HTTP URI pointing to the resource could be `/pets`. In this case, the HTTP operation `GET /pets` is used to retrieve the list of the pets and `POST /pets` could be used to add a new pet to the collection.

The resource URI and the HTTP methods may accept input parameters to specify additional information for executing the API operations, such as the identifier of the object to retrieve (e.g., `/pets/{petId}`) or a structured object to be added to the collection using the POST method.

2.2. OpenAPI and Swagger

OpenAPI[4] defines a standard to document REST APIs. According to OpenAPI, an API service is described using a structured file (either YAML or JSON) that specifies how to reach the API using a URI, what authentication schema is used and the details of all the operations available in the API: the input parameters (and their schema) to be used in requests and the schemas of responses. Previous versions of this specification (older than version 3.0.0) were called *Swagger*. In this paper we will use the term OpenAPI specification to mean both the old and the new format of an API specification, since older versions are still used, and anyway there is no major difference.

Listing 1 contains an example of an OpenAPI specification for *PetStore*, an API for managing *Pet* resources within a store. After an initial header that specifies versions and licenses, the field *servers* contains the base URL of the API (`http://petstore.swagger.io/v1` in the example).

The array *paths* contains the list of URL paths available in the API. In our example, there are two paths, i.e. `/pets` and `/pets/{petId}`.

Each path supports one or more HTTP method operations. Combination of path and method are usually specified by an *OperationID*. The method *GET* in `/pets` (*getPets*) is used to retrieve the list of all the pets. The method *GET* of the path `/pets/{petId}` refers to the operation *getPetById*, meant to retrieve the *Pet* object that matches a specific *petId*. Path parameters are specified directly in the path URL enclosed by curly braces, such as the *petId* input parameter in our example.

Modifiers can be used to attach constraints to data fields. For instance, the modifier *required* specifies that a parameter is mandatory, and it can not be omitted. Moreover, each request input and output is associated with a *schema* that specifies its type and, optionally, a set of constraints on its value (e.g., a *min* or *max* value for numeric parameters). Types can be atomic (e.g., integers and strings) or structured (i.e., compound objects). For instance, the parameter *petId* (line 41) is of type string (line 46), while the response is expected to be in JSON (line 51) according to the schema *Pet*, which is composed of the fields *id*, *name* and *tag*, as specified at lines 69-75.

[4]https://www.openapis.org/

DOI: 10.1002/stvr

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: getPets
      tags:
        - pets
      responses:
        '200':
          description: PetIds
          content:
            application/json:
              schema:
                type: array
                items:
                  type: object
                  properties:
                    petId:
                      type: integer
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
  /pets/{petId}:
    get:
      summary: Info for a specific pet
      operationId: getPetById
      tags:
        - pets
      parameters:
        - name: petId
          in: path
          required: true
          description: The id of the pet to retrieve
          schema:
            type: string
      responses:
        '200':
          description: Expected response to a valid request
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pet"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
    # ...
components:
  schemas:
    Pet:
      type: object
      required:
        - id
        - name
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
        tag:
          type: string
```

Listing 1. OpenAPI specification example

The OpenAPI specification not only describes the response format in the nominal case (i.e., response code `200`, line 17), but also the response format of the API when an error occurs (line 33).

3. APPROACH OVERVIEW

RESTTESTGEN is a black-box tool, intended to automatically generate test cases for REST APIs. As a black-box approach, RESTTESTGEN does not assume access to the source code nor to the compiled binary code of the REST API under test. It only assumes to have input/output access to the API via the HTTP protocol. The OpenAPI specification of the REST API should be also available, to know which operations can be called and their input/output data format to send well-formed HTTP requests.

A black-box approach is the only option when the source code is not available, or only partially available, e.g., when third-party components or commercial libraries are integrated, whose source code is not available. Additionally, a black-box approach is a valuable option when testing APIs with an architecture that is very complex for a white-box approach, e.g., because consisting of many (micro-)services, possibly developed with different languages and technologies. In fact, a black-box approach is independent from the programming language used to implement the API to test.

RESTTESTGEN includes different modules, as shown in Figure 1. The tool takes as input the OpenAPI specification of the service under test to have information on the available REST API endpoints, the available operations and the input/output data formats.

The first module analyzes the specification and computes the *Operation Dependency Graph*, a graph that models the data dependencies among the operations available in the service. This graph is meant to help RESTTESTGEN in sorting the operations to test, whose order will depend on their data dependencies. In particular, the operations whose output is needed as input to other operations are tested first. Then, RESTTESTGEN can use the data produced by these operations to feed the subsequent operations with meaningful inputs.

The next module, namely the *Nominal Tester*, reads the Operation Dependency Graph and the OpenAPI specification to automatically create test cases for the REST API. We called the test cases

DOI: 10.1002/stvr

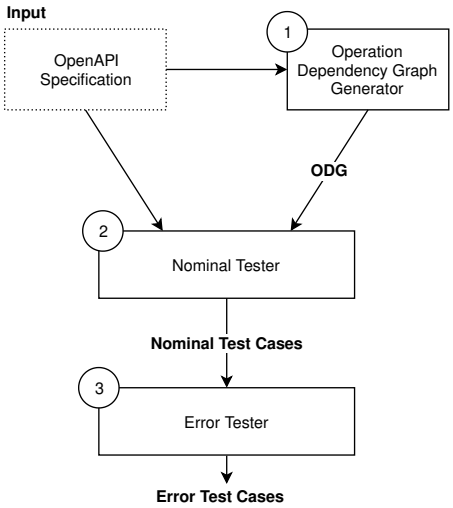

Figure 1. Automated test case generation: structure overview

generated by this module *nominal* test cases since they are created according to the specification, trying as much as possible to follow the specified data constraints.

Nominal tests represent the input for the subsequent module, the *Error Tester*. This last module applies a catalog of mutation operators to the nominal tests with the aim of violating data constraints from the OpenAPI specification, verifying how the REST API handles malformed inputs. For this reason, the tests generated by this module are called *error* test cases.

4. OPERATION DEPENDENCY GRAPH

This section describes the *Operation Dependency Graph* and how to build it, starting from the information present in the OpenAPI specification.

4.1. Graph Construction

The *Operation Dependency Graph*, or ODG for short, is a directed graph $G = (N, V)$. Nodes N are the operations in the REST API. The graph has an edge $v \in V$, with $v = n_2 \rightarrow n_1$, when there exists a data dependency between nodes n_2 and n_1. We define a data dependency between two nodes n_2 and n_1 when there exists a *common field* in the output (response) of n_1 and in the input (request) of n_2. The intuitive meaning of this dependency is that n_1 should be tested before n_2 because the output of n_1 could be used to guess valid input values to test n_2.

We define two fields as *common* when:

- They are of atomic type (i.e., string or numeric), and they have the same name;
- They are of non-atomic type (i.e., structured), and they are associated to the same schema.

Edges are labeled with the name of *common field(s)* between the source and target nodes.

As an example, consider a segment of the specification in Listing 1 reporting two operations: operation *getPets* lists all the *petId* identifiers of all the pets available in the shop; operation *getPetById* returns all data related to a particular object of type *Pet*, whose schema is defined in lines 63-75 in Listing 1.

These two operations have a data dependency on the common field *petId*. The field is present in the output of *getPets* (lines 26 in Listing 1) and in the input of *getPetById* (lines 41 to 46 in Listing 1). Thus, the corresponding ODG shown in Figure 2 has two nodes, one for each of these two operations, and an edge labeled *petId* from *getPetById* to *getPets*.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

The intuitive meaning of this graph is that to test the operation *getPetById* we require a valid *petId* value, that would be difficult to guess with a black-box testing framework. So, the operation *getPets* should be tested earlier in order to fetch a valid value for *petId*.

The Operation Dependency Graph can grow quadratically with the number of nodes, because, in the worst case, each pair of nodes is connected by a data dependency. In a complex REST API, with many dependent operations, the ODG will represent a valuable support to sort the operations to test and to plan for the acquisition of the needed data to be used during testing.

In the following, we will present our approach using the dependencies in the ODG to decide in which order to test the operations in a REST API.

4.2. Dependency Inference

The example in Listing 1 represents the ideal case of a correctly defined OpenAPI specification. However, there are no constraints forcing developers to use the same syntactic name for fields with a (semantic) dependency. For instance, the field *petId* to be used in *getPetById*, could be simply called *id* and a developer would still understand what data to use for that field.

Alternatively, field names could contain typos (*petId* could become *pettID* or *pedID*), or characters in the wrong case (*petID* or *petid*). Thus, a perfect match among field names might fail. For this reason, the matching algorithm adopts a relaxed approach that tolerates few typing mistakes.

In particular, the following strategies are used to match field names.

Case insensitivity The comparison of field names is case insensitive to work around developer mistakes in using a consistent casing across operations.

ID completion When a field is just named *id* we add a prefix to its name. In the case of a field belonging to a structured object, the prefix is the name of the object. For instance, the field *id* of the object *pet* is renamed *petId*. In the case of a field that is not part of a structured object, it is prefixed with the name of the operation in which it is involved, after removing get/set verbs from the operation name. For instance, the operation *getPet* becomes *Pet* after removing the verb "get", and it is used to change the field *id* into *petId*.

Stemming Instead of requiring two field names to be exactly the same, we tolerate some difference. We apply the Porter Stemming algorithm [45] to each parameter name to compare their *stem* instead of their exact names. For instance, two parameters named *pet* and *pets* are converted to the same normalized term *pet* and considered as the same parameter.

5. TESTING OF NOMINAL CASES

The aim of the *Nominal Tester* module is to automatically generate test cases meant to run nominal interactions as they are documented in the OpenAPI specification. To achieve this objective, three issues need to be addressed. In particular, (i) choosing the order in which operations should be tested, (ii) generating input values, and (iii) deciding if the test scenario exposed a fault.

5.1. Operation Testing Order

To elaborate the testing order among operations we resort to two distinct dependencies: the CRUD semantics and the data dependencies from the ODG (see Section 4).

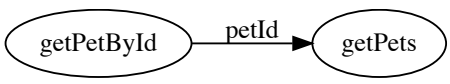

Figure 2. Sample of Operation Dependency Graph

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
while (length(graph.vertexes) > 0) {
    # get graph leaves
    to_test = graph.leaves

    # random vertex if there is no leaf (only cycles)
    if length(to_test == 0):
        to_test = [random(graph.vertexes)]

    # sort leaves according to CRUD
    to_test_sorted = sort_for_crud(to_test)

    # test each leaf
    for (operation: to_test_sorted) {
        is_tested = run_test_on(operation, operation_budget)

        # remove successfully tested leaves
        if is_tested:
            graph.remove(successful)
    }

    # budget expired check
    if global_budget_exceeded():
        break
}
```

Listing 2. Pseudo-code for ordering the operations to test.

CRUD semantics. To successfully test an operation, a particular resource might be required to be in a certain state. Considering the CRUD semantics, a successful DELETE operation requires that the target resource is available, so the resource should be first added using a POST operation. A similar argument holds for PUT/PATCH operations: updates should be performed only on existing resources. Conversely, a POST operation that creates a resource requires the resource not to exist yet. The dependencies related to the CRUD semantics are modeled based on the following priorities.

1. **HEAD** operations are the first in the priority list, because they are often used to check the validity of an API operation and to retrieve the header of a resource.

2. **POST** operations are commonly used to add a new resource; they have a high priority since other operations may reference these operations and their parameter values.

3. **GET** operations retrieve information of an existing resource. Retrieved information can be used as input values for other operations.

4. **PUT/PATCH** operations are used to modify an existing resource with new parameter values.

5. **DELETE** operations have the lowest priority because they remove existing resources that can not be referenced anymore by other operations.

Zhang et al. [47] also proposed a testing approach based on the CRUD semantics using templates of pairs of related operations (e.g., POST+DELETE and POST+PUT), and this showed a positive impact on the test coverage. Instead of using templates of operation pairs, we enforced an order to prioritize the operations to test.

