Automated Black-box Testing of Mass Assignment Vulnerabilities in RESTful APIs

Davide Corradini*, Michele Pasqua‡ and Mariano Ceccato§

Department of Computer Science
University of Verona – Verona, Italy
Email: *davide.corradini@univr.it, ‡michele.pasqua@univr.it, §mariano.ceccato@univr.it

Abstract—Mass assignment is one of the most prominent vulnerabilities in RESTful APIs. This vulnerability originates from a misconfiguration in common web frameworks, such that naming convention and automatic binding can be exploited by an attacker to craft malicious requests writing confidential resources and (massively) overriding data, that should be read-only and/or confidential.

In this paper, we adopt a black-box testing perspective to automatically detect mass assignment vulnerabilities in RESTful APIs. Execution scenarios are generated purely based on the OpenAPI specification, that lists the available operations and their message format. Clustering is used to group similar operations and reveal read-only fields, the latter are candidate for mass assignment. Then, interaction sequences are automatically generated by instantiating abstract testing templates, trying to exploit the potential vulnerabilities. Finally, test cases are run, and their execution is assessed by a specific oracle, in order to reveal whether the vulnerability could be successfully exploited.

The proposed novel approach has been implemented and evaluated on a set of case studies written in different programming languages. The evaluation highlights that the approach is quite effective in detecting seeded vulnerabilities, with a remarkably high accuracy.

Index Terms—REST API, Security testing, Black-box testing, Automated software testing, Mass assignment

I. INTRODUCTION

RESTful APIs (or REST APIs for short) are becoming the standard technology to access web-oriented resources and to interconnect software systems across the public Internet. They operate on the web using the HTTP protocol, typically exchanging JSON payloads, and for this reason they are also known as *Web APIs*.

Considering their dominant responsibility as cornerstone integration technology to interconnect different computer systems, it is crucial to reveal defects and vulnerabilities in their implementation as soon as possible. In fact, the security of the overall integrated system builds on top of the security and reliability of the atomic REST APIs that take part to the composition.

Specific vulnerabilities are known to impend REST APIs, as reported by the OWASP Foundation in their annual survey[1]. In particular, *mass assignment* is among the most prominent vulnerabilities that are peculiar to this programming domain. This vulnerability originates from a wrong configuration of common REST API frameworks, that typically provide automatic binding between input data fields (controlled by a potential attacker) to internal data representation (e.g., to database columns). A successful exploit to a mass assignment vulnerability would allow attackers to manipulate private data, provided that they are able to guess the names in the internal data structures (e.g., database tables and columns) used by the REST API.

A largely adopted perspective when automatically testing REST APIs [1]–[5] is black-box testing, that does not require access to the APIs source code. Also in this work we adopt a black-box approach, since it allows to successfully test REST APIs that are implemented in different languages, that are based on any possible REST framework, and that are possibly composed by closed-source components. Indeed, a black-box perspective allows to test REST APIs independently of their internal architecture (typically micro-service oriented).

In this paper, we propose a novel approach to automatically test REST APIs with respect to mass assignment vulnerabilities. The specification of the REST API to test is analyzed to identify what data should not be overwritten by incoming requests, namely data that are supposed to be read-only and hence candidate for mass assignment attacks. Our analysis first identifies groups of operations with similar data using clustering, then some heuristics are exploited to identify read-only

---

[1] https://owasp.org/www-project-api-security/

fields. Subsequently, concrete test scenarios are automatically generated by instantiating a catalog of abstract test templates. Eventually, a security oracle monitors the execution of these test cases to reveal when a vulnerability is exposed.

Empirical assessment suggests that our approach is very accurate in detecting mass assignment vulnerabilities, in fact all the vulnerabilities could be detected in almost all the considered case studies, with no false positive.

To the best of our knowledge, no automated black-box approach is available in the literature to detect mass assignment vulnerabilities on REST APIs.

The paper is organized as follows. After covering the required background on REST APIs and mass assignment in Section II, Section III presents our novel testing approach for mass assignment. Section IV presents the definition of our experimental design, while Section V applies it and comments the collected results. After comparing our approach with related work in Section VI, Section VII closes the paper.

II. BACKGROUND

A. RESTful APIs

A RESTful API (or REST API for short) is a web API that adheres to the REST (REpresentational State Transfer) architectural style [6]. REST APIs provide a uniform interface to create (C), read (R), update (U), and delete (D) resources (known as CRUD semantics). A resource is identified by a HTTP URI, and CRUD operations are typically mapped to the HTTP methods POST, GET, PUT (or PATCH) and DELETE.

As an example, consider *VAmPI*[2], an open source REST API to manage users and books of a library. The HTTP URI pointing to a user resource is /users, and the HTTP operations GET /users and POST /users/register are used to retrieve the list of registered users and to register a new user in the system, respectively.

A REST API may accept input parameters to specify additional information for executing an operation, such as the identifier of the user to retrieve (e.g., /users/{username}) or a structured object to be registered in the system in the body of the request by means of a POST method.

B. The OpenAPI Specification

OpenAPI[3] 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, which authentication schema is adopted, and the details of all the operations available in the API. In particular, for each operation the input parameters (and their schema) to be used in requests, and the schema of responses are given.

Fig. 1 contains an excerpt of the OpenAPI specification for our example *VAmPI*. After an initial header that specifies versions, licenses, and the base URL of the API (not shown in the code for space reasons), the OpenAPI specification

```
/users:
  get:
    operationId: retrieve_all_users
    responses:
      '200':
        description: See all details of the users
        content:
          application/json:
            schema:
              type: array
              items:
                type: object
                properties:
                  admin:
                    type: boolean
                  email:
                    type: string
                  password:
                    type: string
                  username:
                    type: string
/users/register:
  post:
    operationId: register_new_user
    requestBody:
      content:
        application/json:
          required: true
          schema:
            type: object
            properties:
              username:
                type: string
              password:
                type: string
              email:
                type: string
```

Fig. 1. Part of the OpenAPI specification for *VAmPI*.

contains an array of *paths*, namely the list of URL paths available in the API. In our example, we just focus on two URL paths: /users, and /users/register. Each path supports one or more HTTP methods. A path together with an HTTP method composes an *operation*, which is usually identified by an *operation ID*. For instance, the method GET in /users (*retrieve all users*) is used to retrieve the list of all registered users in the system, while the method POST in /users/register refers to the operation *register user*, meant to register a new user in the system.

