

Fuzzing for Detecting Access Policy Violations in REST APIs

Andrea Arcuri

Kristiania University of Applied Sciences and OsloMet
Oslo, Norway
andrea.arcuri@kristiania.no

Omur Sahin

Erciyes University
Kayseri, Türkiye
omur@erciyes.edu.tr

Man Zhang*

Beihang University
Beijing, China
manzhang@buaa.edu.cn

Abstract—Due to their widespread use in industry, several techniques have been proposed in the literature to fuzz REST APIs. Existing fuzzers for REST APIs have been focusing on detecting crashes (e.g., 500 HTTP server error status code). However, security vulnerabilities can have major drastic consequences on existing cloud infrastructures.

In this paper, we propose a series of novel automated oracles aimed at detecting violations of access policies in REST APIs. These novel automated oracles can be integrated into existing fuzzers, in which, once the fuzzing session is completed, a “security testing” phase is executed to verify these oracles.

Our novel techniques are integrated as an extension of EVOMASTER, a state-of-the-art fuzzer for REST APIs. Experiments are carried out on a series of artificial examples and 13 real-world REST APIs. Results show that our novel oracles and their automated integration in a fuzzing process can lead to detect security issues in some of these APIs.

Index Terms—REST, API, fuzzing, security, BOLA, BFLA

I. INTRODUCTION

The Open Worldwide Application Security Project (OWASP) keeps track of the most common security vulnerabilities in web applications.¹ In their 2023-report targeted at web APIs,² the most common vulnerability is *Broken Object Level Authorization* (BOLA). A similar vulnerability, called *Broken Function Level Authorization* (BFLA), ranks fifth. Wrong authorization checks in the APIs can lead users to access resources they are not supposed to access (BOLA), or do operations they are not have rights to (BFLA).

As of 2024, it is estimated³ that 94% of companies worldwide use cloud computing, with costs measured in the hundreds of billions of euros. As such, reliability and correctness of these systems is of paramount importance in modern society.

Currently, REST APIs are the most common type of APIs on the web. As testing the functionalities of APIs can be expensive and time consuming, lot of research has been carried out to automatically generate test cases for them [1]. However, to deal with authorization-related faults, a fuzzer should be able to handle the authentication of different users with

different roles and access rights. Unfortunately, many fuzzers in the literature are unable to handle authentication information during the fuzzing process. For the few fuzzers that can handle authentication of different users, they are unable to automatically detect BOLA and BFLA related vulnerabilities.

To fill this gap in the literature, in this paper we define three novel automated oracles aimed at detecting faults related to misconfigured authorization checks. We integrated our novel oracles in a state-of-the-art fuzzer for REST APIs, namely EVOMASTER [2]. Experiments on 3 artificial APIs widely used in security research, as well as on 10 real-world APIs, show that our novel automated oracles can be used to automatically detect 27 new security vulnerabilities that cannot be detected with other current approaches in the literature.

Our approach is integrated and evaluated in EVOMASTER, but any other state-of-the-art fuzzer that supports authentication of users (e.g., via authorization headers, or dynamic login tokens) could had been used for our study. We chose to use EVOMASTER as starting point for our implementation, as developing a robust fuzzer from scratch is a major engineering effort [3]. We chose EVOMASTER because it is among the oldest [4] and mature fuzzer still in active development [2], giving among the best results in large tool comparisons [5]–[7]. As it is a mature tool with extensive documentation, it is used in several enterprises, such as Meituan [8], [9] and Volkswagen [10], [11].

II. RELATED WORK

There is a large body of research related to fuzz REST APIs [1]. Several fuzzers have been proposed in the literature, including, in alphabetic order: APIRL [12], APIF [13], ARAT-RL [14], bBOXRT [15], DeepREST [16], EVOMASTER [2], LlamaRestTest [17], MINER [18], Morest [19], Nautilus [20], ResTest [21], RestCT [22], RESTler [23], RestTestGen [24], Schemathesis [25] and VoAPI2 [26]. These fuzzers aim at achieving high coverage (e.g., in terms of schema coverage and code coverage), and detecting faults. The most common type of oracle is detecting server errors [27], usually represented with a 500 HTTP status code [1]. Other types of oracles include checking for schema mismatches (i.e., the test API is returning data that is not valid according to the API’s schema), and robustness properties (i.e., sending wrong data on purpose, and then check whether the API correctly flags them as user

* Corresponding Author

¹<https://owasp.org/www-project-top-ten/>

²<https://owasp.org/www-project-api-security/>

³<https://edgedelta.com/company/blog/how-many-companies-use-cloud-computing>

errors). In large empirical comparisons [5], [6], fuzzers for REST APIs are typically compared on code coverage and on the number of detected 500 status codes.

When it comes to verify security properties in REST APIs, some initial work on this topic exists. One of the earlier approaches was in RESTler [28], in which 4 security rules were defined. These included checking the semantics of HTTP operations (e.g., “A resource that has been deleted must no longer be accessible” [28]) as well as checking user-namespace properties (e.g., “A resource created in a user namespace must not be accessible from another user namespace” [28]). However, this latter rule might be too restrictive, and not considering that there can be complex access control policies in place where some users might have access to some resources of other users (e.g., a typical case would be administrator users).

The work presented in Nautilus aims at detecting “vulnerabilities caused by improper user input handling” [20]. However, the presented approach is not fully automated, as it requires the testers to manually annotate the OpenAPI schemas with extra information needed by Nautilus to detect vulnerabilities.