ODG data dependencies. Data dependencies among operations are read from the ODG and used to sort the operations to test. The goal is to maximize the chances of reusing data already collected from previously tested operations to test new operations.

The pseudo-code in Listing 2 shows the algorithm that we use to combine CRUD dependencies and ODG data dependencies, to choose operations testing order.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

The algorithm starts testing the graph leaves, extracted with the query at line 3. Leaves are those operations with no outgoing edges, i.e., operations with no dependencies. No dependency means either no input fields (e.g., *getPets* in Listing 1) or input fields found in the output of no operations.

In case there are no leaves, then every graph node has at least one outgoing edge (i.e., the graph is cyclic). Indeed, in this case, there is at least a cycle in the graph involving a subset of the graph nodes. Cycles should be broken to start testing. In this case, at line 7, we randomly pick a node to start testing, i.e., we open the cycle at a random position.

If we only consider data dependencies, all these leaf nodes would have equivalent precedence (with respect to our selection algorithm, based on the ODG), because they have no outgoing edges in the ODG. However, their order might still be optimized according to their CRUD semantics. Thus, at line 10, leaf nodes are then sorted according to their CRUD dependencies, if any, to increase the likelihood of testing an operation in the correct context (e.g., deleting or changing a resource after it has been created).

Then, at line 14, the algorithm attempts to test each leaf operation, in the sorted order. Each operation is fuzzed, with different heuristics, to generate its input values (input generation heuristics are described in more detail in Section 5.2). Multiple input generation attempts may be needed to guess valid input values. The task of fuzzing an operation is associated with a budget (`operation_budget`) that determines when to halt the process, and then move to the next operation to fuzz. This budget can be specified as maximum number of attempts to test an operation, or as maximum amount of time to spend testing it. The value of `operation_budget` is a global constant set during the initialization of the tool. Our oracles (see Section 5.3) are used to decide when a test can be considered successful.

We keep track of successfully tested operations (by means of the local variable `is_tested`). We consider an operation to have been successfully tested when we observe a testing scenario with a `2XX` response status code, that stands for a correct execution. Such an execution, in fact, might have provided useful output data to be used as input when testing next operations. In case an operation is successfully tested, it is removed from the graph (at line 18). In this way, we remove from the graph those dependencies that are now satisfied, and now operations that only depended on the just tested operation will become leaf nodes in the updated graph. They will be tested in the next iterations of the algorithm.

The main loop (line 1) completes when the graph remains with no nodes, namely when all the operations have been tested (testing complete), or when the global testing budget expires (line 22). Again, the testing budget can be set as maximum number of attempts or as maximum amount of elapsed time.

It is important to note that the operations order cannot be completely precomputed off-line, it is continuously updated at testing-time. In fact, we can mark a node as tested, and remove it from the graph, only after we could test it, and this is known only at testing time.

5.2. Input Value Generation

In order to test an operation, we need to guess appropriate input values. Input values are generated for each request using a probabilistic algorithm. With high probability (e.g., 80%) the algorithm applies a strategy based on the *response dictionary*, because reusing observed data is very likely to be effective in testing new operations. In the remaining cases (e.g., with 20% probability or when the former strategy fails), a new parameter value is generated starting from its schema.

Response dictionary. This heuristic is meant to reuse the knowledge of already tested operations to test new operations. Suppose, for instance, that we need to test the operation *getPetsById* in the *PetStore* API. Our approach would need to have prior knowledge of valid *pet ids*.

For this reason, inspired by Ed-douibi et al. [12], we use the concept of *Response Dictionary*. The Response Dictionary is a map between field names and their observed values. For each operation that is successfully tested, the values of all the output fields that can be found in the response content are saved in the Response Dictionary, so that the values can be reused later when an input field with the same name is needed.

DOI: 10.1002/stvr

One of our extensions with respect the approach by Ed-douibi et al. is represented by the matching algorithm that we use to look up into the Response Dictionary. While Ed-douibi et al. require an exact match between the name of the input value and the key in the dictionary, we tolerate some differences. In fact, our matching algorithm takes as input the field name for which a value is needed, and returns the key with the *closest* name among the keys in the dictionary. A non-perfect match is needed because of naming inconsistencies among input/output fields, due to implicit assumptions of developers, or because of typing errors (as discussed in Section 4.2).

The name matching algorithm implements the following look-up strategy:

1. look for a key with an exact match with the field name;
2. look for a key with an exact match with the concatenation of object name and field name (e.g., *petId* matches *pet*+*id*);
3. look for a key with edit distance less than a given threshold thr to the field name;
4. look for a key with edit distance less than a given threshold thr to the concatenation of object name and field name (e.g., *pets*_*Id* matches *pet*+*id*);
5. look for a key that is a substring of the field name to find.

In the low probability case and when the look-up strategy fails, the schema-based field generation is applied.

Schema-based field generation. A new parameter value is generated respecting the parameter schema type and its constraints, defined in the OpenAPI specification.

Default and example values. The OpenAPI grammar supports an option to specify default values for input parameters and examples of values to be used. In case these values are specified, they can be used to test an operation.

Enum. When the type is enum, a value is randomly picked with uniform probability among the closed set of available values.

Random input generation. A random value that matches the type of the input field (e.g., a random integer/decimal number or a random string) is generated. In particular, in half of the cases, we generate zero (on numeric input) or empty string (on string input). In the other half, a numeric value is randomly picked (with uniform probability) from the allowed range. A random string is generated by concatenating random alphanumeric characters respecting the parameter schema constraints *minLength* and *maxLength*.

Authentication. Some APIs can be used with no authentication, while others require clients to certify their identity before allowing access. Hereinafter, we refer to the former as *free access APIs* and to the latter as *access controlled APIs*. In order to deal with access controlled APIs, we have equipped RESTTESTGEN with the capability to attach authentication tokens to the requests sent to the services under test.

Services that expose a web interface often support password-based authentication. However, APIs usually adopt authentication based on a token (commonly called API-key). An authentication token could be obtained using identity providers via OAuth (e.g., login with Facebook or with Google accounts). Alternatively, static tokens are used (at least in some of the 29 APIs in our experimental validation). Obtaining an authentication token involves some manual effort, because a distinct account requires to be created for each REST API. This operation often consists in applying for the account that a moderator has to approve, especially for free-of-charge accounts. When an access token is available, it should be saved in the configuration file of RESTTESTGEN, so that it can be used when testing the corresponding API.

The authentication token is included in HTTP requests, either as a query parameter (in the form `GET /myEndpoint?tokenParamName=xxxxxx`), or as header parameter (e.g., `Authorization: Bearer xxxxxx`).

DOI: 10.1002/stvr

Considering that authentication depends on the REST API under test, the configuration file of RESTTESTGEN allows to specify both the token value and the protocol to send it (header or query).

When generating test cases, if authentication is required and properly configured, RESTTESTGEN will not fuzz the authentication input field; the token value from the configuration file will be used instead. This feature allows RESTTESTGEN to test both free access and access controlled APIs.

5.3. Oracles

Our approach includes two oracles to assess if the automated test case generation is successful, based on the status code and on the match with the declared schema.

Status Code Oracle. The status code is a three digit integer value in the HTTP response, meant to describe the outcome of a request. A status code of the form `2XX` stands for a correct execution, for instance `200` means OK, while `201` means that a resource has been successfully created.

A status code `4XX` stands for an error that was correctly handled, e.g., `400` stands for *Bad Request* and `404` stands for *Not Found*. Conversely, a status code `5XX` means that the server encountered an error that was not handled correctly, e.g., `500` means server crash.

We use the response status code as an oracle to assess if a test case was successfully executed, as described in the following.

`2XX` Status Code When obtaining a status code `2XX` in a response, we assume that our approach correctly guessed the input values to test an operation according to a nominal scenario. We conclude that this operation was successfully tested, so we mark this operation as tested and we can use the data in the response to populate the Response Dictionary.

`4XX` Status Code This status code means that the testing was not successful. It could be due to two distinct reasons: either incorrect input values have been rejected by the server, or input values were correct and they exposed an implementation defect. However, from a black-box viewpoint, we can not tell which of the two cases applies. Conservatively, we assume that the correct input values have not been guessed. Thus, we discard this scenario and we continue the test case generation.

`5XX` Status Code This status code means an internal server error occurred, e.g., status code `500` means that a server crash was not handled gracefully. A `5XX` status code is probably due to a programming defect that should be fixed. So, this is an interesting scenario to document with a test case.

Our approach emits JUnit test cases for those interactions that cause status codes `2XX` and `5XX`, to let a developer quickly replicate these scenarios.

Response Validation Oracle. The OpenAPI specification documents the operation responses and their schema, i.e., the intended status code and the fields in the responses, as in lines 47 to 59 of the example in Listing 1. Consistency between actual responses and their schema is important for remote programs that connect to a REST API. In fact, remote connection might fail in parsing inconsistent or malformed responses, and cause service discontinuity. Our second oracle reveals a mismatch between the intended response syntax (documented in the specification) and the actual response (observed at execution time), by using a schema validation library, namely *swagger-schema-validator*[5], on each response (either successful responses and error responses). In case of mismatch, a JUnit test case is emitted to document the defect.

For instance, consider a nominal execution of the operation *getPetById*. The operation response must contain an object of type *Pet*, as shown in the response example in Listing 3. This *Pet* object matches the *Pet* schema specified at lines 63 to 75 in Listing 1, i.e., the three properties *id*, *name* and *tag* are present and have correct types.

[5]https://github.com/bjansen/swagger-schema-validator

DOI: 10.1002/stvr

```
{
    "id": 1,
    "name": "doggy",
    "tag": "dog"
}
```

Listing 3: Example of *getPets* response compliant with the schema.

Listing 4 shows, instead, a case where the response content does not match the schema. Indeed, the object *Pet* in this example misses the parameter *name*. This particular execution output is classified as an implementation error by the oracle.

```
{
    "id": 1
    "tag": "dog"
}
```

Listing 4: Example of *getPets* response NOT compliant with the schema.

Limits of the Proposed Oracles. OpenAPI specifications do not come with a description of services business logic, so our oracles cannot detect faults related to deviations from the intended behavior of the APIs, that cannot be described in the specification. Additionally, our oracles cannot detect faults when the response status code is inconsistent with the internal execution, e.g., our oracles cannot realize the presence of a defect when an API hides an internal crash by returning a `200` status code. Furthermore, we are not considering, at the moment, faults related to security vulnerabilities, like SQL-injections or access control issues. We plan to extend RESTTESTGEN with oracles taking into account security-related aspects as a future work.

6. TESTING OF ERROR CASES

The objective of the *Error Tester* module is to test exceptional scenarios for the REST API, to assess if the REST API handles wrong requests in the appropriate way, i.e., they are discarded and errors are handled gracefully. To this aim, this module starts from nominal executions and mutates them to generate malformed and inconsistent executions. In case a mutated request is not discarded or if it causes a fatal error, a defect is detected.

6.1. Mutation Operators

Nominal test cases are changed according to a catalog of mutation operators. Currently the subsequent mutation operators are available.

Missing required In the OpenAPI grammar, input parameters support the modifier *required*, which means that a field is mandatory. A request that misses a required field should be discarded by the REST API. This mutation consists in altering a request by removing a required input field.

Wrong input type Input parameters are strongly typed. This mutation alters an existing test case by changing the value of an input parameter such that its type becomes incorrect. In case the declared type is string, the new value is a random number (integer or float). In case the declared type is numeric, the new value is a random string. In case of type enum, a random value is picked outside the set of values in the enumeration.

Constraint violation The OpenAPI grammar also allow to specify additional constraints for strings (e.g., *minLength* and *maxLength*) and for numeric values (e.g., *min* and *max* values). This mutation operator changes the value in a request such that it violates such constraints. For instance, a string is trimmed or extended with a random suffix, or a numeric value is decreased/increased by a random delta until it violates the intended bounds.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

A nominal test is mutated many times, by applying each mutation operator once per each input field in the request. Mutated requests are then sent to the REST API and the response is analyzed by the oracles.