Input and output of operations are associated to a *schema* that specifies their type and, optionally, a set of constraints on values (e.g., a *minimum* or *maximum* value for numeric parameters). Types can be atomic (e.g., integers and strings) or structured (i.e., compound objects). For instance, the *register new user* operation expects a request body containing a JSON object with three string fields: username, password, and email. Similarly, the response schema of the *retrieve all users* operation is a JSON array of JSON objects, each one composed by four fields: three string fields username, password, and email; and a boolean field admin.

C. Mass Assignment Vulnerability

Developers usually rely on frameworks to develop their REST API. These frameworks, available for all the major web programming languages, provide a set of features and automations to speed up and simplify the development of a

REST API. Among the most adopted frameworks we find Spring (Java), Express.js (JavaScript), FastAPI (Python), Flask (Python), Laravel (PHP), and Slim (PHP).

A common practice supported by REST API frameworks is to use a naming convention to map input data in HTTP requests to the back-end data representation inside the REST API, such as input parameters to database columns, when they have the same name. This feature is typically enabled by default, and it is exploited to speed up the development of REST APIs by simplifying the flow of data.

As stated by the OWASP Foundation, a REST API is said to be vulnerable to *mass assignment* when the value of a resource meant to be read-only can be manipulated by external users, by exploiting a misconfiguration of the automatic parameter binding.

For example, the *VAmPI* API is built on top of the Flask framework, that automatically maps the JSON `user` object in an HTTP request to the `User` Python class, that is eventually mapped to the `users` database table.

The `users` table (and `User` class) contains an `admin` boolean column (class field) meant to be read-only, i.e., while other properties of a user are meant to be edited by the end-user (e.g., the `password` or the `email` address), the `admin` field is supposed not to be changed through this operation, in fact it is not documented among the input parameters in the OpenAPI specification of this operation.

However, if the REST framework is not properly configured (as in the case of the default configuration), then the framework will automatically bind an additional `admin` HTTP parameter in a *register user* request to the corresponding `admin` column in the `users` database table. An attacker could forge a request to the operation `POST /users/register` meant to register a new user, by injecting an additional `admin` parameter, that is not mentioned in the OpenAPI specification. The additional parameter `admin` is automatically linked to the `admin` column in the `users` table, so the HTTP parameter value controlled by the attacker will overwrite the legitimate value in the database.

Automatic binding boosts productivity: it dispenses developers from the manual configuration of input/output data and database tables/columns relations. Indeed, it is a common feature, provided by all main REST API frameworks. Nevertheless, blind use of this feature often leads to mass assignment vulnerabilities, also known as *auto-binding* vulnerabilities or *object injection* vulnerabilities.

REST API frameworks nowadays provide a way to mitigate this vulnerability, by allowing developers to explicitly define which HTTP input parameters are allowed to be mapped to the internal data representation. However, developers should be aware of mass assignment vulnerabilities in order to apply this mitigation correctly. As a matter of fact, the mass assignment vulnerability is still widespread and listed among the most common vulnerabilities in REST APIs.

III. APPROACH

The OWASP Foundation provides some guidelines [7] to detect and mitigate mass assignment vulnerabilities, even in the case when APIs source code is not accessible (i.e., from a black-box perspective). These require the identification of HTTP requests that may create or update resources in the back-end, and the identification of sensitive object fields (e.g., an `isAdmin` attribute), analyzing the responses received by the back-end. Guidelines are meant to be exploited manually from developers, hence they do not provide any hint on how to automatize the process of mass assignment vulnerabilities detection.

Hence, we propose a *security testing* approach to automatically find potential mass assignment vulnerabilities in REST APIs. Usually, the source code of a REST API is not (or just partially) available. Furthermore, REST APIs are typically composed by many dynamically allocated distributed components (distinctive of a micro-services architecture), making source code analyses computationally challenging. For these reasons, we adopt a *black-box* perspective, i.e., our approach does not assume source code access. This also implies that our approach is applicable to any REST API, independently by the programming language or framework used for its implementation. The only requirements of our approach are: (i) the availability of an OpenAPI specification, to retrieve operations and their input/output message format; and (ii) black-box HTTP access to the REST API under test, to send test requests and receive responses.

Our strategy to test mass assignment vulnerabilities is composed of these main three steps.

1) *Identification of read-only attributes.* We parse the OpenAPI specification of the REST API under test to list operations and their input/output attributes. Operations are subject to clustering, to group together those that handle similar data resources. We then compare operations handling similar resources to spot read-only attributes, i.e., those that appear as output of *read* operations, but not as input in *write* operations. These fields are the candidate for mass assignment vulnerabilities.

2) *Test case generation.* Test scenarios are automatically generated as sequences of requests, with the aim of trying to overwrite read-only attributes. Test cases are generated by computing concrete interactions, starting from abstract testing templates.

3) *Security oracle.* Test cases are executed on the REST API under test, and their execution is monitored by our *security oracle*. The oracle reveals a successful exploitation of a mass assignment vulnerability when a test case manages to overwrite a read-only parameter.

In the following, we detail all the aforementioned steps.

A. Identification of Read-only Attributes

The first step towards testing mass assignment vulnerabilities consists in the identification of read-only attributes, according to the OpenAPI specification, since they are potential targets of successful attacks. In particular, this requires

Operation	admin	email	password	usernam
register user	False	True	True	True
retrieve all users	True	True	True	True

to identify whether operations read or write data (i.e., infer operations CRUD semantics) and when these actions are performed on the same resource.

CRUD semantics and operations resources. We propose an automatic procedure to infer operations CRUD semantics and the resources that operations act on. This information is collected by inspecting the OpenAPI specification only.