VoAPI2 [26] does different kinds of security attacks, including SSRF, XSS, and Command Injection testing. Different techniques are used to identify subsets of the API operations and parameters that are more likely to be affected by these kinds of security issues.

Similarly, APIF [13] does different kinds of security attacks based on SecLists,⁴ like for example XSS, SQL Injection and SSRF. However, APIF is not able to handle OpenAPI schemas, and it requires first to capture live data via a reversed proxy.

“Mass assignment” is one of the most common security vulnerabilities in APIs, according to OWASP’2023 report.² This issue happens when an attacker can update object properties that they should not have access to. The authors of RestTestGen [24] provided an extension to it to enable detection of this kind of vulnerability [29].

The work in [30] presents an LLM-based approach to generate tests aimed at Broken Object Level Authorization (BOLA) vulnerabilities in REST APIs. However, it is unclear how the automated oracles are designed. In particular, “this does not represent the detection of actual BOLA vulnerabilities ... The potential for generating tests that incorrectly identify non-existent vulnerabilities needs further investigation” [30].

An alternative approach to detect BOLA vulnerabilities in REST APIs has been presented in [31]. There, OpenAPI schemas are transformed into Petri nets, and then server logs are used to analyze possible BOLA vulnerabilities. However, a limitation is that it “requires having an OpenAPI specification with links. Unfortunately, this field is not widely used” [31].

Our work is significantly different from this existing literature. In this paper, we define a series of automated oracles aimed at detecting faults related to access control. Given a fuzzer that generates a set of test cases maximizing different

criteria (e.g., schema and code coverage), we apply a post-processing strategy to generate new test cases based on those, aimed at checking our automated oracles. The only requirements are the presence of an OpenAPI schema describing the API, and login information for at least two different users. No further manual effort or custom schema annotation is required. As far as we know, none of these automated oracles presented in this paper have been proposed and evaluated in the literature so far.

III. SECURITY ORACLES

In this paper, we define three new automated oracles to detect different kinds of faults related to authorization and access control. To detect these faults, we need to generate test cases with specific scenario patterns, and then verify if the HTTP status codes we get back from the tested API might reveal the presence of these faults.

To enable these security oracles, authentication information for at least two different users are required by the fuzzer. Better if more (at least one per different role), especially considering different access policies they might have (e.g., administrators). No assumption on their relative access policies is needed, and no formal description of these policies is needed either. Such information is inferred dynamically based on the 401 (not authenticated) and 403 (not authorized) HTTP status codes received back during the fuzzing session. The functionalities of an API can be expressed with an OpenAPI⁵ schema, specifying which endpoints (combination of URL *paths* and HTTP *verbs*) are available, and what type of data they expect as input (e.g., query parameters and JSON body payloads). However, OpenAPI schemas are not able to express information about access policies.

To simplify the explanations of how our novel automated oracles work, in our examples we will provide HTTP call sequences in the following format:

```
(AUTH) VERB PATH -> CODES
```

Like for example a test case T_k could be:

```
(A) POST /items -> 201
(A) GET /items/42 -> 200
(B) GET /items/42 -> 403
```

In parentheses we have the user (where “?” means it does not matter), followed by the verb (e.g., GET) and the URL path, with then the resulting HTTP status code(s) of making this call. For simplicity, here we are omitting any query parameters, headers and body payloads. How those are handled will be discussed next. In this 3-step test case example T_k , a user A creates (status 201) a new item via a POST request, and then successfully accesses it with a GET request (status 200). However, another user B fails to access the same resource created by A (status 403) as they are not authorized.

To obtain calls on specific endpoints (e.g., POST /items) returning some specific HTTP status codes (e.g., 201), there might be the need for the right input data in the query

⁴<https://github.com/danielmiessler/SecLists>

⁵<https://swagger.io/specification/>

parameters, body payloads and headers. If some data is not correct (e.g., according to some defined constraints), then the API most likely would return a 4xx user error status code. As such data and constraints could be arbitrarily complex, it is not necessarily the case that a fuzzer is able to generate such HTTP calls. When evaluating our novel automated oracles, we need to create test scenarios with specific HTTP call sequences, where each call might require to return some specific HTTP status code. To achieve this, we use and copy valid test cases generated during the fuzzing session. If for example a fuzzer is left running for 1-hour, and it generates n test cases T (with $|T| = n$) as output (based on maximizing some coverage criteria), we use those n test cases T_i as our initial pool to construct our new security tests.

Assume we need a call on `GET /items/id` that returns a 200 status code. This might not be trivial to obtain, if the call to generate such resource (e.g., via a `POST` or a `PUT`) requires complex input data. At the end of the fuzzing session, we can check if in the set T there is any test case with such an HTTP call. Assume the previous example T_k was generated. That would fit our needs. However, all calls *after* our target calls would not be necessary. Those unnecessary calls can be *sliced* away. However, calls *before* the target might or might not be needed, as they might modify the state of the API. We cannot be sure. In this particular case, the first call `POST /items` creates the resource used in our target call `GET /items/id`. We can therefore make a copy C_k of T_k , and remove all calls after the target, but leave as they are all calls before. In this case, C_k would be:

```
(A) POST /items    -> 201
(A) GET /items/42  -> 200
```

When generating security tests that identify the presence of faults, it is of paramount importance that users are properly informed about it. EVOMASTER can output test cases in different formats, including Java, Kotlin, JavaScript and Python. When a test case identifies a security fault, we make sure that in the generated test cases a comment is generated and added before the HTTP call triggering/showcasing the fault.