6.2. Oracles

Similarly to the Nominal Tester, also the Error Tester is supported by two oracles.

Status Code Oracle. The status code in the response is inspected to classify whether this test case reveals a programming defect.

`2XX` Status Code This status code stands for a correct execution. This means that incorrect inputs, that should have been rejected, are instead processed as valid. This scenario exposes a mismatch between the features declared in the OpenAPI specification and those actually implemented.

`4XX` Status Code This status code, instead, means that wrong inputs have been correctly detected and the request caused a graceful error. This is the expected execution on malformed requests.

`5XX` Status Code An internal server error is exposed thanks to malformed input data. This is clearly a defect.

A JUnit test case is emitted for those error scenarios that cause status codes `2XX` and `5XX`.

Response Validation Oracle. Similarly to the Nominal Tester, also the Error Tester includes an oracle that checks if the response is valid with respect to the format defined in the specification.

7. EXPERIMENTAL VALIDATION

In this section, we provide an experimental validation of RESTTESTGEN, with respect to its effectiveness in revealing programming defects in real-world REST APIs. We also conducted an empirical evaluation on the contribution of the ODG in effectively generating test cases. The complete package to replicate our experiment is available online[6] .

7.1. Research Questions

To define our empirical investigation, we formulated the following research questions.

- RQ_N: Is the *Nominal Tester* module effective in generating test cases?
- RQ_E: Is the *Error Tester* module effective in generating test cases?
- RQ_O: Is the *Operation Dependency Graph* relevant for generating test cases?

The first research question RQ_N is intended to investigate if the *Nominal Tester* module is capable of testing the nominal scenarios of REST APIs, as they are documented in the OpenAPI specification.

The second research question RQ_E instead focuses on the *Error Tester* module, and it is meant to investigate if it is able to expose failures of the APIs in handling incorrect inputs, resulting from mutations of nominal test cases.

Finally, with the third research question RQ_O we aim to asses how much the new proposed operation testing order based on the *Operation Dependency Graph* is contributing to the testing capability of RESTTESTGEN.

[6]The replication package will be published on paper acceptance.

DOI: 10.1002/stvr

Experiments. The experimental validation has been conducted with two distinct sets of case studies. The first set of case studies (already presented in the companion paper [44] and listed in Table X) includes a remarkable number of REST APIs (87), whose access is quite simple, because it does not require authentication, i.e., they are *free access APIs*. This first validation has served as initial benchmark for our experimental framework and for RESTTESTGEN, to assess its effectiveness on real-world services, but with limited complexity. More sensitive operations, such as changing or deleting resources, usually require a user to be authenticated.

Based on the experience collected on this first experiment, we identified some changes to improve the experimental settings and those features that were urgently needed to extend our tool implementation. In order to test more realistic and challenging REST APIs, that expose a more complete set of features (e.g., deleting resources), we needed to extended our implementation and include features typically required by actual industry-class services. Therefore, we improved RESTTESTGEN by adding support for authentication. Additionally, the improved version of the tool offers also a more flexible management of test case generation budget based on time, for a more accurate experimentation.

The second set of case studies includes private and *access controlled APIs*. They are authenticated and closed-access services for a total of 29 novel services. They include industry-class and complex REST APIs like Google Drive, Google Calendar, YouTube and Spotify.

Unfortunately, we could not run the latest improved version of RESTTESTGEN on the first data set, because some of these APIs are now discontinued or no longer accessible. Thus, we will present the validation on the two data sets in separate sections, and discuss the results aggregately in Section 8.

Finally, we have conducted a third experimentation to asses the impact of the ODG on the testing capability of RESTTESTGEN. To this aim, we have chosen services with increasing levels of complexity. They are: (i) small services with no dependencies; (ii) small services with some dependencies; (iii) medium services with few dependencies; and (iv) a big service with lots of dependencies.

7.2. Experiment on Free Access APIs

Case Studies. We applied RESTTESTGEN to an extensive group of REST APIs, taken from the website *APIs.guru*[7] on June 18th, 2019. For each REST API, this website also provides the corresponding OpenAPI specification. However, we had to apply some sanity checks and filtering, to select case studies that are appropriate for assessing fully automated test case generation.

First of all, we probed the REST APIs to filter out those that were not responding, which were probably discontinued or just temporarily down. Then, we also excluded all those that declared to require authentication, because still not supported by RESTTESTGEN when this first data set was considered. Eventually, we manually sent some probe requests to the remaining services to dynamically verify them, before finalizing the case studies of our experiment. We had to further exclude some REST APIs because, despite their specification did not mention authentication, their actual implementation did require it.

After filtering, the final case studies used in our empirical evaluation consists of 87 REST APIs, for a total of 2,612 operations, which means 30 operations per REST API on average.

Experimental Procedure. Based on the research questions formulated above, we defined these settings of our experiment. RESTTESTGEN is run on all the OpenAPI specification files for the 87 case studies. For each specification, the ODG is computed and then the Nominal Tester module is run. In this first experiment, the operation budget is the total amount of attempts n_{fuzz} to fuzz an operation (see Section 5.1), which is set to 5. The edit distance threshold thr used in the Response Dictionary (see Section 5.2) is set to 1. The Nominal Tester was assigned a global testing budget of 30 minutes per case study. In the result section, we will see that this time budget will prove to

[7] https://apis.guru/browse-apis/

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table I. Free access APIs: case studies tested by the Nominal Tester.

Total APIs	87
APIs with status code `2XX`	62
APIs with status code `5XX`	20
APIs with validation error	66

Table II. Free access APIs: operations tested by the Nominal Tester.

Total operations	2,612
Tested operations	2,560
Operations with status code `2XX`	625
Operations with status code `5XX`	151
Operations with validation errors	1,733

be appropriate, in fact our algorithm completes in less than 10 minutes in the majority of the case studies. During this time budget, the maximum number of sent requests has been 13,944, with a mean of 162 requests per case study. However, the amount of time taken to test a service is strongly dependent on how long the tested service takes to respond.

The nominal test cases that obtained a response status code `2XX` were then mutated by the Error Tester module, to automatically generate error test cases. This second module was also assigned a maximum time budget of 30 minutes per case study.

RQ_N (Nominal Tester). The results of the Nominal Tester module on the *free access APIs* are shown in Table I. As we can see, all the 87 case studies (first line) have been subject to automated test case generation and for 62 of them (second line) at least one test for a nominal execution scenario (status code `2XX`) could be automatically generated. For 20 case studies, the test cases automatically generated by RESTTESTGEN exposed errors that were not properly handled by the REST API (status code `5XX`). On 66 case studies, invalid response messages were observed; they are responses inconsistent with their schema defined in the OpenAPI specification.

Table II shows a more detailed perspective, focusing on case studies operations. Among the total 2,612 operations, for 2,560 of them test cases could be generated by RESTTESTGEN. The untested operations are due to some failure of the tool, such as unsupported input parameter generation (e.g., files to be uploaded).

In particular, automatically generated test cases found a nominal execution for 625 of them (status code `2XX`) and an ungraceful error for 151 (status code `5XX`).

For 435 test cases the status code was `4XX`, but they are not shown in the table, because they were hard to classify with a black-box access. In fact, they might be graceful errors due to programming defects, or just rejected requests due to failures of RESTTESTGEN in generating appropriate inputs.

Test cases with validation errors are still a majority: in 1,733 tests the response did not match the declared schema. Considering these results, we can formulate the following answer to RQ_N:

The Nominal Tester module of RESTTESTGEN *is effective in automatically generating black-box test cases, because it was able to test 2,560 operations out of 2,612 on free access REST APIs. These tests exposed 151 faults in the form of not correctly handled internal errors and 1,733 inconsistent response messages.*

RQ_E (Error Tester). Subsequently, the 625 nominal test cases with a status code `2XX` have been subject to mutation by the Error Tester module, and executed on the *free access APIs*. The results of this second module are shown in Table III. For each mutation operator (first column) the table reports in the second column how many mutants (i.e., mutated test cases) could be generated.

The number of mutants is dependent by the mutation operator adopted, because different mutations impose different applicability preconditions. For instance, *Missing required* mutation needs an input field with the *required* modifier. In case this modifier in not present, the mutation does not apply. The largest amount of mutants (i.e., 707 tests) could be generated by *Wrong input type*, because it is the mutation with the simpler preconditions, i.e., a field of type string, numeric or enum. Then, *Missing required* mutation generated 459 mutants and, while *Constraint violation* mutation generated only 119 mutants, because only few case studies specify value constraints in their specification. In total, 1,285 mutants have been generated.

DOI: 10.1002/stvr

Table III. Free access APIs: test cases automatically generated by the Error Tester.

Mutation operator	Mutants	Status code 2XX	Status code 5XX
Missing required	459	283	7
Wrong input type	707	511	16
Constraint violation	119	68	11
Total	**1,285**	**864**	**23**

These mutated test cases are then executed on the case studies and the response status codes are evaluated to assert the presence of programming defects. Table III reports how many mutants are still processed as valid input with status code 2XX (third column) and how many of them exposed an unhandled error with status code 5XX (fourth column). The majority of defects (864 cases) have been recorded for status code 2XX; they are wrong data that are still handled as correct. Only 23 cases exposed server errors with status codes 5XX. It should be noted, that these 23 cases of status code 5XX are different and additional with respect to the 151 5XX cases observed in the previous testing phase. In fact, the test cases generated by the Nominal Tester and by the Error Tester are not overlapping, i.e., the former module relies on inputs whose type and value match the specification, while the latter module relies on inputs that violate the specification.

Given these data, we can answer to RQ_E as follows:

The Error Tester module of RESTTESTGEN *is effective in automatically generating black-box error test cases on free access APIs, because they revealed 864 cases where wrong data are accepted as valid, and 23 cases with unhandled errors.*

7.3. Experiment on Access Controlled APIs

The experiment on free access APIs gave us a clear idea of the capabilities of RESTTESTGEN, and it offered the opportunity to revise the experimental settings, in order to further improve the tool. In particular, industrial-ready APIs can often be accessed only with authorized credentials, especially when irreversible operations can be performed on persistent data (e.g., data change or deletion). Thus, a second experiment has been conducted, involving a novel set of case studies.

Case Studies. While case studies of the first experiment were limited to non-authenticated and public REST APIs, now also private and authenticated services are taken into account.

For the second set of cases studies, we relied on API directories to identify publicly hosted services, not only limited to *APIs.guru* as in the first experiment, but including also those listed in *RapidAPI*[8] and *ProgrammableWeb*[9]. Additionally, we searched for REST APIs published as open source projects on *GitHub*[10].

The criteria applied to select publicly hosted case studies are similar to those already used to select free access APIs, i.e., they should be responsive and coming with their OpenAPI interface definition. However, based on our experience with free access APIs, we also enforced new constraints for case studies. When selecting access controlled APIs, we discarded those APIs with a quota that was too restrictive. For example, an API with a 100 requests per day limit, or even 1000 per month, would be impractical to test with a fuzzing-based approach. Furthermore, in contrast to the previous experimental setting, now services that require authentication are welcome. According to the revised selection criteria, we could experiment with actual well-known and largely adopted services (e.g., provided by *Google* and *Spotify*). The exhaustive list of case studies is reported in the Appendix.

Publicly hosted access controlled APIs selected in this way consists of 20 REST APIs (listed in Table XI), for a total of 198 endpoints and 285 operations.

[8]https://rapidapi.com/
[9]https://www.programmableweb.com/category/all/apis
[10]https://github.com/

DOI: 10.1002/stvr

Table IV. Access controlled APIs: case studies tested by the Nominal Tester.

Total APIs	29
APIs with status code `2XX`	27
APIs with status code `5XX`	8
APIs with validation error	22

Table V. Access controlled APIs: operations tested by the Nominal Tester.