Inferring operations CRUD semantics. To infer the read-/write semantics of operations, we rely on their HTTP method as follows. We assume that POST methods are used to *create* (that we annotate as C) resources, while GET methods are used to *read* resources. We distinguish reading a single resource (that we annotate as R) when the return schema type is a single object, from reading multiple resources at once, that we call *read-multi* (that we annotate as RM), when the schema return type is an array of objects. Furthermore, PUT and PATCH methods are considered to be used to update (that we annotate as U) resources, while DELETE methods (that we annotate as D) are considered to be used to delete resources.

With respect to the example in Fig. 1, the operation GET /users is tagged as a read-multi (RM), in fact its return type is an array, while the operation POST /users/register is recognized as create (C).

Despite there in no constraint for REST APIs to meet this mapping, typically programmers follow this convention: the converse is generally considered an antipattern and a bad practice [8].

Grouping operations by resource type. To detect read-only attributes it is important to understand when operations handle the same type of resource, either in read or in write mode. In fact, as in the example in Fig. 1, operations definition does not mention that operations handle *user* resources. Moreover, operations handling the same type of resource rarely operate on exactly the same set of fields. For instance, *register new user* has three input parameters, while *retrieve all users* has four output parameters.

In order to group together operations with the most *similar* parameters, we use clustering. We collect the names of the input/output parameters for all the operations in the OpenAPI specification. These names are normalized using the Porter stemming algorithm [9], to keep only the root of words (e.g., in case singular/plural is used). After this, duplicate names are discarded and the remaining ones are sorted alphabetically into a global list. Eventually, as shown in Table I, each operation is encoded as a boolean array, with the i-th element of the array set to true, when the i-th parameter in the global list is an input or output parameter of this operation, according to the OpenAPI specification.

Then, we apply the expectation maximization (EM) clustering algorithm on this operation representation. The EM clustering algorithm automatically determines the optimal number of clusters, and which operations belong to each cluster, based on common parameters. In our example, it is very likely that the operations *retrieve all users* and *register new user* would be assigned to the same cluster of *user* resources, because they have a very similar set of input/output parameters.

Detecting resource identifiers. Finally, it is not enough to understand when operations insist on the same resource type (e.g., on *users*), we should also detect when the same resource instance is used in subsequent operations in a test scenario (e.g., the same *user* resource). To achieve this objective, we detect parameters that act as resource (unique) identifiers, that we call the *resource-ids*, commonly used by REST APIs.

We recognize the resource-id with a simple but effective heuristic, that relies on common naming practices[4]. Among the input/output parameters in an operation, we consider as resource-id the field that ends with the suffix *id* or *name*. In case more than one field satisfies this simple pattern, in a complex schema with different level of nesting, we pick the less nested field. In case we find more than one field at the same level, we prefer the one with *id* as suffix. In case the selection is still not unique, we take the first one.

In our example, the field username would be selected as resource-id for all the operations on the resource *user*.

Specification annotation. Once CRUD semantics, resource types and resource-ids are identified, they are saved in the OpenAPI itself. To this aim, we elaborated an annotation syntax to add these information to the OpenAPI specification. Annotations are based on custom extension fields as follows.

- x-crudOperationSemantics annotates an operation with its CRUD semantics. This field can be create, read, update, delete, read-multi, or none[5].
- x-crudResourceType annotates an operation with the type of the handled resource. This annotation accept a string value.
- x-crudResourceIdentifier annotates the parameter that is used as resource-id in the operation.

Alternatively, a developer might manually annotate the OpenAPI with these custom tags to skip our automated detection. In fact, despite guidelines and best practices inspired our identifications of the CRUD semantics, resource types and resource-ids detection might be imprecise.

Read-only parameters detection. After all the required information is collected, we can start the actual identification of read-only parameters. Inspecting the tags in the instrumented specification (either by means of the clustering or manually), operations that handle resources of the same type are grouped together. Read-only parameters are simply detected as those parameters that are available as output in read (R, or RM)

---

[4]https://restfulapi.net/resource-naming/

[5]The annotation none indicates that no CRUD semantics is present for the operation, since it does not involve any resource (e.g., login operations).

operations, but are never used as input in create (C) nor update (U) operation in the same group.

In our example, the `admin` parameter would be classified as read-only, because it can be accessed by the read operation `GET /users`, but it can never be modified by the write operation `POST /users/register`.

B. Test Case Generation

After read-only fields are identified, we need to generate execution scenarios that indeed attempt at overwriting them. The testing approach relies on *abstract test templates* that adopt the following notation.

- For a resource of type τ, requests to operations involving τ resources are labeled as C_τ, R_τ, RM_τ, U_τ, and D_τ, to represent, respectively, requests meant to create, read, read multiple resources, update or delete a resource of type τ.
- An operation request is labeled with the plus symbol C_τ^{+f}, when a request to the operation C_τ is modified by adding the extra parameter f that is not documented in the specification of C, in the attempt of overwriting the read-only attribute f.
- A sequence of requests is tagged with the question mark when it is optional (but recommended). They can be used, for instance, to clean up the REST API from testing data.

An abstract test template is formalized as a sequence of labels describing operations and resources type. In particular, we identified two abstract test templates: *Update-injection Sequence* and *Create-injection Sequence*.

Template 1: Update-injection Sequence

$$< C_\tau, R_\tau, U_\tau^{+f}, R_\tau, (D_\tau, R_\tau)^? >$$

This sequence is intended to test a potential mass assignment vulnerability in the update operation U_τ. The sequence starts with the creation of a resource in the REST API. A read operation on this resource is then performed to check a successful creation. The third operation is the update operation U_τ^{+f} in which the read-only parameter f is injected. The subsequent read operation R_τ is meant to check if the exploit is successful, by comparing the value of f before and after the injection U_τ^{+f}. In case the read-only parameter has changed, a mass assignment vulnerability has been exposed.

Additionally, the created resource may be deleted, with the aim of cleaning testing side effects (i.e., testing data). In this case, a last read operation checks if the resource has been successfully deleted or if it is still present (i.e., the delete operation has failed).