A. Not Recognized Authentication

Typically, if a HTTP request is made without any authentication information, and the called endpoint requires authentication, then the API would answer with a 401 status code. If the user is authenticated, but, if they try to access a resource that does not belong to them, then the API should return a 403 status code (authenticated but not authorized).

If a user makes an authenticated call, but if then they receive a 401 status code (instead of for example a 403), then it is a fault. This is not a *security* vulnerability, but rather a *usability* fault. Based on the returned 401, the user might wrongly believe that their authentication info is wrong or outdated, and waste time in trying to resolve such issue (e.g., by contacting the IT support of the API). As such, it would be best to fix this kind of usability fault.

In a fuzzer, simply flag as fault any 401 returned status code with an authenticated user would be unwise, as it could create

many false-positives. For example, if a tester misconfigures the fuzzer, and provide a wrong user info, or such auth info has expired, then false-positive faults could be identified on each single API endpoint. Depending on the size of the API, this could result in hundreds of wrongly identified faults that could make harder to highlight any found real fault.

To avoid this kind of false-positives, we designed the following strategy. First, for each endpoint X in the OpenAPI schema, we check if the fuzzer has generated any test case T_1 with an authenticated user (e.g., A) in which a 401 was returned. If so, X is potentially faulty. We then check if authentication information for A was correct. To do so, we check if the fuzzer has generated any test T_2 in which A was used on an endpoint $Y \neq X$ in which a 2xx, success status code was returned (i.e., no 401 for A). This is a first step to check the validity of A , as Y could be an open endpoint in which no authentication check is required. To verify this, we check if in the generated tests of the fuzzer there is any test T_3 in which any call on Y resulted in either a 401 or 403. If so, it would mean that Y requires authentication (T_3), the credentials of A on Y are correct as we get a 2xx status code (T_2), while the call on X with A identifies a fault as a wrong 401 is returned (T_1).

For each endpoint in the OpenAPI schema, if these three conditions are met, then we create a new test $K = C_3, C_2, C_1$, where C_i is a copy of test T_i , where all unnecessary calls after the target are sliced away, as previously explained.

Figure 1 shows an example of generated test for an artificial API we developed to verify if our novel oracles can identify this kind of faults. Two users are set up: FOO and BAR. According to the internal access policies of the API, the user FOO has no rights to call `POST /api/resources/`. However, instead of getting a 403 response, the last call in that test returns a wrong 401. The previous call shows that the authentication info for FOO was correct, i.e., this detected fault is not a false-positive due to a misconfiguration of the authentication information provided to the fuzzer.

B. Existence Leakage

Assume that the fuzzing process generated at least one test case on an endpoint definition X in which a 403 status code is returned, and one test case in which a 404 is returned. The two test cases do not need to use the same or different users to get these results. For example, we could have X being something like `/data/id` where a `GET` call on `/data/42` returns a 403 status code, and a call on `/data/77` returns 404. These two calls could be done with the same user, or with two different users.

Such scenario is problematic, as it leads to information leakage on the existence of resources. If a user has no access to some resources (i.e., 403), they should not be able to learn which resources exist (403) and which ones do not exist (404), as such information can be used to narrow down follow up security attacks targeted at existing protected resources. The HTTP standards in RFC9110⁶ acknowledge the issue and

⁶<https://www.rfc-editor.org/rfc/rfc9110.html>

```

1 /**
2 * Calls:
3 * 1 - (201) PUT:/api/resources/{id}
4 * 2 - (403) PUT:/api/resources/{id}
5 * 3 - (201) PUT:/api/resources/{id}
6 * 4 - (401) POST:/api/resources/
7 * Found 1 potential fault of type-code 801
8 */
9 @Test @Timeout(60)
10 fun test_8() {
11
12     given().accept("*/")
13         .header("Authorization", "BAR") // BAR
14         .header("x-EMextraHeader123", "")
15         .put("${baseUrlOfSut}/api/resources/721")
16         .then()
17         .statusCode(201)
18         .assertThat()
19         .body(isEmptyOrNullString())
20
21     given().accept("*/")
22         .header("Authorization", "FOO") // FOO
23         .header("x-EMextraHeader123", "")
24         .put("${baseUrlOfSut}/api/resources/721")
25         .then()
26         .statusCode(403)
27         .assertThat()
28         .body(isEmptyOrNullString())
29
30     given().accept("*/")
31         .header("Authorization", "FOO") // FOO
32         .header("x-EMextraHeader123", "
33         _EM_11_XYZ_")
34         .put("${baseUrlOfSut}/api/resources/577")
35         .then()
36         .statusCode(201)
37         .assertThat()
38         .body(isEmptyOrNullString())
39
40     // Fault801. Wrongly Not Recognized as
41     // Authenticated. POST:/api/resources/
42     given().accept("*/")
43         .header("Authorization", "FOO") // FOO
44         .header("x-EMextraHeader123", "")
45         .post("${baseUrlOfSut}/api/resources/?
46         EMextraParam123=_EM_9_XYZ_")
47         .then()
48         .statusCode(401)
49         .assertThat()
50         .body(isEmptyOrNullString())
51 }

```

Fig. 1. Example of generated test for an artificial API in which a not-recognized authentication fault is correctly identified.

explicitly states: “An origin server that wishes to “hide” the current existence of a forbidden target resource MAY instead respond with a status code of 404 (Not Found)”. For non-authorized users, to avoid information leakage either the APIs should always return 403 or always 404, regardless of whether the resource exists or not.