Total operations	388
Tested operations	388.0
Operations with status code `2XX`	165.9
Operations with status code `5XX`	13.8
Operations with validation errors	308.9

To overcome the problem represented by the quota limitation in publicly hosted APIs, we also searched for open source implementation of REST APIs among the projects shared in GitHub. In fact, running an API locally on our machine would allow to overcome any quota limitation problem. Additionally, testing a local non-production installation would turn testing less problematic. In fact, automated testing could reveal and exploit implementation defects and damage production APIs. With local services, we not only mitigate the risk of testing side effects, but we also have the possibility to restore the original database in case of destructive events, or inspect the API source code for a better comprehension of a candidate defect.

We used the search strings "REST", "RESTful API", "OpenAPI" and "Swagger" to search for case studies on GitHub. We only filtered out those projects that did not provide the OpenAPI specification (neither as part of source code, nor through libraries such as *swagger-ui*, allowing to query the specification from the running service). Then, we removed projects with compilation and runtime errors.

In GitHub, we found 9 REST APIs satisfying these criteria, for a total of 49 endpoints and 103 operations (listed in Table XII). These projects have been organized in separated *Docker* containers to isolate them and to have an effective way of restoring a common starting point after each test iteration.

All in all, considering publicly exposed APIs and open source APIs projects from GitHub, the access controlled APIs count 29 REST APIs in total, with 247 endpoints and 388 operations.

Experimental Procedure. Based on the experience gained with free access APIs, we adopted a refined experimental setting for the new set of case studies. First, instead of setting an operation budget based on the number of attempts, we opted for a time budget. In fact, we realized that it would be much more natural for a test engineer to specify the resources to spend in testing as an objective time duration, rather than a cryptic parameter that is internal to the tool.

Specifically, we let the Nominal Tester run for $t_{fuzz} = 5$ seconds for each operation documented in the specification of the REST API under test. Setting the operation budget to t_{fuzz} instead of n_{fuzz} comes with an additional major advantage: a more appropriate global testing budget. In fact, the time allocated to test case generation depends on the number of operations described in the specification. Larger services will be granted more testing time rather than smaller services.

Another major difference in the experimental procedure consists in replicating the test generation process multiple times. In fact, both the Nominal Tester and the Error Tester rely on non-deterministic algorithms, e.g., for the generation of input values. To contain side effects due to non-determinism, the testing process is repeated 10 times for each case study, and the average values are reported when aggregating the results.

RQ_N (Nominal Tester). The results of the experiment on *access controlled APIs* with the Nominal Tester are shown in Table IV. The Nominal Tester could test at least one successful nominal execution scenario (`2XX` status code) in 27 of the new 29 case studies. RESTTESTGEN was capable to expose errors (`5XX` status code) that were not properly handled by 8 of the tested case studies. In 22 case studies we observed invalid response messages, whose schema does not match the one described in the OpenAPI specification.

DOI: 10.1002/stvr

Table VI. Access controlled APIs: test cases automatically generated by the Error Tester.

Mutation operator	Mutants	Status code 2XX	Status code 5XX
Missing required	95.5	10.2	3.0
Wrong input type	428.6	33.6	3.7
Constraint violation	84.1	1.9	0.0
Total	**608.2**	**45.7**	**6.7**

Table V shows the results from the perspective of the operations. RESTTESTGEN could test all the 388 operations described in the specifications of the case studies. For 166 operations, the generated test cases obtained at least one success status code (2XX); for 14 operations, instead, test cases exposed an ungraceful error (5XX status code). The status code 4XX was obtained in 301 operations (not shown in the table because hard to classify).

Operations with validation errors are once again the majority: in 309 operations, responses schema did not match the one declared in the specification.

Considering these results, we can formulate the following answer to RQ_N:

The Nominal Tester module of RESTTESTGEN *is effective in automatically generating black-box test cases, because it was able to test all the 388 operations described in the specification of 29 access controlled REST APIs. These tests exposed 14 faults in the form of not correctly handled internal errors and 309 inconsistent response messages.*

RQ_E (Error Tester). Operations that responded to nominal test cases with at least one status code 2XX have been subject to mutation by the Error Tester, and executed on the *access controlled APIs*. Received status codes are evaluated by our oracle to assess the behavior of the REST API in presence of malformed inputs. Table VI shows the results of this experiment.

The Error Tester module generated a total of 608 mutants: 96 missing required, 429 wrong input type and 84 constraint violation mutants. The target APIs responded 46 times with 2XX status codes, meaning that the mutated input was interpreted as valid. For other 7 test cases we obtained the 5XX status code, meaning that the malformed input caused an error in the server elaborating the request.

Given these data, we can answer to RQ_E as follows:

The Error Tester module of RESTTESTGEN *is effective in automatically generating black-box error test cases on access controlled REST APIs, because they revealed 46 cases where wrong data are accepted as valid, and 7 cases with unhandled errors.*

7.4. Experiment on the Contribution of the ODG

To determine the actual contribution of the *Operation Dependency Graph* in automatically generating test cases, we conducted a third experiment with the aim of comparing the capability of RESTTESTGEN when adopting different operation ordering approaches. In particular, we considered two variants of RESTTESTGEN: (i) with the smart and dynamic order based on the ODG; and (ii) with a random order of the operations to test. By comparing the test cases emitted by the two variants, we will be able to quantify the contribution of the ODG to the performance of RESTTESTGEN with respect to a baseline random approach.

Case Studies. To perform a fair comparison between the two different operation ordering approaches, we must provide identical working conditions to each variant of the tool. In practice, we must provide to RESTTESTGEN, for each case study, the same OpenAPI specification and the same initial state for the REST APIs under test. Moreover, the REST APIs under test should not be accessed by external users, to avoid potential interference with the experiment.

To satisfy these requirements, we selected case studies whose state could be directly controlled and restored before each test iteration. They are four open source REST APIs we downloaded from

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table VII. Characteristics of the case studies used in the experiment on the ODG contribution.

Case study name	Operations	Dependencies
CS-GoogleDrive	46	5855
CS-RealWorld	19	6
CS-CRUD	4	0
CS-OrderAPI	3	0
CS-Users	5	23

GitHub (taken from the *access controlled REST APIs* dataset of Section 7.3) and ran locally, and the Google Drive REST API (as provided by Google). Restoring a common initial state for the locally installed APIs is trivial: we just need to reset the whole service and its database. Concerning Google Drive, which is running outside our controlled environment, the solution was to use a private folder that is accessible only by us. This folder will be initialized with the same resources (uploaded files, folders, shared documents, etc.) before each test iteration.

Instead of experimenting with all the open source locally installed REST APIs used in the previous experiment, in this comparison experiment we reduced the number of these APIs to better fit the experiment intention. In fact, we noticed that some REST APIs were of limited interest for the comparison, because their ODG had very few edges due to incomplete or inconsistent OpenAPI specifications. We prefer to perform this experiment with case studies with increasing number of dependencies, and understand how this would affect the behavior of the ODG.

In Table VII are reported the characteristics of the selected case studies. They are representative of four kinds of REST APIs: CS-CRUD and CS-OrderAPI are small REST APIs (4 and 3 operations) with no dependencies; CS-Users is a small REST API (5 operations) with some dependencies (23 edges in the ODG); CS-RealWorld is a medium-sized REST API (19 operations) with a few dependencies (6 edges in the graph); and CS-GoogleDrive is a big REST API (46 operations) with many dependencies (the ODG has almost 6000 edges).

Experimental Procedure. The goal of the experiment is to compare the baseline random ordering strategy against the smart and dynamic order provided by the ODG, to understand what is the actual role of the ODG. First, we run RESTTESTGEN on each case study with the ODG strategy and with the parameter $t_{fuzz} = 5$. Then, we repeated the test case generation sessions adopting the random ordering strategy, with the same global time budget.

The experiment has been repeated 10 times to control fluctuations in results due to the non-deterministic components of RESTTESTGEN.

RQ_O (Operation Dependency Graph). The results collected after running the two variants of RESTTESTGEN allow a direct comparison of their performance. We applied the Wilcoxon test to check whether the differences between two variants of RESTTESTGEN are statistically significant. We assume statistical significance when the statistical test returns a p-value < 0.05. To quantify the magnitude of differences among the two strategies, we used the Cliff's δ effect size [19]. The effect size was computed in R using the `effsize` package [42]. For independent samples, Cliff's δ provides an indication of the extent to which two data sets overlap. Cliff's δ ranges in the real interval $[-1, 1]$: it is equal to 1 when all values of one group are higher than the values of the other group and -1 when none of them is higher. Two overlapping distributions would have a Cliff's δ equal to 0. The effect size is considered small for $0.148 \leq \delta < 0.330$, medium for $0.330 \leq \delta < 0.474$, and large for $\delta \leq 0.474$. [11].

Results are reported in Table VIII and Figure 3 for the Nominal Tester module, and in Table IX and in Figure 4 for the Error Tester module. The two tables consist, in turn, of several sub-tables, each one focusing on a distinct metric. Different case studies are reported in different rows. For each case study the tables report: (i) the mean value of the ODG-based approach for the particular metric; (ii) the mean value of the random (Rand) approach for the particular metric; (iii) the p-value of the

DOI: 10.1002/stvr

Table VIII. Wilcoxon Test for the Nominal Tester module.

Operations with status code `2XX`

	ODG	Rand	p-value	δ eff. size
CS-GoogleDrive	10.4	7.6	**0.001**	0.91 (L)
CS-RealWorld	6.0	9.5	< **0.001**	-0.90 (L)
CS-CRUD	2.0	2.0	-	-
CS-OrderAPI	0.3	0.5	0.398	-
CS-Users	4.8	3.5	**0.003**	0.69 (L)

Operations with status code `5XX`

	ODG	Rand	p-value	δ eff. size
CS-GoogleDrive	0.9	0.9	1.000	-
CS-RealWorld	1.3	1.0	0.273	-
CS-CRUD	0.0	0.0	-	-
CS-OrderAPI	0.0	0.0	-	-
CS-Users	0.1	0.8	**0.003**	-0.70 (L)

Operations with status code `4XX`

	ODG	Rand	p-value	δ eff. size
CS-GoogleDrive	44.7	44.5	0.278	-
CS-RealWorld	11.0	10.2	0.229	-
CS-CRUD	3.0	3.0	-	-
CS-OrderAPI	3.0	3.0	-	-
CS-Users	0.3	2.2	**0.001**	-0.85 (L)

Operations with validation errors

	ODG	Rand	p-value	δ eff. size
CS-GoogleDrive	44.9	45.0	0.368	-
CS-RealWorld	17.0	17.0	-	-
CS-CRUD	0.0	0.0	-	-
CS-OrderAPI	3.0	3.0	-	-
CS-Users	0.3	2.3	**0.001**	-0.85 (L)

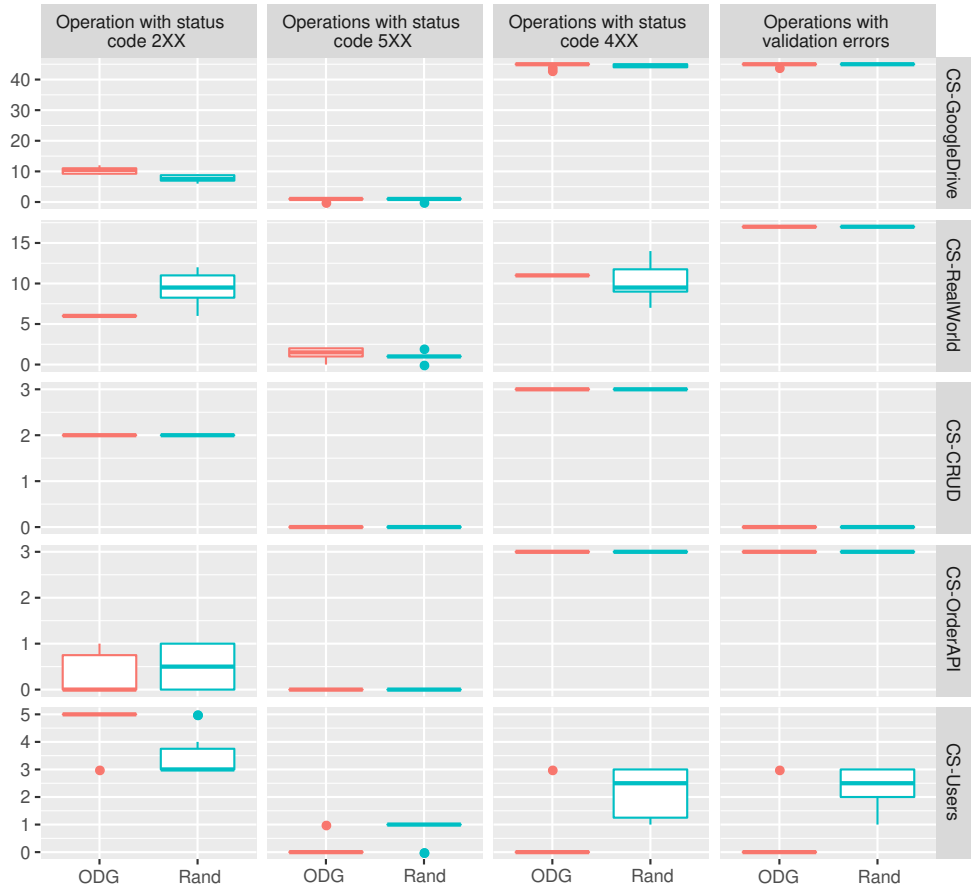

Figure 3. Boxplots for the Nominal Tester comparison between ODG and random (Rand) approaches.