Template 2: Create-injection Sequence

$$< C_\tau^{+f}, R_\tau, (D_\tau, R_\tau)^? >$$

This sequence is intended to test a potential mass assignment vulnerability in the create operation C_τ. The first operation in the sequence tries to create a resource by specifying an additional read-only parameter. Subsequently, a read operation checks if the read-only parameter in the new fresh resource has been successfully overwritten.

As in the previous case, two optional requests close the sequence to clean the state of the REST API: the deletion of the testing resource and a check for successful deletion.

Note that, the last two requests $(D_\tau, R_\tau)^?$ in both abstract test templates are not directly involved in the mass assignment vulnerability, indeed they are optional. For instance, in case no delete operation D_τ is available in the REST API under test, this optional cleanup sequence has to be skipped.

From abstract to concrete test cases. Abstract template sequences have to be instantiated in order to be executed on the REST API under test. For each abstract operation in the sequence, we have to generate the corresponding HTTP request, including values for the input parameters.

Input values. Values for input parameters of requests (that are not resource-ids) are obtained either from the OpenAPI specification, or by random generation. The OpenAPI specification provides for each parameter a set of example values and the parameter default value. Moreover, in the case of enum parameters, also a set containing the supported enum values is usually provided. We resort to these values, when available. In case these values are not available or do not allow generating successful requests (e.g., incorrect example values), we switch to random values, that are generated to match the parameter format defined in the specification. For string parameters that specify no specific format, we try to infer the format from the parameter name. For instance, if a field is named `email`, we try to generate a valid e-mail address. We support more than 20 string formats including dates, timestamps, e-mail addresses, phone numbers, URLs, UUIDs, etc.

Resource-id values. All the operations in a test template are supposed to operate on the same resource instance. For example, after creating a resource, the subsequent *read* operation is supposed to access the just created resource. To make sure that this is the case, the resource-id field is inspected in the output of *create* operations C_τ and used as input in subsequent *read*, *update* and *delete* operations R_τ, U_τ and D_τ. Since the resource-id would be the same for all the operations in a sequence, in order to keep the notation simple, we did not specify which resource-id is handled by each operation.

However, sometimes the output of a *create* operation C_τ does not contain the resource-id of the fresh resource, supposed to be used in the subsequent *read* operation R_τ. To solve this problem, whenever available, we resort to a read-multi operations RM_τ to list all the resources of type τ before and after the creation C_τ. We expect to identify the newly created resource as the difference between these two lists. The two abstract test templates are modified in this way:

Template 1': $< RM_\tau, C_\tau, RM_\tau, U_\tau^{+f}, R_\tau, (D_\tau, R_\tau)^? >$

Template 2': $< RM_\tau, C_\tau^{+f}, RM_\tau, (D_\tau, R_\tau)^? >$

Concrete test cases. Instantiating a concrete test case starting from an abstract template requires to successfully execute each operation from the corresponding sequence. However,

there could be multiple candidate operations for a step in the sequence, e.g., more than one operations R_τ to read a resource of type τ. A candidate operation can be tested with different input values until it succeeds (status code 2XX). The maximum number of attempts for testing an operation (MAX_OPERATION_ATTEMPTS) is configurable. In case no attempt can obtain a successful response, we consider the whole instantiation of the template as failed. The instantiation of the template is restarted from scratch until all the operations are executed successfully, for a maximum number of attempts MAX_TEMPLATE_ATTEMPTS.

A larger number of attempts will increase the probability of successfully instantiating a sequence, but also a longer test generation time.

C. Security Oracle

The responsibility of the security oracle is to classify test case executions and judge whether they are able to reveal a mass assignment vulnerability in the REST API under test. To this aim, the oracle verifies that the same resource-id is used by all the operations in a sequence, and that the injection was successful with a value different from the default one. These three aspects of the oracle are described in the following.

Same resource-id check. As described earlier, all the operations in a test scenario should, by construction, handle the same resource instance, by using the same value of the resource-id field. The security oracle has to check that this constraint is met when a test case is executed.

Read-only fields overwrite. When the oracle detects that a read-only value has been successfully overwritten, it reports a mass assignment vulnerability. In practice, the oracle checks if the injected update request U_τ^{+f}, or the injected create request C_τ^{+f} could overwrite the value of the read-only parameter f. This is achieved by comparing the injected value of the read-only parameter f with the value observed from the subsequent read request R_τ. In case the two values correspond, the test case managed to write the read-only field f.

Injection of default values. When instantiating an abstract test template into a concrete test case, values should be chosen for all the input fields. When choosing the value for the read-only parameter, we could unintentionally guess its *default* value, especially for boolean or enum types that support a limited set of possible values. If the subsequent read operation reveals that the resource contain the injected value in the read-only field, our oracle could report a vulnerability. However, it could be the case that the attack failed, and the read-only parameter has the value that it would have had in the nominal case, i.e., its default value. To check for this case and avoid a false positive, each test case that our oracle classifies as a successful injection is repeated using a different value for the injected read-only parameter. Our objective is to make sure that the injection is successful with at least two distinct values of the read-only field, and avoid false positives due to default values.

Identification of other defects. Despite the objective of this oracle is to reveal mass assignment vulnerabilities when running test cases, it is also able to detect other kind of defects.

Injection consists of a request that violates the specification by adding an undocumented field. Thus, a correctly implemented REST API should respond with a 4XX status code, and reject the malformed request with the unexpected parameter f, because it is not consistent with the documented request format. A 2XX status code, instead, means that the REST API accepted as valid the injected, and thus invalid, request. Even if injection did not happen, accepting a malformed request by itself is a defect. Additionally, a response status code 5XX stands for an internal server error that was not handled correctly (e.g., because of an uncontrolled exception). This suggests a potential implementation defect, that is however different from the mass assignment vulnerability.

IV. EXPERIMENTAL SETTING

In this section, we present the experimental setting for the empirical validation of our approach.

A. Research Questions

First, our approach requires to identify operations CRUD semantics, what resource types they handle, and what parameter they use as resource-id. Hence, our empirical investigation first assesses the accuracy of the automated identification of such information.