However, there are cases in which a 404 is not leading to information leakage. This happens when the requested resource is a subresource that belongs to the user. Consider a new URL path Y , like for example $/data/id/bar$. Considering $/data/42/bar$ returning 403, having $/data/77/bar$ returning 404 with an authenticated user A does not necessarily

mean a leakage. It could happen for example that a GET on $/data/77$ returns 200, i.e., user A owns it and all its sub-resources like for example $/data/77/bar$. But what if $/data/77$ does not exist (404)? Or the user A has no access to it (403)? Or A is actually no user (i.e., making a call with no authentication information, which should had rather led to a 401)? Then, we would have an information leakage.

To test this oracle, for each GET endpoint in the OpenAPI we need to create a new test representing the following scenario:

```

(?) GET /P/X -> 403
(A) GET /P/X -> 404
(A) GET /P -> 403, 404

```

Here we have two steps, with an optional third step. First, for each GET endpoint we check if we have at least one test case (T_1) returning 403 and one (T_2) returning 404. If yes, we can instantiate this scenario. We first make a copy of T_1 (including all input data, like query parameters and body payloads), and slice away all calls after that target GET $/P/X$. A test case is composed of one or more HTTP calls. As previously discussed, calls after our target are not needed for our oracle. However, calls before the target might be required, as they might set the state of the API (e.g., by creating the resource). For example, T_1 could be something like:

```

(B) POST /data -> 201
(C) GET /data/42 -> 403
(B) GET /data/42 -> 200
(B) DELETE /data/42 -> 204

```

Then, after the slice, our copy C_1 would be:

```

(B) POST /data -> 201
(C) GET /data/42 -> 403

```

Then, we apply the same process to T_2 : we make a copy C_2 , where we slice all calls after the target. For example, assume we have T_2 being:

```

(A) GET /data/77 -> 404
(A) POST /data -> 201

```

Then, the copy C_2 would be just the first call. Then, we can create a new test K by combining C_1 and C_2 :

```

(B) POST /data -> 201
(C) GET /data/42 -> 403
(A) GET /data/77 -> 404

```

One possible issue here is that the execution of C_1 might have side effects on the execution of C_2 , as it might change the state of the API (e.g., creating new data in the database, if any). This is not a problem when executing tests in isolation, as state-of-the-art tools such as EVOMASTER can automatically reset the state of databases after executing each test [32]. This means that we must execute K , and verify that, for each call, we receive the same HTTP status codes as those returned in the original T_1 and T_2 tests. If the status codes remain the same, then we have detected a potential security-related fault.

However, we are not done, as we need to check if the 404 in the last call was valid, i.e., if the user owns any ancestor resource. If there is no authenticated user doing such call (i.e.,


```

1 /**
2 * Calls:
3 * 1 - (201) PUT:/api/resources/{id}
4 * 2 - (403) GET:/api/resources/{id}
5 * 3 - (404) GET:/api/resources/{id}
6 * Found 1 potential fault of type-code 800
7 */
8 @Test @Timeout(60)
9 fun test_10() {
10
11     given().accept("*/")
12         .header("Authorization", "BAR") // BAR
13         .header("x-EMextraHeader123", "")
14         .put("${baseUrlOfSut}/api/resources/12")
15         .then()
16         .statusCode(201)
17         .assertThat()
18         .body(isEmptyOrNullString())
19
20     given().accept("*/")
21         .header("Authorization", "FOO") // FOO
22         .header("x-EMextraHeader123", "")
23         .get("${baseUrlOfSut}/api/resources/12")
24         .then()
25         .statusCode(403)
26         .assertThat()
27         .body(isEmptyOrNullString())
28
29     // Fault800. Leakage Information Existence of
30     // Protected Resource. GET:/api/resources/{id}
31     given().accept("*/")
32         .header("Authorization", "BAR") // BAR
33         .header("x-EMextraHeader123", "")
34         .get("${baseUrlOfSut}/api/resources/339")
35         .then()
36         .statusCode(404)
37         .assertThat()
38         .body(isEmptyOrNullString())
39 }

```

Fig. 2. Example of generated test for an artificial API in which a leakage information fault is correctly identified.

in our example if A represents no user), then there is nothing more to check, and the fault is identified. Otherwise, given the called target endpoint, i.e., `/data/id` in this case, we check in the OpenAPI the top ancestor resource (in the URL path) that has a GET endpoint. If there is none, then there is nothing more to check, and the fault is identified. Otherwise, we create a new GET call on this ancestor resource using the same user authentication (e.g., A in this example). If we still get a 404, then the fault is identified. If not, the result remains inconclusive.

Figure 2 shows an example of generated test for an artificial API we developed to verify if our novel oracles can identify this kind of faults. Two users are set up: FOO and BAR. When FOO accesses a resource that belongs to BAR, they get a 403 unauthorized response. However, accessing a non-existing resource on the same endpoint leads to a 404, which leads to information leakage.

C. Missed Authorization Checks

Resources should be protected from unauthorized access. For example, a user of a bank should not be able to do transactions on the accounts belonging to other users. However,

access policies could be arbitrarily complex. Simply stating that a user should not access or manipulate resources of other users would just result in many false-positive faults.