Wilcoxon test (significant cases are highlighted in boldface, i.e., when p-value < 0.05); and, (iv) the δ effect size, reported only for statistically significant cases. In some cases, the p-value could not be computed, so it has been omitted. We also report the δ eff. size interpretation: S for small, M for medium, and L for large. For instance, in the first table concerning the metric *Operations with status code `2XX`* we see, in the first row, that (on average) 10.4 operations of CS-GoogleDrive could be

DOI: 10.1002/stvr

Table IX. Wilcoxon Test for the Error Tester module.

Missing required - Status code `2XX`

	ODG	Rand	p-value	δ eff. size
CS-GoogleDrive	0.0	0.0	-	-
CS-RealWorld	0.0	0.0	-	-
CS-CRUD	0.0	0.0	-	-
CS-OrderAPI	0.0	0.0	-	-
CS-Users	0.0	0.0	-	-

Missing required - Status code `5XX`

	ODG	Rand	p-value	δ eff. size
CS-GoogleDrive	0.0	0.0	-	-
CS-RealWorld	0.0	0.0	-	-
CS-CRUD	0.0	0.0	-	-
CS-OrderAPI	0.0	0.0	-	-
CS-Users	1.9	1.3	**0.009**	0.60 (L)

Wrong input type - Status code `2XX`

	ODG	Rand	p-value	δ eff. size
CS-GoogleDrive	0.0	0.0	-	-
CS-RealWorld	1.9	1.9	0.900	-
CS-CRUD	0.2	0.4	0.366	-
CS-OrderAPI	0.0	0.0	-	-
CS-Users	0.0	0.0	-	-

Wrong input type - Status code `5XX`

	ODG	Rand	p-value	δ eff. size
CS-GoogleDrive	0.0	0.0	-	-
CS-RealWorld	0.0	0.0	-	-
CS-CRUD	0.0	0.0	-	-
CS-OrderAPI	0.0	0.0	-	-
CS-Users	1.9	1.3	**0.009**	0.60 (L)

Constraint violation - Status code `2XX`

	ODG	Rand	p-value	δ eff. size
CS-GoogleDrive	0.0	0.0	-	-
CS-RealWorld	0.0	0.0	-	-
CS-CRUD	0.0	0.0	-	-
CS-OrderAPI	0.0	0.0	-	-
CS-Users	0.0	0.0	-	-

Constraint violation - Status code `5XX`

	ODG	Rand	p-value	δ eff. size
CS-GoogleDrive	0.0	0.0	-	-
CS-RealWorld	0.0	0.0	-	-
CS-CRUD	0.0	0.0	-	-
CS-OrderAPI	0.0	0.0	-	-
CS-Users	0.0	0.0	-	-

tested with the ODG-based strategy and 7.6 operations with the random strategy. We see that this difference is statistically significant, because the p-value is 0.001, and the effect size is large (0.91).

The two figures, instead, show the boxplots for the same metrics, and allow us to visually compare the results of the two variants of RESTTESTGEN. In each boxplot, the ODG-based strategy is shown on the left-hand side in orange, while the random strategy is on the right-hand side in green. Boxplots are organized in rows and columns: rows for case studies, and columns for metrics. For instance, the first boxplot of Figure 3 shows the distribution of *Operations with status code `2XX`* in CS-GoogleDrive. In this graph we can notice that the ODG-based strategy can test a larger number of operations than the random strategy.

Looking at the results of the Nominal Tester module, we observe that the contribution of the ODG is negligible when the corresponding graph encodes no dependency (CS-CRUD and CS-OrderAPI). In fact, in absence of data dependencies, the ODG-based strategy only relies on the CRUD semantics to decide the testing order. Conversely, when many dependencies are available in the graph (CS-Users and CS-GoogleDrive), the contribution of the ODG over the random approach is more remarked. For instance, in CS-GoogleDrive we observe that, according to the Wilcoxon Test, the number of operations successfully tested with the ODG with `2XX` status code is statistically significantly larger than random (p-value < 0.05), with a large effect size ($\delta = 0.91$).

In the case study CS-Users, along with an increase in the number of successfully tested operations (similarly to CS-GoogleDrive), we also observe a statistically significant decrease in the number of operations that obtained a `4XX` status code, meaning that the ODG helped the Nominal Tester in avoiding client-side failures.

In CS-Users, we also observe a significant decrease in the number of tests that caused an internal server error (status code `5XX`). This might be due to the order provided by the ODG, which allowed the Nominal Tester to retrieve, store and reuse (though the response dictionary) mostly valid input values and, thus, emit successful nominal test cases. Conversely, when the random ordering is used, the dictionary might not contain the needed values which can only be guessed randomly, with a higher chance of input values that are incorrect according to the REST API business logic, that might have caused server crashes.

DOI: 10.1002/stvr

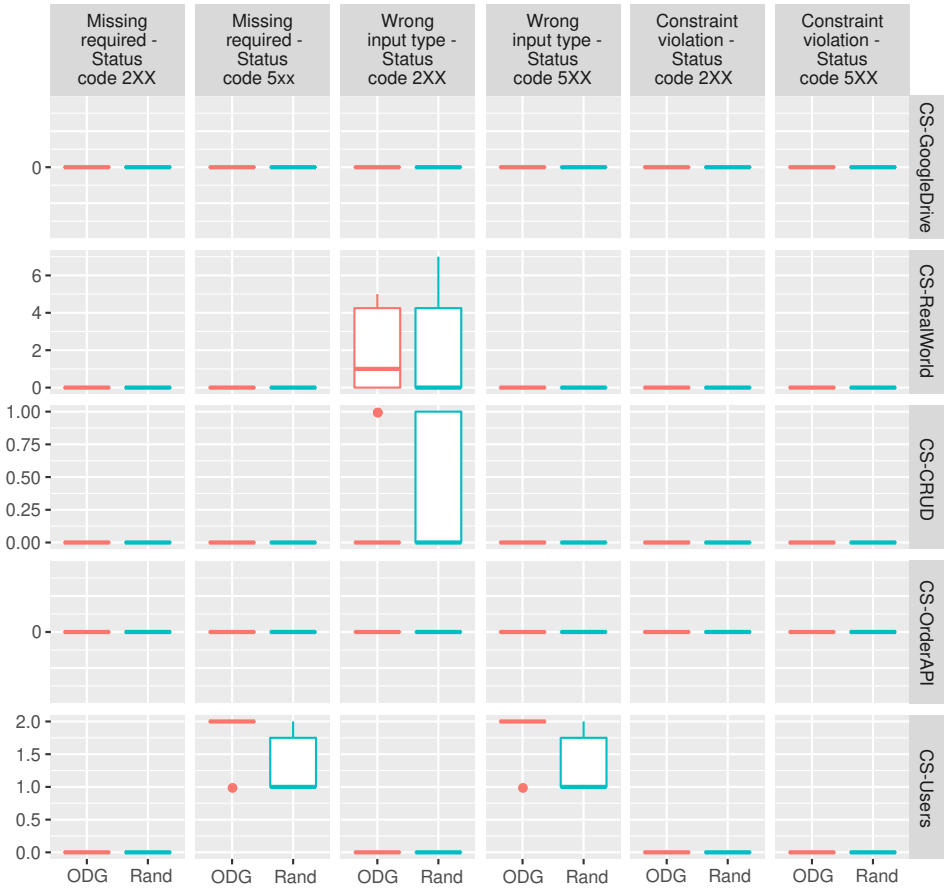

Figure 4. Boxplots for the Error Tester comparison between ODG and random (Rand) approaches.

Finally, we observe that the number of operations with validation errors is higher for the random approach. This is caused by validation errors occurring in all the `4XX` responses, which contain the log of a server-side exception, instead of an empty body as described in the OpenAPI specification. Thus, for this particular case study, the number of operations with validation errors is correlated to the number of operations that obtained a `4XX` status code.

We can observe that in the case study CS-RealWorld, the ODG-based approach could test less operations with status code `2XX` than the random approach. Since RESTTESTGEN could only extract six dependencies from the OpenAPI specification of CS-RealWorld, most of the nodes in the graph are not connected to each other. Consequently, rather than by the actual data dependencies, the operation testing order is deterministically decided by the CRUD semantics. When the right dependencies can not be inferred, a randomized ordering is on average preferable.

Let's consider now the results of the Error Tester module in Figure 4 and Table IX. Here the strategy based on the ODG seems to overcome the random strategy. Let's recall there is a strong link between Nominal Tester and the Error Tester modules. In fact, the error test cases are created by directly mutating those test cases that are successfully elaborated by the Nominal Tester module. The number of error test cases is therefore directly proportional to the number of successful nominal test cases. In Figure 4 we see that in the case study CS-Users, the variant of RESTTESTGEN adopting the ODG-based ordering strategy could detect more `5XX` server-side errors than the random strategy. This improvement descends from the larger number of successfully tested operations by the Nominal Tester in the same case study. Statistically significant difference can not be reached in other case studies, probably because their code is of higher quality, so it is harder to expose server crashes by either strategy.

DOI: 10.1002/stvr

Considering these results, we can formulate the following answer to RQ_O:

The Operation Dependency Graph conveys a remarkable contribution to the generation of black-box test cases, in particular for those REST APIs with many dependencies among operations. In fact, the Nominal Tester using the ordering strategy based on the ODG could test a larger number of operations that returned a successful status code (`2XX`) and a lower number of operations that returned a client-side error status code (`4XX`) than the random ordering strategy. Also the Error Tester could reveal a larger number of errors when using the ODG-based ordering strategy. These trends have a large effects size. Instead, the contribution of the ODG-based strategy is negligible or even detrimental when no data dependencies are available in the graph.

7.5. Threats to Validity

There is a number of limitations that could potentially threat the validity of our empirical results.

Black-box access. With the aim of considering realistic case studies, we meant to involve real and existing REST APIs, hosted by their respective owners. As such, only for a fraction of the case studies (i.e., the open source projects from GitHub) we could inspect the case study source code. For the majority of the case studies we could not inspect source code to manually validate the results of automated testing, with respect to the correct classification reported by our oracles. Moreover, we could not measure the code coverage achieved by automatically generated tests. However, for the second oracle, the classification was quite objective, because it revealed a defect whenever the response was inconsistent with the documented schema.