RQ1: What is the accuracy of the automated identification of operations CRUD semantics, resource types, and resource-id parameters?

Second, the main objective of our approach is to reveal mass assignment vulnerabilities in REST APIs, this is investigated by the following research question.

RQ2: What is the accuracy in revealing mass assignment vulnerabilities in REST APIs?

Finally, our approach should be able to deal with real-world production-ready REST services, that can be complex and large in size. Hence, the last research question investigates the scalability of the approach on large REST APIs.

RQ3: Does the proposed approach to detect mass assignment vulnerabilities scale to large REST APIs?

B. Case Studies

The experimental validation is conducted on a set of REST APIs. However, considering that test cases are supposed to exploit a security vulnerability, it would not be ethical to test publicly hosted APIs (such as those on *APIs.guru*[6]), since their integrity could be compromised by a successful attack. Hence, we opted for case studies that we can download and run in a controlled environment.

For these reasons, we looked for open-source projects hosted on *GitHub*, so that we can compile and install them locally. This also grant us full control over the REST APIs state and data, allowing us to restore an API initial state and,

---

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

TABLE II
VULNERABLE REST APIS CONSIDERED FOR ASSESSING THE ACCURACY
OF THE APPROACH.

Case study	Language	Framework	# Ops.	# Vuln.
VAmPI [10]	Python	Flask	12	1
OWASP [11]	Java	Spring	10	4
Toggle [12]	ASP.NET	.NET Core	16	2
Bookstore [13]	Java	Spring	5	1
CRUD [14]	Node.js	Express	4	2

hence, avoiding potential side effects originated by previous injections.

We queried the GitHub search engine with the following base strings: "REST", "RESTful API", "mass assignment", "autobinding", "object injection", "OpenAPI" and "Swagger". Then, we also added query strings representing framework commonly used to implement REST APIs, such as "swagger-ui", "SpringFox", "swagger-jsdoc", "flask-swagger". Among the REST APIs resulting from the search, we selected those containing an OpenAPI specification, which is a requirement of our black-box testing approach.

These potential case studies have been downloaded, compiled and run to discard those that failed either in compiling or in running. After this last filtering, our final set of case studies is shown in Table II. It consists of 5 REST APIs, written in different programming languages (Java, ASP.NET, Python, and Node.js), based on different frameworks and DBMSs, for a total of 26 endpoints and 47 operations.

The table also reports the number of vulnerabilities in each case studies. While *VAmPI* and *OWASP* already contained some vulnerabilities (1 and 4, respectively), the other case studies did not, so they have been manually seeded with vulnerabilities with the following procedure. We edited the internal representation of a resource in the REST API (e.g., resource *user*) to add a new field with arbitrary name. Then, we edited the code of a *write* operation that handles this type of resource (e.g., PUT /user) to enable automatic mapping. Eventually, we edited a *read* operation for this resource type and its section in the OpenAPI specification to add the novel field to those that can be read.

Moreover, we also prepared non-vulnerable (safe) versions for all the case studies. *VAmPI* supports an environment boolean variable to enable or disable the vulnerability. The repository of *OWASP* contains both a vulnerable and a patched version. For the remaining case studies, we fixed the vulnerable version, adopting the mitigation features of frameworks.

Eventually, our approach requires a correct OpenAPI specification as input. However, in some cases, specifications of open-source REST APIs are incorrect or incomplete. So, before running the experiment, all the OpenAPI specifications have been checked against the source code by a team of experts, including an author of this paper and two other external contributors. Few corrections have been applied, e.g., response schemas have been added in case they were missing. The same team also elaborated the ground truth for CRUD semantics, resource types, and resource-ids of all the operations.

Finally, to answer the third research question, we considered a collection of more complex case studies, namely 10 mainstream Google REST APIs (Gmail, Google Analytics, Calendar, Classroom, Custom Search, Drive, Fitness, My Business, Search Console and YouTube).

C. Experimental Procedure

We implemented our approach as a testing strategy, MassAssignmentSecurityTestingStrategy, on top of RestTestGen [1]. RestTestGen is an open-source REST APIs testing tool (available on GitHub[7], where you can also find the mass assignment strategy) and it provides some commodity and utility features that facilitated the implementation of our approach. They are a solid parser for the OpenAPI specification, procedures to construct and send HTTP requests to the REST API under test, and receive and parse the corresponding HTTP responses.

We configured the parameters of the template instantiation with the following values: MAX_OPERATION_ATTEMPTS = 12, and MAX_TEMPLATE_ATTEMPTS = 3. These values happened to be an optimal trade-off between testing time and successful vulnerabilities identifications during our dry run experiments.

Case studies have been taken into a testable state, i.e., their states have been initialized with some data, preparatory for testing. Most of the case studies come with a specific script, while some other required several manual HTTP interaction in order to provide initialization data.

Our testing strategy has been executed on each case study, both on the vulnerable and safe version. The process has been repeated ten times, to control potential random variation due to the non-deterministic algorithms in our approach. After each testing session, we restored the same identical state of the REST APIs under test.

D. Metrics

In our experiment, we computed and collected the following metrics.

Correctness of CRUD identification. For each REST API, we compute the ratio of the operations for which the correct CRUD semantic was identified divided by total number of the operations with a CRUD semantics in the specifications, according to the manually provided ground truth.

Clustering correctness. We manually inspect the result of clustering to identify what is the resource type that is handled by the majority of the operations in each cluster. Those operations that handle a resource type different from the majority do, are considered as assigned to the wrong cluster. This metric is then computed as the ratio of the operations assigned to the correct cluster divided by the total number of CRUD operations.

Correctness of resource-id identification. This metric is computed as the ratio of the operation schemas for which the

---

[7]https://github.com/SeUniVr/RestTestGen

resource-id field has been correctly identified divided by the total number of schemas of CRUD operations that contain a resource-id. Operations can have an input schema and/or an output schema.

Accuracy in vulnerability detection. To measure the accuracy in detecting mass assignment vulnerabilities, we use standard information retrieval metrics.