The most obvious example is administrator users. A tester using a fuzzer could provide authentication information for some basic users as well as for an administrator one. This latter would be essential for testing endpoints that only administrators can have access to. An administrator would likely be able to access and manipulate resources of other users (albeit possibly still with some constraints). However, there are plenty of cases in which more complex scenarios would arise. For example, in a web application for discussion forums, the “moderator” for a specific forum might not have administration rights on the whole application. However, they might have special rights on all the messages created by other users in that forum (e.g., to delete them if they violate the terms of access of the application). Another example is websites related to children (e.g., government benefits or application for kindergarten), where groups of users (e.g., guardians for a child, typically their parents) share the same rights and resources, but cannot access the resources of other groups (i.e., other families). While these scenarios illustrate only a few examples, similarly complex access control rules can exist in various contexts.

Ideally, if there was a formal specification of these access policy rules, a fuzzer could use such specifications as automated oracle, i.e., by checking if any rule is violated during the fuzzing session (e.g., getting a resource with a 2xx whereas it was supposed to get a 403). Unfortunately, we are aware of no existing formal specification for access policies in REST APIs which is widely used in industry. Some languages exist, like for example XACML⁷ (used for example in [33]), but we have never encountered any REST API using those formal specifications for access policies.

Without a formal specification telling us what are the access policies for a REST API, how can a fuzzer identify that a resource should not have been accessed? For example, if a user A accesses resource X and they get a 200 HTTP status code, how can a fuzzer tell that being a fault (i.e., a 403 was expected with no returned data)? This is an instance of the more general oracle problem: given $f(x) = y$, how can we be sure that y is the correct output for software f given the input x ? If f crashes, then clearly we have identified a fault for input x . But what about all other cases? The “oracle problem” is indeed one of the major challenges in software testing research [34].

In our particular case, we use the following strategy to address the oracle problem. In REST APIs, there are three distinct HTTP operators that can manipulate a specific resource: PUT (for whole updates), PATCH (for partial updates) and DELETE (for removing the resource). Our *hypothesis* is that, if any two of these operations is forbidden (i.e., 403), but the third is allowed (i.e., 2xx), then it is most likely a misconfiguration of the API. For example, an authorization check on an endpoint could have been forgotten. This would

⁷<https://groups.oasis-open.org>

be a *major* security vulnerability, with possibly catastrophic consequences.

The intuition about this oracle is rather straightforward. Would it make any sense for a user to be forbidden to do any modification on a resource (e.g., with PUT and PATCH), but then be allowed to just delete it from the system with a DELETE? Likewise, would it make any sense that a user is not allowed to DELETE a resource, but then be allowed to do any kind of changes on it that they want with a PUT or PATCH? Or could it be just that an authorization check is wrongly missing? Among all possible APIs and different kinds of access policy rules, there could be cases in which these scenarios could happen and be correct. As such, we cannot exclude the presence of some false-positives when using these rules as automated oracles. Still, as they can be used to detect critical security faults, a non-zero amount of false-positive identified faults could be tolerable in practical contexts.

Our approach works as following. For each path X and each one of the verb V such as DELETE, PUT and PATCH, we check if it exists in the OpenAPI schema (not all defined paths might support all different HTTP verbs, such as for example PATCH). If so, we check in the generated tests T of the fuzzer if there is any T_k for which on that verb/path $V : X$ we get a 403 response. If T_k exists, we make a copy C_k in which all calls after the 403 are sliced away. If not, we try to create it. This is done by first checking for a test T_c that creates the resource at X , and then add new call with a different user executing $V : X$ on it (due to space constraints, we cannot provide full details of all the edge cases dealing with such resource creation; we refer the interested reader to check our open-source implementation). If this latter call returns a 403, then such instantiated test will be our C_k . If not, our oracle is not applicable for $V : X$.

Given C_k in which the last call returns a 403 with a user A on path X for verb V , we need to add a new call $V' : X$ with same user A with a different verb V' out of the considered three. For example, if V is DELETE, for V' we consider PUT and PATCH. As making a success call with V' might require specific input data (e.g., query parameters and body payloads), we check if in T there is any existing T_j satisfying all these criteria. If not, we cannot apply our oracle on this path X . Otherwise, we make a copy C_j of T_j , where all calls *before* and after the target $V' : X$ are sliced away, i.e., C_j is composed of only a single call. Then, there are two further steps.

First, in C_j we modify the authentication information to rather use A (in case it was using a different user). Second, we need to make sure that both C_k and C_j handle the same resource. For example, C_k could deal with `/items/42` whereas C_j with `/items/77`. We create a new test Z which is the concatenation of C_k and C_j , where the resource path of C_j is *bound* to be the same as in C_k . For example, given C_k being:

```
(B) POST /items -> 201
(A) DELETE /items/42 -> 403
```

and C_j being:

```
(C) PUT /items/66 -> 200
```

then Z would be:

```
(B) POST /items -> 201
(A) DELETE /items/42 -> 403
(A) PUT /items/42 -> 200
```

The binding is done by either modifying the variables in the path elements of the endpoint URL, or by using the same dynamic updates used in the tests when resources are generated dynamically. For example, a resource could be created with a POST request, where the id of the newly generated resource could be dynamic and non-deterministic. The id could be then returned in the response of the POST request, e.g., in the `location` header or as a field in the body payload. Such id needs to be extracted, and then used in the following HTTP calls for referring to the same resource. To achieve this, we simply rely on the existing algorithms in EVOMASTER to bind endpoints on same resources [2].