Oracle based on status code. Using the status code to assess the result of automated testing might represent a threat. In fact, while the classification of tests with correct executions (status code `2XX`) and with *unhandled* errors (status code `5XX`) is more objective, the classification of tests with *handled* errors (status code `4XX`) is more dubious. In fact, handled errors might occur either because of defects in the REST API implementation, or because of limitations of our tool that caused wrong input values to be used in test cases. Since we could not accurately classify executions with `4XX` status code, we conservatively assumed them to be due to failures of RESTTESTGEN.

Most of the defects detected by the *Error Tester* are incorrect data accepted as valid (status code `2XX`). This result highlights a potential limitation of the *Status Code Oracle* used in the *Nominal Tester*. This oracle might have incorrectly classified a test case run as a *pass* just because of its status code `2XX`, that was also observed in case of invalid input values.

Missing authentication. To increase the number of REST APIs in our experiment, we decided to minimize the manual effort required to prepare each case study. Since account creation would have required a substantial effort and, sometimes, interaction with the REST API owner, e.g., to motivate why API access is required, we tested some case studies that did not require account creation. This might represent a threat to the validity of our results, because sensitive operations might be forbidden to anonymous users (e.g., delete a resource) and the experiment might be limited to less crucial operations that, thus, are considered of lower importance by a case study owner. Nonetheless, the second set of case studies includes access controlled APIs that do require authentication to be tested, thus mitigating this threat.

Nondeterminism. Our algorithm contains non-deterministic components (e.g., in the fuzzer of input parameters). For the second set of case studies, we adopted a more accurate experimental setting and we included multiple executions to control and measure non-determinism. However, a more extensive and complete experimentation is required, with more case studies, to corroborate our findings.

DOI: 10.1002/stvr

8. DISCUSSION ON THE VALIDATION RESULTS

In this section, we discuss the experimental results obtained during the validation of RESTTESTGEN, distilling some general considerations about the tool and highlighting the weaknesses we have found in the tested REST APIs. Furthermore, we detail some particular faults detected in largely adopted APIs (e.g., *YouTube* and *Google Drive*).

8.1. Examples of Revealed Faults.

We present now some actual faults detected by RESTTESTGEN while conducting our empirical validation. All the discovered faults, including the ones not presented in this section, have been reported the respective owners[11].

Internal server error. RESTTESTGEN was able to automatically generate test cases that caused `5XX` status codes in several APIs, including *YouTube* and *Google Drive*. The operations in the API of *YouTube* related to live chats allow to specify which particular chat to operate, with the input parameter `liveChatId`. When this parameter contains a value that is not a valid chat ID, we would expect to receive a response with a `404` status code. Instead, when using invalid IDs, our test cases obtained a `500` Internal Error. This is, probably, caused by a missing validation of the chat ID. So, the server code throws an exception when using an incorrect value of chat ID. Listing 5 shows an example of the JSON body for an HTTP response from *YouTube* with a `500` status code.

```
{
  "error":
    {
      "code": 500,
      "message": "Internal␣error␣encountered.",
      "errors":
        [{
          "domain": "youtube.api.v3.LiveChatModeratorListResponse.Error",
          "reason": "LIVE_CHAT_NOT_FOUND"
        }],
      "status": "INTERNAL"
    }
}
```

Listing 5: The body of the YouTube live chat `500` response.

A similar fault can be observed in the *Google Drive* API, in the operation supposed to retrieve the list of revisions for a file. The revisions list can be obtained issuing the request `GET /files/{fileId}/revisions`, where the path parameter *fileId* should be replaced by the ID of the interesting file. When testing this endpoint, RESTTESTGEN could obtain responses with `500` status code, which suggests that some error occurred at server side. However, the body of such HTTP repose, shown in Listing 6, contains no useful information to guess the exact origin of this error, because no explanatory message can help debugging.

We hypothesize that the cause of the failure might be either an invalid *fileId* or the random values assigned to some parameters. Although the provided *fileId* was observed in the previous requests, it might be invalid in the system because of deletions or not yet allocated files. The *Google Drive* API, in fact, exposes an endpoint that generates brand new *fileIds* to be attached in future file creation requests and RESTTESTGEN might have used one of those. Moreover, some other parameters of the request, such as *quotaUser* and *pageToken*, could also be invalid in the system (although correctly generated by the Nominal Tester according to the OpenAPI specification) because guessing appropriate values from scratch is very unlikely.

[11] We are still waiting for official acknowledgment, but we are confident that we can reference defect' acknowledgments on the final version of this paper.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
{
  "error":
    {
      "code": 500,
      "message": null
    }
}
```

Listing 6: The body of the `500` response of the Google Drive list file revisions operation.

Another `500` internal error could be revealed by the Error Tester module of RESTTESTGEN in the *Deutsche Bahn (DB) Fahrplan* API. DB is a German railway company, the second-largest transport company in the world[12]. The Timetable API (called "Fahrplan" in the API) exposes an endpoint of the form `GET /location/{name}`. According to its OpenAPI specification, the `name` path parameter accepts a string value corresponding to the name of the location for which to query the timetables. In one of the malformed requests sent by the Error Tester module, this string field is replaced with an integer value, as shown below.

```
GET /fahrplan-plus/v1/location/-66
```

This makes the server respond with the `500` status code, suggesting some error in the server-side logic has occurred. The body of the response is the following.

```
{  "error":  {  "code": 500, "message": "Internal␣Server␣Error"  }  }
```

After a deeper investigation of the reports of RESTTESTGEN relative to this operation, including the results of nominal test cases, we noticed that the operation tries to answer with a list of the locations whose name better matches the input string. Probably, the server-side search algorithm raises an exception when integers are provided as input.

Inconsistent Response Content Type. The *Response Validation Oracle* integrated in RESTTESTGEN is meant to check that responses are consistent with the schema defined in the OpenAPI specification. *Content Type* is among the fields in the schema, and it is meant to specify how to parse data in the response. Typically, the content type is *application/json*, which means that the data in the response are formatted in JSON and they should be parsed using a JSON grammar, but many more formats are supported (e.g., XML). In the *YouTube* API, according to its schema, the operation `GET /membershipsLevels` should respond with data in JSON format. However, one of the test cases generated by RESTTESTGEN obtained a `200` status code, and the response contained JavaScript code instead of the expected JSON data, as shown in Listing 7. Probably, such response was triggered by the two parameters `alt=json` and `callback=bSrnJNf`, asking for a response in JSON format, inside a callback function named *bSrnJNf*. This defect may cause failures (e.g., crashes) on those clients that connect to this service, when they try to parse this JavaScript response with a JSON grammar.

Access Control. Authentication tokens can have different scopes depending on which resources they can access (to be decided when generating a token). Indeed, different tokens can be requested for the same user to access different resources. An API should enforce correct access control and allow access only to those resources that are authorized by a token.

The response shown in Listing 7 highlights also another defect than inconsistent response content type. Despite we used a valid authentication token to access the *YouTube* API, this operation would require a peculiar authorization scope (prior approval from YouTube) that was not actually granted to the token that we were using. In fact, other testing scenarios for this operation correctly obtained responses with `403` status code, i.e., *Unauthorized*. The case shown in Listing 7, instead, obtained

[12] https://en.wikipedia.org/wiki/Deutsche_Bahn

DOI: 10.1002/stvr

a `200` HTTP status code, which is incorrect because the request should have been unauthorized. A `403` code is only mentioned in the response body that is inconsistent with the observed status code. Our speculation is that the two defects are related, and the incorrect response type causes also an incorrect response status code.