- *True Positives (TP):* number of vulnerabilities that are correctly detected.
- *False Positives (FP):* number of cases reported as vulnerabilities that are not actually vulnerable (false alarms).
- *False Negatives (FN):* number of vulnerabilities that are missed (not reported by the tool).
- *Precision (Pr):* number of correctly detected vulnerabilities on the number of reported vulnerability (*Pr = TP / (TP + FP)*).
- *Recall (Re):* number of correctly reported vulnerabilities on the number of vulnerabilities (*Re = TP / (TP + FN)*).

E. Threats to Validity

Here we discuss the main threats to the validity of our findings.

Threats to the *conclusion validity* are concerned with issues that affect the ability to draw the correct conclusion. To limit this threat, we considered standard information retrieval metrics (i.e., precision and recall). Additionally, to increase heterogeneity of samples in the data set, we considered software projects written in multiple programming languages and based on different frameworks.

Threats to *internal validity* concern the subjective factors that might have affected the results, such as the definition of the gold standard to compare the results of our approach. To limit this subjectivity, we inspected the source code of operation to determine its actual CRUD semantic, the type of the read/written resource and what field is used as resource-id (if any).

The threats to *construct validity* concern the data collection and analysis procedures. To mitigate this threat, we made sure that, when injecting faults in case studies, the vulnerability definition as explained by the OWASP Foundation was precisely followed.

Threats to *external validity* concern the generalization of our findings. Despite our analysis considers REST APIs written in different programming languages and based on different rest frameworks, our analysis could still be biased because we only considered open-source projects. Our results might not extend in general to other software projects, for instance to closed-source industrial software. Only replications with more samples can confirm our findings beyond our experimental settings.

V. EXPERIMENTAL RESULTS

This section presents the results of our experimental validation. The replication package for this experiment is available online [15].

TABLE III
CORRECTNESS OF CRUD SEMANTICS IDENTIFICATION, OF OPERATIONS CLUSTERING, AND RESOURCE-ID IDENTIFICATION.

Case study	CRUD	Clustering	Resource-id
VAmPI	100%	100%	67%
OWASP	100%	80%	100%
Toggle	88%	88%	100%
Bookstore	100%	100%	100%
CRUD	100%	100%	100%
Average	98%	94%	93%

A. RQ1: Accuracy of CRUD Semantics, Resource Type, and Resource-id

Before starting the actual test case generation, some preliminary results should be computed purely based on the OpenAPI specification. The accuracy of this first step is investigated by the first research question, for all the case studies. The tool automated labeling has been compared with the gold standard, and results are reported in Table III. For each case study (first column), the table reports the correctness of the CRUD semantics identification (second column), the clustering correctness (third column), and the correctness of resource-id identification (fourth column).

The identification of CRUD semantics is 100% for almost all the case studies, reaching an average correctness of 98%. Such a high accuracy is due to a disciplined use of HTTP methods in the observed REST APIs. Despite accuracy was still very high (i.e., 88%), a wrong semantics have been identified in some operations of *Toggle*. A manual investigation revealed that *read multi* operations were automatically labeled as *read*, because of an uncommon return type. Instead of returning an array of objects, a single JSON object is returned, that, in turn, contains many sub-objects that corresponds to resources.

The clustering correctness is also quite high, with an average value of 94%. Operations that have been allocated to the wrong cluster turned out to be only *delete* operations. This is due to the fact that, typically, *delete* operations only require no other input than the resource-id of the resource to delete. When the other operations insisting on the same resource type have a lot more input parameters than the *delete* operation, the clustering algorithm is very likely to assign the latter to a separate cluster (only containing the *delete* operation itself). Nevertheless, availability of *delete* operations is not strictly required for our testing approach (D_τ is optional in both *Template 1* and *Template 2*).

Resource-ids identification is also quite accurate, with an average correctness of 93%. Inaccurate detection was due to custom naming in some case studies that violated common practices. For instance, in *VAmPI* uses the field `book_title` as resource identifier, instead of a more intuitive `book_id` or `book_name`.

Based on this results we can answer RQ1 in this way:

TABLE IV
ACCURACY IN REVEALING MASS ASSIGNMENT VULNERABILITIES.

Case study	Safe		Vulnerable					
	Tests	FP	Tests	TP	FP	FN	Pr	Re
VAmPI	4.0	0.0	4.0	1.0	0.0	0.0	100%	100%
OWASP	8.0	0.0	7.4	3.6	0.0	0.4	100%	90%
Toggle	2.0	0.0	2.0	2.0	0.0	0.0	100%	100%
Bookstore	2.0	0.0	2.0	1.0	0.0	0.0	100%	100%
CRUD	2.0	0.0	2.0	2.0	0.0	0.0	100%	100%

> *Our approach shown a quite high accuracy in analyzing the OpenAPI specification, because the CRUD semantics was detected correctly for 98% of the operations, 94% of the operations have been grouped together when they handle a resource of the same type, and the 93% of resource-ids have been correctly identified.*

B. RQ2: Accuracy of Vulnerability Detection

Then, we applied our testing approach to two variants of each case study, i.e., with and without vulnerabilities. The outcome of the oracle evaluating the automatically generated test cases has been compared with the gold standard. The accuracy of vulnerability detection is shown in Table IV in terms of true positives (TP), false positives (FP), false negatives (FN), precision (Pr), and recall (Re). The table also reports the number of concrete test cases that could be successfully generated for the read-only fields detected on each variant of each case study. The table reports the average values for the ten repetitions of each testing session (thus, decimal values).

Accuracy in testing mass assignment vulnerabilities shown to be very high. All the vulnerabilities are detected in almost of all case studies, with no false positive. Only in one run out of ten (this is why the average TP is a decimal number in the table) on *OWASP*, our approach missed few vulnerabilities. Manual investigation on test execution traces revealed that, in a particular execution on *OWASP*, our approach could only instantiate two sequences out of eight, so the corresponding vulnerabilities could not be tested.