Once Z is created, we execute it, and verify if the status codes are as expected, i.e., the DELETE returns a 403, followed by a PUT or PATCH on same resource with same user returning a 2xx. If this happens, then we have identified a potential security vulnerability.

Figure 3 shows an example of generated test for an artificial API we developed to verify if our novel oracles can identify this kind of faults. Two users are set up: `FOO` and `BAR`. `BAR` creates a resource via a POST, and they can modify it with a PUT. A different user `FOO` is forbidden from deleting this resource (403 on DELETE). However, this user can completely replace the resource with a PUT (as it gives a 204).

IV. FUZZER INTEGRATION

To use our novel oracles, we need to create specific scenarios, and then verify if any fault is detected based on the HTTP status codes we receive back from the tested API. To check and verify authentication and authorization properties, we need to be able to generate test cases that can exercise all the different functionalities of the API. Input data can have complex constraints (e.g., in query parameters and body payloads), and so test cases that trivially return 400 user errors would not be helpful for our goals.

The integration of our approach into existing fuzzers is as follows. We apply our security validation as a *post-processing* phase. We let a fuzzer run for the amount of time specified by the users. In academic experiments, this is typically 1 hour [1]. Within 1-hour, a fuzzer could evaluate hundreds of thousands of HTTP calls. However, most likely no human test engineer would be interested in analyzing and using test suites with hundreds of thousands of test cases. It is simply not feasible. As such, fuzzers typically provide as output a minimized test suites, aimed at maximizing different criteria. These could be about covering different black-box aspects of the API [35] (e.g., based on the OpenAPI schema, like covering different HTTP status code for each defined endpoint), or code coverage metrics in white-box testing [36].

```

1 /**
2 * Calls:
3 * 1 - (201) POST:/api/forbiddendelete/resources
4 * 2 - (204) PUT:/api/forbiddendelete/resources/{
5   id}
6 * 3 - (403) DELETE:/api/forbiddendelete/resources
7   /{id}
8 * 4 - (204) PUT:/api/forbiddendelete/resources/{
9   id}
10 * Found 1 potential fault of type-code 802
11 */
12 @Test @Timeout(60)
13 fun test_12() {
14
15     var location_resources : String? = ""
16     var location_id : String? = ""
17
18     val res_0: ValidatableResponse = given().
19     accept("*/")
20     .header("Authorization", "BAR") // BAR
21     .header("x-EMextraHeader123", "")
22     .post("${baseUrlOfSut}/api/
23     forbiddendelete/resources")
24     .then()
25     .statusCode(201)
26     .assertThat()
27     .body(isEmptyOrNullString())
28     location_resources = res_0.extract().header("
29     location")
30     assertTrue(isValidURIorEmpty(
31     location_resources));
32
33     given().accept("*/")
34     .header("Authorization", "BAR") // BAR
35     .header("x-EMextraHeader123", "")
36     .put(resolveLocation(location_resources,
37     baseUrlOfSut + "/api/forbiddendelete/
38     resources/975"))
39     .then()
40     .statusCode(204)
41     .assertThat()
42     .body(isEmptyOrNullString())
43
44     given().accept("*/")
45     .header("Authorization", "FOO") // FOO
46     .header("x-EMextraHeader123", "")
47     .delete(resolveLocation(
48     location_resources, baseUrlOfSut + "/api/
49     forbiddendelete/resources/975"))
50     .then()
51     .statusCode(403)
52     .assertThat()
53     .body(isEmptyOrNullString())
54
55     // Fault802. Forbidden Delete But Allowed
56     Modifications. PUT:/api/forbiddendelete/
57     resources/{id}
58     val res_3: ValidatableResponse = given().
59     accept("*/")
60     .header("Authorization", "FOO") // FOO
61     .header("x-EMextraHeader123", "")
62     .put(resolveLocation(location_resources,
63     baseUrlOfSut + "/api/forbiddendelete/
64     resources/975"))
65     .then()
66     .statusCode(204)
67     .assertThat()
68     .body(isEmptyOrNullString())
69     location_id = res_3.extract().header("
70     location")
71     assertTrue(isValidURIorEmpty(location_id));
72 }

```

Fig. 3. Example of generated test for an artificial API in which a missed authorization check fault is correctly identified.

After the fuzzing session is over, the fuzzer would have generated a set T of test cases. These tests are then used as a base to create new test scenarios, as previously described in Section III. As such, how these T tests are created is not so important, and any state-of-the-art fuzzer could be used, as long as it can generate tests using authentication information.

For our experiments, we used EVOMASTER, as it fits those constraints. However, one limitation we noticed is that it was not good at generating tests in which 403 unauthorized accesses were detected. Given a set of authentication information as input to the fuzzer (besides a copy of the OpenAPI schema for the tested API), EVOMASTER can generate tests with and without authentication info. However, calls with different users on the same resources (which could lead to 403 errors) do not seem to be created. As such, we needed to make sure to try to create such kind of tests before applying our novel oracles, as some of those rely on existing tests with 403 responses to create the needed test scenarios.