```
// API callback
bSrnJNf(
  {
    "error":
      {
        "code": 403,
        "message": "Request had insufficient authentication scopes.",
        "status": "PERMISSION_DENIED"
      }
  }
);
```

Listing 7: The body of the `200` response from YouTube containing JavaScript.

Inconsistent Resource Creation. A last defect was exposed by RESTTESTGEN, even if not captured by its oracles, but manually noticed while inspecting API responses obtained while generating test cases. The *YouTube* video upload operation `POST /youtube/v3/videos` is supposed to have a media video as resource in the body of the request. However, file attachment is not supported by the Nominal Tester that, in fact, could never successfully complete video uploads, and always obtained a `400` status code.

Nevertheless, after the experiment with RESTTESTGEN was concluded, the YouTube Studio *Content* page shows a long list (hundreds) of uploaded videos, while we expected an empty list because, according to their status code, no upload was successful. Our conjecture of a faulty behavior is corroborated by the evidence that for each video there is no thumbnail, the title is *"unknown"* and the status is set as *"Processing will begin shortly"*. The same list is still visible many days after the testing session was completed.

8.2. General Considerations

Based on the experimental results, we could formulate these subjective considerations.

Urgent need for automated testing support. When assessing RESTTESTGEN, we discovered faults and defects that affected even services published by reputable software companies (e.g., Google), that are supposed to invest and spend substantial effort in software quality enforcement. This highlights how hard it is to avoid mistakes in the implementation of REST APIs, even when effort is spent on quality control. Probably, this is due to the peculiar nature of this kind of software, that is more abstract and intangible than more traditional software artifacts (e.g., smartphone apps or web applications), or just because mature automated testing support is available for traditional software but not for APIs, for instance to automatically generate test cases to help and reveal faults.

There is an urgent need for automated approaches and tool support to help developers in validating the implementation of REST APIs, to identify defects in the exposed behavior and to check that the actual service is consistent with the published interface. Deviations from the declared interface might, in fact, cause problems to the client software that relies on defective services, e.g., because of failure in parsing malformed responses.

Robustness on malformed input. In some cases, REST APIs try to be robust and accept even malformed input data coming with service requests. For instance, a permissive service might accept an alphanumeric value when a numeric value was expected, simply dropping alphabetic characters and then converting the string to a number. Alternatively, a service might accept a numeric value out of the allowed range, by truncating a too large value to the upper limit of the range. While this strategy might be considered a robust service implementation, it clearly deviates from the documented (OpenAPI) interface and it might permit incorrect clients to proliferate. Instead,

DOI: 10.1002/stvr

enforcing the correctness of input data, e.g., by throwing an error on malformed requests, would make defective clients fail immediately. Consequently, developers would be forced to revise and fix their defective clients.

Empty result instead of error. Query-oriented APIs often adopt a specific strategy for hiding problems related to input data in service requests. When an input field is malformed, e.g., out of the documented domain, instead of responding with an error code, the service simply discards the flawed field and replaces it with an empty string or with a *null* value. For instance, a music streaming API that receives the request for a malformed *playlist-id* might respond with an empty list of songs, rather than with an error message about the wrong request. For the service consumer software, a clear and informative error message would be preferable than a silent error. In fact, while both a silent and an evident error deliver no service to the end-user, at least the evident error is informative of the problem cause and might be helpful towards its solution (e.g., for filling a costumer support claim).

Security requirements. Considering the two oracles currently available in RESTTESTGEN, most of the faults revealed in the experimental investigation are related to unexpected status codes or to malformed responses. However, some scenarios highlighted the need to search for APIs defects also according to a security viewpoint. In fact, we noticed a potential defect related to access control policies, where an API seems to accept requests that missed the required permissions. We might have just scratched the surface of a much larger problem, that shall require further investigation. Security related requirements might deserve appropriate and customized approaches to be thoroughly tested, that go beyond a general purpose testing tool. It is part of our research agenda to investigate novel oracles and testing scenario generation approaches able to reveal security defects and vulnerabilities in REST APIs.

9. RELATED WORK

The specificity of REST APIs attracted the attention of the testing community only recently. From the industry side, we can find some commercial test authoring tools [7,14,21,33,36,38], which help developers to *manually* write tests that can be then automatically run by the tool. These approaches have a different goal from RESTTESTGEN, whose aim is to test REST APIs without manual efforts.

From a pure research standpoint, novel approaches to the (semi-)automatic tests generation for REST APIs have been proposed, that can be divided in black-box approaches and white-box approaches. A *white-box* perspective in automated testing relies on the availability of APIs source code to perform static analysis, or to instrument it to collect execution traces and metric values. A *black-box* approach, instead, does not require any source code, which is often the case when using closed source components and libraries. However, a black-box access to the REST API lacks much information potentially useful for the automatic test case generation.

Fuzzers [1, 15, 16, 40, 41] are black-box testing tools that generate new tests starting from previously recorded API traffic: they fuzz and replay new traffic in order to find bugs. Some of these also exploit the OpenAPI specification of the service under test [15, 16, 40]. Godefroid et al. [17] propose a methodology to fuzz body payloads intelligently using JSON body schemas (i.e., those defined in the OpenAPI specification) and advanced fuzzing rules, in order to extend fuzzing engines used in testing tools like RESTler [4]. This applies also to RESTTESTGEN, since it leverages a fuzzer in order to generate input values for API requests. In our approach, the fuzzer implements two strategies: response dictionary and random values. Extending the fuzzing engine with the methodology proposed by Godefroid et al. [17] will, in principle, increase the test coverage for our tool.

The most related work is, probably, by Ed-douibi et al. [12]. They propose a model-based approach for black-box automatic test case generation of REST APIs. A model is extracted from the OpenAPI specification of a REST API, to generate both nominal test cases (with input values that match the model) and faulty test cases (with input values that violate the model). However,

DOI: 10.1002/stvr

they do not explicitly model the dependencies among operations, while we define the *Operation Dependency Graph* to this aim. Moreover, we dynamically update this graph to decide the most appropriate operation for the next test. Additionally, we integrate the response dictionary in a series of heuristics to automate input data generation.

Another limitation of their approach is that it only applies on read-only operations, called *safe* operations by the authors, because they meant to avoid operations with side effects on the API state. Conversely, our approach explicitly models side effects of operations (i.e., the CRUD semantics, see Section 5.1) and exploits them to decide the order in which to test operations.

Similarly to our approach, also Atlidakis at al. [4] model the dependencies among the operations in a REST API to elaborate an appropriate ordering. However, while they use dependencies to pre-compute the order to test operations (e.g., using Breadth-first search or random walk), we propose to compute the next operations to test dynamically, based on the outcome of the operations that could be tested so far. Martin-Lopez et al. [30] present a domain-specific language (IDL) for the specification of dependencies among input parameters in web services. Then, they translate an IDL document into a constraint satisfaction problem (CSP), enabling the automated analysis of IDL specifications using standard CSP-based reasoning operations. Unfortunately, the design of the (IDL) specification requires a manual intervention, so it is not suitable for the integration with fully-automatic testing tools like our RESTTESTGEN.

Another similar approach is the work of Karlsson et al. [23], in which the authors propose a black-box tool (QuickREST) aiming at automatically *exploring the behavior* of a REST API. Alongside the exploration, their tool is able to test services for a wide scope of properties, not only faults detection. Indeed, their approach is property-based, and relies on the OpenAPI specification of the service under test. They use the status code as oracle, with a further check validating that payloads received actually conform to what is described the OpenAPI specification. This is basically what we do in RESTTESTGEN with the two oracles of the Nominal Tester module presented in Section 5. QuickREST also generates random parameter values and sequences of operations that leverages previously returned results to perform stateful operations, but they do not exploit dependencies among operations, as we do in RESTTESTGEN. Furthermore, their test inputs generation, even if guided by the specification, is completely random. In RESTTESTGEN, we also exploit a response dictionary strategy, in order to generate more accurate input values.

Segura et al. [37] propose a complete different black-box approach, where the oracle is based on metamorphic relations among requests and responses. For instance, they send two queries to the same REST API, where the second query has stricter conditions than the first one (e.g., by adding an additional constraint). The result of the second query should be a proper subset of entries in the result of the first query. When the result is not a sub-set, the oracle reveals a defect. However, this approach only works for search-oriented APIs. Moreover, this technique is only partially automatic, because the user is supposed to manually identify the metamorphic relation to exploit and what input parameters to test.

In addition to pure functional testing approaches, works in the field of security testing are starting to rise [5, 25, 26, 28], in order to find potential security vulnerabilities in REST APIs. In particular, Mai et al. [28] use metamorphic relations to address the oracle problem: 22 system-agnostic metamorphic relations are defined in order to automate security testing in Web systems. Atlidakis et al. [5] introduce four security rules that capture desirable properties of REST APIs. They also show how a REST API fuzzer can be extended with active property checkers that automatically test and detect violations of these rules. Luo et al. [25, 26] focus on access control. In their first work [25], their aim is to simplify the privilege partitioning problem into a classification problem of RESTful functions. They propose a REST API classification approach (RestSep) based on genetic algorithms. In their second work [26], they propose a policy language (RestPL) to express authorization policies for REST APIs. A RestPL policy can be automatically generated from an actual request, which helps mitigating users intervention. Security is surely an import aspect that testing tools should address. At the moment, RESTTESTGEN does not provide specific mechanisms to enforce security properties; we plan to extend the testing capabilities of our tool towards the security direction as a future work.

DOI: 10.1002/stvr

In the companion paper [44], we presented an earlier version of our approach to automatically test REST APIs with a black-box approach (e.g., authentication was not supported). In the present paper, we extend the previous tool implementation, we refine the experimental settings and we extend the empirical validation by including access controlled APIs.

White-box approaches are complementary to ours, because they assume to have access to the source code of the API to test. Arcuri [2] propose a fully automated white-box testing approach, to generate test cases with evolutionary algorithms. Similarly to ours, Arcuri's approach requires the API specification (i.e., the OpenAPI specification). Differently than us, his approach also requires access to the Java bytecode of the REST API to test. In fact, the objective is to achieve high code coverage. This approach has been implemented and available as a tool prototype called EvoMaster. In another work, Arcuri et al. [3] extend EvoMaster introducing a series of novel testability transformations aimed at providing guidance in the context of commonly used API calls.

Many approaches have been proposed so far to test Web-services, based on their WSLD specification [6, 22, 24, 27, 29, 32, 39, 43, 46]. An extensive survey of techniques for automated testing of Web-services has been conducted and reported by Bozkurt et al. [8] and by Canfora et al. [9, 10]. Despite similar objectives, Web-services and REST APIs are conceived on top of different interaction models. Web-services are mostly based on SOAP [20], a message oriented model (mainly meant to overcome limitations of previous solutions, such as CORBA, Java/RMI, DCOM), while REST APIs rely on the concept of *web resources* accessible through stateless operations [13].

For these reasons, white-box approaches and testing tools for Web-services are not directly comparable with our RESTTESTGEN.

Regarding test coverage, Martin-Lopez et al. [31] propose ten *coverage criteria* to assess the adequacy of REST APIs testing approaches in this context, to automatically measure and compare their effectiveness. They then arrange these criteria into eight test coverage levels (TCLs) of increasing strength. This enables the automated assessment and comparison of testing techniques according to the overall coverage and TCL achieved by their generated test suites. In the present work, we do not measure the coverage of our testing approach, we just assess the effectiveness of RESTTESTGEN on the empirical results given by the validation (Section 7). We plan to adopt the metrics introduced by Martin-Lopez et al. [31] as a future extension.

Finally, *regression testing* has been applied also in the context of REST APIs, mostly in commercial tools [33, 36, 38]. However, these tools do not generate tests automatically and they do not perform differential testing. Godefroid et al. [18], instead, use differential testing applied to REST APIs in order to compare various client-service configurations. Regression testing is, at the moment, out of scope of our tool RESTTESTGEN.