As a side effect of security testing, our tests could also reveal the presence of other defects such as crashes (status code 5XX) in all case studies, except for *Toggle*. In total total, 24 HTTP responses occurred with status code 500. Manual inspection on HTTP requests and responses revealed that they originated from 5 distinct bugs.

Based on these results, we can formulate the subsequent answer to RQ2:

> *The accuracy in revealing mass assignment vulnerabilities with automated black-box testing is very high, because all the vulnerabilities could be detected in our case studies with no false positives, with the only exception of a single execution out of ten in a single case study.*

TABLE V
GOOGLE REST APIs CONSIDERED FOR ASSESSING THE SCALABILITY OF THE APPROACH.

Case study	# Ops.	Time (s)	# Read-only fields
Gmail	68	3.0	23
Analytics	88	5.0	166
Calendar	37	2.0	11
Classroom	61	5.0	15
Custom Search	2	1.0	66
Drive	48	3.0	49
Fitness	13	1.4	4
My Business	50	7.4	527
Search Console	11	1.0	10
YouTube	76	8.4	110
Total	**454**	**37.2**	**981**

C. RQ3: Scalability of the Approach

To study the scalability of our approach, we ran the static analysis part on the specifications of 10 mainstream Google services, listed in Table V, that include a total of 454 operations. To mitigate the impact of non-determinism introduced by clustering, each service have been tested 5 times independently. Note that, we could not run the dynamic test case generation part for ethical reasons, i.e., we meant to avoid the risk of mounting a successful injection attack on a service that runs in production.

The static analysis of all services took 37.2 seconds (on average) and, in total, it revealed 981 read-only fields. Even if we could not run the automated test case generation part, we expect that not a long time would be needed to validate these candidate vulnerabilities, with few test cases. So, we can speculate that scalability on large REST APIs does not represent an issue for our approach.

Thus, we can answer RQ3 in this way:

> *Our approach seems to scale well on large APIs, because it took 37.2 seconds to analyze remarkably large OpenAPI specifications, detecting 981 read-only parameters (on a total of 454 operations) that are potential candidates of mass-assignment vulnerabilities.*

VI. RELATED WORK

While most of the literature on automated testing on REST APIs is focused on functional testing, just few approaches to automate security testing are starting to rise [2]–[5], in order to detect potential security vulnerabilities in REST APIs.

In particular, Mai et al. [3] use metamorphic relations to address the oracle problem: 22 system-agnostic metamorphic relations are defined in order to automate security testing in Web systems. Their approach is not black-box and does not target REST APIs. Furthermore, mass assignment vulnerabilities are not part of their catalog of metamorphic relations.

Atlidakis et al. [2] introduce four security rules that capture desirable properties of REST APIs, specifically meant to test some security relevant aspects of Azure REST APIs. In particular, their objective is to test four conditions: (i)

use-after-free, when it is possible to read a resource after it has been deleted; (ii) resource-leak, when a resource can be read after its creation failed; (iii) resource-hierarchy, when a resource is accessible under the wrong parent resource; and (iv) user-namespace, when a private resource of a user is accessible to another user. Despite being effective, their approach targets particular security aspects, that do not comprise mass assignment vulnerabilities. Recall that, the latter is indicated by the OWASP Foundation as one of the most important REST APIs vulnerability to mitigate.

Luo et al. [4], [5] focus on access control. In their first work [4], 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 [5], 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 to mitigate users intervention. Access control is surely an import aspect that testing tools should address, but it is not related to a mass assignment defect.

To the best of our knowledge, no approach is available to automatically test REST APIs with respect to mass assignment vulnerabilities.

Remaining literature on functional testing is mainly split into two different lines of work.

One consists in *white-box* approaches, that rely on the availability of REST APIs source code to perform static analysis, or to instrument it to collect execution traces and metric values. In this context, Arcuri [16] proposes a fully automated solution to generate test cases with evolutionary algorithms, that requires the OpenAPI specification and the access to the Java bytecode of the REST API to test. This approach has been implemented as a tool prototype called EvoMaster, extended with the introduction of a series of novel testability transformations aimed at providing guidance in the context of commonly used API calls [17]. Unfortunately, white-box approaches are very often not practically usable in the context of REST APIs, where usually the source code is not available.

On a complementary direction, *black-box* approaches do not require any source code, which is often the case when using closed-source components and libraries. *Fuzzers* [18]–[22] 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 [19]–[21].

Godefroid et al. [23] propose a methodology to fuzz body payloads intelligently using JSON body schemas and advanced fuzzing rules. Even if they are automatic black-box tools, their goal is to generate input values to tests, so they cannot be used as standalone testing tools (except for the approach of Godefroid et al. [23] that has been implemented in RESTler).

Ed-douibi et al. [24] 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).

Karlsson et al. [25] propose QuickREST, a tool for property-based testing of RESTful APIs. Starting from the OpenAPI specification, they generate test cases with the aim to verify whether the API under test complies with some properties (i.e., definitions) documented in the specification (e.g., status codes, schemas).

Segura et al. [26] propose another black-box approach for REST APIs testing, with an oracle 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 a 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.

Corradini et al. [1] propose RestTestGen, an automated black-box test case generation tool for REST APIs. The approach allow to test nominal and error scenarios, and the test generation strategy is based on the Operation Dependency Graph, a graph which encodes data dependencies among the operations available in the API.

Atlidakis et al. [27] propose RESTler, a stateful REST API fuzzer developed at Microsoft Research. RESTler generates stateful sequences of requests by inferring producer-consumer relations between request types described in the specification. It also dynamically analyzes responses to intelligently build request sequences and avoiding sequences leading to errors.

Martin-Lopez et al. [28] propose RESTest, another automated black-box testing tool for RESTful APIs. The peculiarity of this tool is the inter-parameter dependencies support. Indeed, some REST APIs impose constraints restricting not only input values, but also the way in which input values can be combined to fill valid requests.

Finally, Laranjeiro et al. [29] propose bBOXRT, a black-box robustness testing tool for RESTful APIs. The aim of bBOXRT is to assess the robustness of REST APIs observing the behavior of services under test when providing invalid requests. The tool provide a fault model consisting of 57 different mutations applicable to input parameters of various types (numbers, strings, booleans, dates, times, arrays, etc.).