For each verb/path $V : X$ in the OpenAPI schema, we check if we have any test case generated in T in which such a call returns a 403 (unauthorized). If not, we try to create a new test for it and add it to T . If there is no call in T with a 401 (not authenticated), then most likely there is no authentication mechanism on that endpoint, and it can be skipped. Otherwise, we then check if there is any test T_a for which a successful 2xx response is obtained with any authenticated user A . If so, we make a copy C_a in which all calls after the target are sliced away. Then, for each other defined user B , we create a copy C_b of C_a , and we duplicate the last call in C_b by replacing A with B . For example, if we have C_a being:

```

(A) POST /data -> 201
(A) GET /data/42 -> 200

```

then C_b would be:

```

(A) POST /data -> 201
(A) GET /data/42 -> 200
(B) GET /data/42 -> 403

```

If executing C_b returns a 403 from the last call, then we are done for $V : X$, and C_b can be added to T . Otherwise, we repeat this process for the other users (if there is more than two) and for any other tests T_a that meet those criteria.

The number of new test cases we generate to evaluate security properties in our post-processing phase is linearly dependent on the number of endpoints in the tested API. Typically, these are at most a few hundreds. As such, creating and evaluating these security tests takes just a few seconds, and typically less than a minute. In the context of fuzzing an API for hours (e.g., between 1 and 24 hours), the overhead of our novel techniques is practically negligible.

V. EMPIRICAL STUDY

In this paper, we carried out an empirical study aimed at answering the following research questions:

RQ1: Are our novel techniques able to identify injected faults in artificial examples?

TABLE I
STATISTICS OF THE EMPLOYED REST APIs IN OUR EMPIRICAL STUDY FOR BLACK-BOX TESTING, INCLUDING NUMBER OF SOURCE FILES, NUMBERS OF LINES OF CODE (LOCs), AND THE NUMBER OF HTTP ENDPOINTS.

| SUT | #SourceFiles | #LOCs | #Endpoints |
|----------------|--------------|-------|------------|
| <i>capital</i> | 79 | 3075 | 17 |
| <i>crapi</i> | 164 | 14274 | 39 |
| <i>webgoat</i> | 369 | 14927 | 203 |
| Total 3 | 612 | 32276 | 259 |

TABLE II
STATISTICS OF THE EMPLOYED REST APIs IN OUR EMPIRICAL STUDY FOR WHITE-BOX TESTING, INCLUDING NUMBER OF SOURCE FILES, NUMBERS OF LINES OF CODE (LOCs), AND THE NUMBER OF HTTP ENDPOINTS.

| SUT | #SourceFiles | #LOCs | #Endpoints |
|---------------------------------|--------------|--------|------------|
| <i>blogapi</i> | 89 | 4787 | 52 |
| <i>familie-ba-sak</i> | 1089 | 143556 | 183 |
| <i>market</i> | 124 | 9861 | 13 |
| <i>ocvn</i> | 526 | 45521 | 258 |
| <i>pay-publicapi</i> | 377 | 34576 | 10 |
| <i>proxyprint</i> | 73 | 8338 | 74 |
| <i>reservations-api</i> | 39 | 1853 | 7 |
| <i>scout-api</i> | 93 | 9736 | 49 |
| <i>tiltaksgjennomforing-api</i> | 472 | 27316 | 79 |
| <i>tracking-system</i> | 87 | 5947 | 67 |
| Total 10 | 2969 | 291491 | 792 |

RQ2: Are our novel techniques able to identify faults in example APIs used in the security literature?

RQ3: What security faults can be found in existing real-world APIs?

A. API Selection

To answer **RQ1**, we created 3 small APIs, using Java and SpringBoot. In each one, we injected a security fault for each of our novel oracles. These APIs are simple, with no complex input constraints, or code execution flow depending on database state. Faults might not be detected if the code in which they reside is never executed due to the complexity of generating the right data. Our goal here in **RQ1** is to study the feasibility of our novel techniques without confounding factors. For reason of space, we cannot provide here full code of these 3 example APIs. They are provided open-source with our extension to EVOMASTER.

To answer **RQ2**, we looked at the security research literature for existing APIs used for demonstrating different types of security vulnerabilities. After a search in popular resources such as OWASP, we selected three widely used APIs: *capital*,⁸ *crapi*⁹ and *webgoat*.¹⁰ Table I shows some statistics on these 3 APIs. Note that *crapi* is a microservice running different services. All are run in our experiments. We fuzz the whole application through its API Gateway.

⁸<https://github.com/Checkmarx/capital>

⁹<https://github.com/OWASP/crAPI>

¹⁰<https://github.com/WebGoat/WebGoat>

To answer **RQ3**, we selected an existing corpus of REST APIs used in several studies in API fuzzing research, called EMB [37]. We selected *all* APIs in its most recent version¹¹ that require authentication. Table II shows some statistics on these 10 APIs.

In total, for our experiments in this paper we used 16 APIs, for a total of more 300 thousand lines of code (for business logic, not including third-party libraries), and more than 1 000 endpoints.

B. Experiments Setup

The 3 artificial examples used for answering **RQ1** are small and simple. As such, experiments were run with our extension of EVOMASTER for just 1 minute, 10 times, for a total of 30 minutes.

Experiments for **RQ2** were run using *black-box* mode of EVOMASTER. On each of these 3 APIs, EVOMASTER was run for 1 hour. Experiments were repeated 10 times per API, for a total of 30 hours.