10. CONCLUSION

In this paper we present RESTTESTGEN, a novel black-box approach for the automatic generation of REST APIs test cases. To model data dependencies among the operations in a REST API we exploit the Operation Dependency Graph (ODG). This allows our approach to dynamically decide in which order to test operations, such that the input data required to test an operation are available from the output data of those already tested.

The tool is composed by two distinct testing modules: the Nominal Tester, that automatically generates test cases related to nominal execution scenarios; and the Error Tester, that automatically generates test cases related to error management scenarios. Both modules rely on two oracles to assess if the test case generation is successful, one based on the obtained status code and the other based on the match with response schema declared in the specification (OpenAPI).

We validated our tool on more than one hundred of real-world REST APIs, supporting authentication. The empirical assessment showed that the proposed approach is effective in testing real-world services, and in detecting a considerable amount of implementation defects. Indeed, during the validation we were able to spot some remarkable bugs in services deployed by important software companies, such as Google.

DOI: 10.1002/stvr

We empirically evaluated the contribution of the ODG, which significantly improves the capabilities of RESTTESTGEN in testing REST APIs, especially those having a large number of data dependencies among operations.

As future work, we plan to extend the testing capability of RESTTESTGEN to try and assess the presence of security defects in the implementation of REST APIs, such as the API vulnerabilities pointed out by the OWASP Foundation [34]. This would require to attempt black-box proof-of-concept attacks and to define brand new oracles, capable to detecting successful attacks.

ACKNOWLEDGEMENT

This paper has been partially supported by project MIUR 2018-2022 “Dipartimenti di Eccellenza”.

REFERENCES

1. API Fuzzer. API Fuzzer. `https://github.com/KissPeter/APIFuzzer`.
2. A. Arcuri. RESTful API automated test case generation with Evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1):3, 2019.
3. A. Arcuri and J. P. Galeotti. Testability transformations for existing APIs. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, pages 153–163. IEEE, 2020.
4. V. Atlidakis, P. Godefroid, and M. Polishchuk. RESTler: Stateful REST API fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE ’19, pages 748–758, Piscataway, NJ, USA, 2019. IEEE Press.
5. V. Atlidakis, P. Godefroid, and M. Polishchuk. Checking security properties of cloud service REST APIs. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*, pages 387–397. IEEE, 2020.
6. X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. WSDL-based automatic test case generation for web services testing. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE ’05)*, pages 207–212. IEEE, 2005.
7. Borvid. HttpMaster. `http://www.httpmaster.net`.
8. M. Bozkurt, M. Harman, and Y. Hassoun. Testing web services: A survey. Technical report, Department of Computer Science, King’s College London, Tech. Rep. TR-10-01, 2011.
9. G. Canfora and M. Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17, 2006.
10. G. Canfora and M. Di Penta. Service-oriented architectures testing: A survey. In *Software Engineering*, pages 78–105. Springer, 2007.
11. J. Cohen. *Statistical power analysis for the behavioral sciences (2nd ed.)*. Lawrence Earlbaum Associates, Hillsdale, NJ, 1988.
12. H. Ed-Douibi, J. L. C. Izquierdo, and J. Cabot. Automatic generation of test cases for REST APIs: A specification-based approach. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 181–190. IEEE, 2018.
13. R. T. Fielding. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.
14. A. Fortress. API Fortress. `http://apifortress.com`.
15. Fuzz-Lightyear. Fuzz-Lightyear. `https://github.com/Yelp/fuzz-lightyear`.
16. Fuzzy-Swagger. Fuzzy-Swagger. `https://github.com/namuan/fuzzy-swagger`.
17. P. Godefroid, B. Huang, and M. Polishchuk. Intelligent REST API data fuzzing. In P. Devanbu, M. B. Cohen, and T. Zimmermann, editors, *ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 725–736. ACM, 2020.
18. P. Godefroid, D. Lehmann, and M. Polishchuk. Differential regression testing for REST APIs. In S. Khurshid and C. S. Pasareanu, editors, *ISSTA ’20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 312–323. ACM, 2020.
19. R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.
20. M. Hadley, N. Mendelsohn, J. Moreau, H. Nielsen, and M. Gudgin. SOAP version 1.2 part 1: Messaging framework. *W3C REC REC-soap12-part1-20030624, June*, pages 240–8491, 2003.
21. J. Haleby. REST Assured. `http://rest-assured.io/`.
22. S. Hanna and M. Munro. Fault-based web services testing. In *Fifth International Conference on Information Technology: New Generations (itng 2008)*, pages 471–476. IEEE, 2008.
23. S. Karlsson, A. Čaušević, and D. Sundmark. QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 131–141, 2020.
24. Y. Li, Z.-a. Sun, and J.-Y. Fang. Generating an automated test suite by variable strength combinatorial testing for web services. *Journal of computing and information technology*, 24(3):271–282, 2016.

DOI: 10.1002/stvr

25. Y. Luo, T. Puyang, X. Sun, Q. Shen, Y. Yang, A. Ruan, and Z. Wu. RestSep: Towards a test-oriented privilege partitioning approach for RESTful APIs. In I. Altintas and S. Chen, editors, *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*, pages 548–555. IEEE, 2017.
26. Y. Luo, H. Zhou, Q. Shen, A. Ruan, and Z. Wu. RestPL: Towards a request-oriented policy language for arbitrary RESTful APIs. In S. Reiff-Marganiec, editor, *IEEE International Conference on Web Services, ICWS 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 666–671. IEEE Computer Society, 2016.
27. C. Ma, C. Du, T. Zhang, F. Hu, and X. Cai. WSDL-based automated test data generation for web service. In *2008 International Conference on Computer Science and Software Engineering*, volume 2, pages 731–737. IEEE, 2008.
28. P. X. Mai, F. Pastore, A. Goknil, and L. Briand. Metamorphic security testing for web systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 186–197, 2020.
29. E. Martin, S. Basu, and T. Xie. Automated robustness testing of web services. In *Proceedings of the 4th International Workshop on SOA And Web Services Best Practices (SOAWS 2006)*, 2006.
30. A. Martin-Lopez, S. Segura, C. Muller, and A. Ruiz-Cortes. Specification and automated analysis of inter-parameter dependencies in web APIs. *IEEE Transactions on Services Computing*, pages 1–1, 2021.
31. A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés. Test coverage criteria for restful web apis. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2019, pages 15–21, New York, NY, USA, 2019. Association for Computing Machinery.
32. J. Offutt and W. Xu. Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.
33. Optimizory Technologies Pvt. Ltd. vREST. `https://vrest.io/`.
34. OWASP. API Security Top 10 2019. `https://owasp.org/www-project-api-security/`.
35. C. Pahl and P. Jamshidi. Microservices: A systematic mapping study. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016)*, pages 137–146, 2016.
36. Postman, Inc. Postman. `https://www.getpostman.com/`.
37. S. Segura, J. Parejo, J. Troya, and A. Ruiz-Cortés. Metamorphic testing of RESTful web APIs. *IEEE Transactions on Software Engineering*, 44(11):1083–1099, 2018.
38. SmartBear Software. SoapUI. `https://www.soapui.org/`.
39. H. M. Sneed and S. Huang. WSDLTest - A tool for testing web services. In *2006 Eighth IEEE International Symposium on Web Site Evolution (WSE '06)*, pages 14–21. IEEE, 2006.
40. Swagger-Fuzzer. Swagger-Fuzzer. `https://github.com/Lothiraldan/swagger-fuzzer`.
41. TnT-Fuzzer. TnT-Fuzzer. `https://github.com/Teebytes/TnT-Fuzzer`.
42. M. Torchiano. *effsize: Efficient Effect Size Computation*, 2015. R package version 0.5.5.
43. W.-T. Tsai, R. Paul, W. Song, and Z. Cao. Coyote: An XML-based framework for web services testing. In *7th IEEE International Symposium on High Assurance Systems Engineering, 2002. Proceedings.*, pages 173–174. IEEE, 2002.
44. E. Viglianisi, M. Dallago, and M. Ceccato. RESTTESTGEN: Automated black-box testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 142–152, 2020.
45. P. Willett. The porter stemming algorithm: then and now. *Program*, 40(3):219–223, 2006.
46. W. Xu, J. Offutt, and J. Luo. Testing web services by XML perturbation. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE '05)*, pages 10–pp. IEEE, 2005.
47. M. Zhang, B. Marculescu, and A. Arcuri. Resource-based test case generation for RESTful web services. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pages 1426–1434, New York, NY, USA, 2019. ACM.

DOI: 10.1002/stvr

APPENDIX

This appendix reports all the case studies used in the experimental validation of RESTTESTGEN.

Table X. List of the 87 *free access APIs* tested during the first experiment.

API name	API URL
Afterbanks	https://www.afterbanks.com/
APIs.guru	https://api.apis.guru/v2/
Auckland Museum	http://api.aucklandmuseum.com
CognitiveServicesManagementClient	https://management.azure.com
Power BI Embedded Management Client	https://management.azure.com
Radio & Music Services	https://rms.api.bbc.co.uk/
BC Laws	http://www.bclaws.ca/civix
Beanstream Payments	https://www.beanstream.com/api/v1
Bhagavad Gita	http://bhagavadgita.io
BikeWise API v2	https://bikewise.org/api
Cnab Online	https://cnab-online.herokuapp.com/v1
College Football Data	https://api.collegefootballdata.com/
EU BON UTIS	http://cybertaxonomy.eu/eu-bon/utis/1.0
Open Skills	http://api.dataatwork.org/v1
DatumBox	http://api.datumbox.com/
Fahrplan-Free	https://api.deutschebahn.com/freeplan/v1
dweet.io	https://dweet.io/
U.S. EPA ECHO	https://ofmpub.epa.gov/echo
Europeana	http://www.europeana.eu/api
ExaVault	https://api.exavault.com/
Flickr API Schema	https://api.flickr.com/services
FraudLabs Pro Fraud Detection	https://api.fraudlabspro.com
FraudLabs Pro SMS Verification	https://api.fraudlabspro.com
bng2latlong	https://api.getthedata.com
Android Device Provisioning Partner	https://androiddeviceprovisioning.googleapis.com/
Hangouts Chat	https://chat.googleapis.com/
SheetLabs vs	https://sheetlabs.com/IND/vs
SlideRoom API V2	https://api.slideroom.com
SpectroCoin Merchant	https://spectrocoin.com/api/merchant/1
City of Surrey Open511	http://data.surrey.ca/open511
City of Surrey Traffic Loop Count API.	http://gis.surrey.ca:8080/fmedatastreaming/TrafficLoopCount
Swagger Generator	https://generator.swagger.io/api
Available API endpoints	https://www.versioneye.com/api/v2
VocaDB	https://vocadb.net
CitySDK Linked Data	http://api.citysdk.waag.org/
Inventory	https://developer.walmart.com/inventoryProxy/inventory-api-doc-app/rest
Item	https://developer.walmart.com/proxy/item-api-doc-app/rest
Orders	https://developer.walmart.com/orderProxy/order-api-doc-app/rest
Price	https://developer.walmart.com/priceProxy/price-api-doc-app/rest
WeGA	https://weber-gesamtausgabe.de/api/v1
XKCD	http://xkcd.com/
Mobility	https://developer.o2.cz/mobility/sandbox/api
Socio-demo	https://developer.o2.cz/sociodemo/sandbox/api
ODWeather	http://api.oceandrivers.com/
OpenALPR Cloud	https://api.openalpr.com/v2
Swagger2OpenAPI Converter	https://openapi-converter.herokuapp.com/api/v1
Open Targets Platform	http://api.opentargets.io/v3
OSF APIv2 Documentation	https://api.test.osf.io/v2
PI Web API 2017 Swagger Spec	https://devdata.osisoft.com/piwebapi
PayRun.IO	https://api.test.payrun.io/
Postmark	https://api.postmarkapp.com/
posty_API	http://posty-api.herokuapp.com/
Refuge Restrooms	https://www.refugerestrooms.org/api
Reloadly Topup	https://topups.reloadly.com/
RiteKit	https://api.ritekit.com/
KeyServ Solutions	https://keyserv.solutions
LanguageTool	https://languagetool.org/api/v2
Magento Enterprise	http://t213.vg/rest/default
MailboxValidator Free Email Checker	https://api.mailboxvalidator.com/
MailboxValidator Disposable Email Checker	https://virtserver.swaggerhub.com/mailboxvalidator/MailboxValidator-Disposable-Email-Checker/1.0.0
MailboxValidator Email Validation	https://api.mailboxvalidator.com
Mandrill	https://mandrillapp.com/api/1.0/
Miataru	http://service.miataru.com/v1
TrainingApi	https://southcentralus.api.cognitive.microsoft.com/customvision/v1.2/Training
Image Moderation	https://moderatecontent.com/api
Moon by Ai Weiwei & Olafur Eliasson	http://moonmoonmoonmoon.com/
Native Ads Publisher	https://api.nativeads.com
Advicent.FactFinderService	https://demo.uat.naviplancentral.com/factfinder
Neblio REST API Suite	https://ntp1node.nebl.io/
Safe Browsing	https://safebrowsing.googleapis.com/
Jumpseller API	https://api.jumpseller.com/v1
NSIDC Web Service Documentation Index	http://nsidc.org/api/dataset/2
SheetLabs rv	https://sheetlabs.com/IND/rv
BC Geographical Names Web Service -	https://apps.gov.bc.ca/pub/bcgnws
BC Gov News API Service 1.0	https://news.api.gov.bc.ca/
DriveBC's Open511	http://api.open511.gov.bc.ca/
Greenwire Public	https://greenwire.greenpeace.org/api/public
Discovery Market Research	https://discovery.gsa.gov/
HackathonWatch	http://www.hackathonwatch.com/api/
Handwrytten	https://api.handwrytten.com/v1
Healthcare	https://www.healthcare.gov/
HHS Media Services	https://api.digitalmedia.hhs.gov/api/v2
shinobiapi	https://api.hillbillysoftware.com
Icons 8	https://api.icons8.com/
Infermedica	https://api.infermedica.com/v2
IP2Location IP Geolocation	https://api.ip2location.com
IP2Proxy Proxy Detection	https://api.ip2proxy.com

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table XI. List of the 20 public *access controlled APIs* tested during the second experiment.

API name	API URL
Airport & City Search	https://test.api.amadeus.com/v1
Airport On-Time Performance	https://test.api.amadeus.com/v1
FireBrowse Beta API	http://firebrowse.org/api/v1
Bandsintown API	https://rest.bandsintown.com/
COVID19 Stats	http://corona-virus-stats.herokuapp.com/api/v1
Freesound	http://www.freesound.org/apiv2
Geneea Natural Language Processing	https://api.geneea.com/
Google Calendar API	https://www.googleapis.com/calendar/v3
YouTube Data API v3	https://youtube.googleapis.com/
Points of Interest	https://test.api.amadeus.com/v1
Stationsdatenbereitstellung	https://api.deutschebahn.com/stada/v2
Studio Ghibli API	https://ghibliapi.herokuapp.com/
Transport for London: Vehicle	https://api.digital.tfl.gov.uk/Vehicle
Transport for London: BikePoint	https://api.digital.tfl.gov.uk/BikePoint
Transport for London: Occupancy	https://api.digital.tfl.gov.uk/Occupancy
Deutsche Bahn: Fahrplan	https://api.deutschebahn.com/fahrplan-plus/v1
Spotify	https://api.spotify.com/v1
ExchangeRate-API	https://api.exchangerate-api.com/v4
Tours and Activities	https://test.api.amadeus.com/v1
Google Drive API	https://www.googleapis.com/drive/v3

Table XII. List of the 9 private *access controlled APIs* tested during the second experiment.

API name	API GitHub repository
Widgets Spa Server	https://github.com/emrachid/widgets-spa-server
SAFRS	https://github.com/thomaxxl/safrs
RealWorld App	https://github.com/gothinkster/laravel-realworld-example-app
CRUD-NodeJS-Sequelize-Swagger-MySQL	https://github.com/lucianopereira86/CRUD-NodeJS-Sequelize-Swagger-MySQL
OrderAPI	https://github.com/jainsiddharth21/OrderAPI
Users: Express + Routing-Controllers + TypeORM	https://github.com/mateusconstanzo/express-typeorm-typescript
REST API IN SLIM PHP	https://github.com/maurobonfietti/rest-api-slim-php
ToggleAPI	https://github.com/pdonatilio/ToggleAPI
spring-boot-docker-rest-api	https://github.com/abhishek70/spring-boot-docker-rest-api

DOI: 10.1002/stvr