All the aforementioned black-box testing tools for REST APIs are not meant to spot security vulnerabilities, and mass assignment in particular. Nevertheless, they can be extended in order to implement the methodology presented in the present work. In this respect, we chose to apply our testing strategy on top of RestTestGen, since it provides a modular and easily extensible architecture.

VII. CONCLUSION

Dependability and confidentiality of data rely on the correct and secure implementation those REST APIs that are supposed to enforce appropriate data access policies. Mass assignment is among the most common vulnerabilities in REST APIs, often caused by wrong settings in web frameworks. These vulnerabilities might allow an attacker to directly override private internal data structures in the REST API back-end, such as sensitive database columns.

We proposed an automated approach based on black-box testing to reveal mass assignment vulnerabilities. Starting from the OpenAPI specification, we apply clustering to identify those operations that are very likely to operate on the same resources. Cluster content is compared to identify resource fields that are not supposed to be exposed in write operations, i.e., read-only data according to the developer intention. Abstract test templates are turned into concrete test cases, to test each alleged read-only field according to mass assignment vulnerability. To the best of our knowledge, this is the first work that proposed an automated approach to reveal mass assignment vulnerabilities using automated black box testing.

Experimental results suggest that a variety of programming languages are prone to these vulnerabilities, when web frameworks are not appropriately configured. As future work, we plan to extend our experimental validation to either a more complete set of open-source projects (hence, more programming languages and web frameworks) and closed-source industrial projects from our industrial partners. Additionally, we plan to extend our approach to other kind of programming defects and security vulnerabilities of REST APIs, such as *broken object level authorization* and incorrect *access control*. Finally, we plan to conduct user studies with developers to identify the most appropriate way to report vulnerabilities in REST APIs, to support a fast and accurate fix of these defects.

REFERENCES

[1] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato, "Automated black-box testing of nominal and error scenarios in restful apis," *Software Testing, Verification and Reliability*, Jan. 2022. [Online]. Available: https://doi.org/10.1002/stvr.1808

[2] 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*. IEEE, 2020, pp. 387–397. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00046

[3] 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)*, 2020, pp. 186–197.

[4] 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 *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*, I. Altintas and S. Chen, Eds. IEEE, 2017, pp. 548–555. [Online]. Available: https://doi.org/10.1109/ICWS.2017.64

[5] Y. Luo, H. Zhou, Q. Shen, A. Ruan, and Z. Wu, "RestPL: Towards a request-oriented policy language for arbitrary RESTful APIs," in *IEEE International Conference on Web Services, ICWS 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, S. Reiff-Marganiec, Ed. IEEE Computer Society, 2016, pp. 666–671. [Online]. Available: https://doi.org/10.1109/ICWS.2016.92

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

[7] OWASP Foundations, "Testing for Mass Assignment," 2022, https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/20-Testing_for_Mass_Assignment.

[8] F. Palma, J. Dubois, N. Moha, and Y.-G. Guéhéneuc, "Detection of rest patterns and antipatterns: a heuristics-based approach," in *International Conference on Service-Oriented Computing*. Springer, 2014, pp. 230–244.

[9] P. Willett, "The porter stemming algorithm: then and now," *Program*, 2006.

[10] github.com/erev0s, "Vampi case study," 2022, https://github.com/erev0s/VAmPI.

[11] github.com/mattiasanti99, "OWASP case study," 2022, https://github.com/mattiasanti99/vulnerability_OWASP_project.

[12] github.com/pdonatilio, "Toggle case study," 2022, https://github.com/pdonatilio/ToggleAPI.

[13] github.com/todo, "Bookstore case study," 2022, https://github.com/todo/Bookstore.

[14] github.com/lucianopereira86, "CRUD case study," 2022, https://github.com/lucianopereira86/CRUD-NodeJS-Sequelize-Swagger-MySQL.

[15] Replication Package, "Automated black-box testing of mass assignment vulnerabilities in restful apis," 2022, https://doi.org/10.5281/zenodo.7348518.

[16] A. Arcuri, "RESTful API automated test case generation with Evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, p. 3, 2019.

[17] 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*. IEEE, 2020, pp. 153–163. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00025

[18] API Fuzzer. (2022) API Fuzzer. https://github.com/KissPeter/APIFuzzer.

[19] Fuzz-Lightyear, "Fuzz-Lightyear," 2022, https://github.com/Yelp/fuzz-lightyear.

[20] Fuzzy-Swagger, "Fuzzy-Swagger," 2022, https://github.com/namuan/fuzzy-swagger.

[21] Swagger-Fuzzer, "Swagger-Fuzzer," 2022, https://github.com/Lothiraldan/swagger-fuzzer.

[22] TnT-Fuzzer, "TnT-Fuzzer," 2022, https://github.com/Teebytes/TnT-Fuzzer.

[23] P. Godefroid, B. Huang, and M. Polishchuk, "Intelligent REST API data fuzzing," in *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*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 725–736. [Online]. Available: https://doi.org/10.1145/3368089.3409719

[24] H. Ed-Douibi, J. L. C. Izquierdo, and J. Cabot, "Automatic generation of test cases for REST APIs: A specification-based approach," *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 181–190, 2018.

[25] S. Karlsson, A. Causevic, 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)*, 2020, pp. 131–141.

[26] S. Segura, J. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic testing of RESTful web APIs," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2018.

[27] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API fuzzing," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 748–758. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00083

[28] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Black-box constraint-based testing of RESTful web APIs," in *Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings*, ser. Lecture Notes in Computer Science, E. Kafeza, B. Benatallah, F. Martinelli, H. Hacid, A. Bouguettaya, and H. Motahari, Eds., vol. 12571. Springer, 2020, pp. 459–475. [Online]. Available: https://doi.org/10.1007/978-3-030-65310-1_33

[29] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A black box tool for robustness testing of REST services," *IEEE Access*, vol. 9, pp. 24738–24754, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3056505