As all the APIs used for **RQ3** are written in Java, to answer this research question we use the *white-box* mode of EVOMASTER. We use the same time budget and repetition settings as in **RQ2**, i.e., EVOMASTER was run for 1 hour, with each experiments repeated 10 times. This resulted in a duration of 100 hours.

Our novel oracles can work both for black-box and white-box testing. We use different settings in **RQ2** and **RQ3** to show this being indeed the case. In total, our experiments took more than 130 hours, i.e., more than 5 days if run in sequence.

C. Results for RQ1

The example shown in Section III were based on these artificial APIs, i.e., the generated tests shown in Figure 1, Figure 2 and Figure 3. In every single experiment for **RQ1**, the injected faults were correctly identified. When there are no complex input constraints that prevent reaching the execution of the faulty code, our novel oracles are able to correctly identify all the security vulnerabilities that we have manually injected.

RQ1: On simple APIs with injected faults, our techniques are able to reliably identify those security vulnerabilities.

D. Results for RQ2

Table III shows the results combined for both **RQ2** and **RQ3**. Besides the results for our novel oracles, we also report what obtained with current oracles in EVOMASTER. As previously discussed, those are based on returned HTTP 500 status codes, and mismatches of the responses with what declared in the OpenAPI schemas.

In all the 3 security example APIs (i.e., *capital*, *crapi* and *webgoat*) we identified security faults with our novel oracles. In *crapi* and *webgoat* we could only identify faults of type *Not-Recognized* (Section III-A). For *capital* we could identify both *Leakage* (Section III-B) and *Missed-Check* (Section III-C).

¹¹<https://github.com/WebFuzzing/EMB>

TABLE III

DETECTED FAULTS, PER API, AVERAGED BY NUMBER OF RUNS (10). WE REPORT FAULTS BASED ON THE EXISTING ORACLES IN EVO MASTER, INCLUDING CHECKING FOR RETURNED HTTP 500 STATUS CODES (*H500*), AND FAULTS RELATED TO MISMATCHED RESPONSES BASED ON WHAT DECLARED IN THE OPENAPI SCHEMAS (*Schema*). RESULTS FOR OUR THREE NOVEL ORACLES ARE LABELLED AS *Not-Recognized* (SECTION III-A), *Leakage* (SECTION III-B) AND *Missed-Check* (SECTION III-C).

| SUT | H500 | Schema | Not-Recognized | Leakage | Missed-Check |
|---------------------------------|-------|---------|----------------|---------|--------------|
| <i>blogapi</i> | 39.2 | 50.6 | 0.0 | 0.0 | 0.0 |
| <i>capital</i> | 0.8 | 6844.9 | 0.0 | 1.0 | 0.7 |
| <i>crapi</i> | 12.1 | 224.5 | 1.0 | 0.0 | 0.0 |
| <i>familie-ba-sak</i> | 277.0 | 0.0 | 11.9 | 0.0 | 0.0 |
| <i>market</i> | 6.8 | 60.6 | 0.0 | 0.0 | 0.0 |
| <i>ocvn</i> | 282.1 | 527.4 | 0.0 | 0.0 | 0.0 |
| <i>pay-publicapi</i> | 10.0 | 44.8 | 0.0 | 0.0 | 0.0 |
| <i>proxyprint</i> | 39.8 | 99.0 | 4.2 | 2.0 | 0.0 |
| <i>reservations-api</i> | 4.4 | 1564.2 | 0.0 | 0.0 | 0.0 |
| <i>scout-api</i> | 75.9 | 0.0 | 0.1 | 0.0 | 0.0 |
| <i>tiltaksgjennomforing-api</i> | 30.4 | 0.0 | 0.0 | 0.0 | 0.0 |
| <i>tracking-system</i> | 2.0 | 804.9 | 0.0 | 0.0 | 0.0 |
| <i>webgoat</i> | 32.0 | 2192.2 | 5.0 | 0.0 | 0.0 |
| Sum | 812.4 | 12413.2 | 22.2 | 3.0 | 0.7 |

Note that faults are not always found in all experiments. For example, the value 0.7 for *Missed-Check* in *capital* means that, most likely, 1 fault was found in 7 out of 10 runs.

Faults can only be found if they exist. We have not manually checked the code and documentation of those 3 APIs to see if there are any other faults that could had been found with our oracles, or if what found is all that is there. Still, the fact that our novel techniques could generate test cases that can detect these faults show that they can be of practical value.

RQ2: All 3 of our novel oracles were effective at detecting faults in vulnerable APIs used as security examples.

E. Results for RQ3

When it comes to real-world APIs, in Table III we can see that security faults are found in 3 APIs, namely *familie-ba-sak*, *proxyprint* and *scout-api*. However, no fault of type *Missed-Check* (Section III-C) was found. This is the most critical type of faults among the ones we can detect with our novel techniques.

An API like *familie-ba-sak* is large and complex (Table II). It is made by the public administration in Norway, dealing with “child benefit case processing”, used by millions of people.¹² Finding a *Missed-Check* (Section III-C) would had rather been unsettling, and we are glad none was found. (If any was found, we would have contacted the Norwegian authorities before making this paper public).

Interestingly, there were several cases of *Not-Recognized* (Section III-A). Figure 4 shows an example of generated test detecting a *Not-Recognized* (Section III-A) fault. This automatically generated test case starts with 2 calls to the authentication service, to get dynamic authentication tokens for the users *Beslutter* (Line 13) and *TaskRunner* (Line 19). The first call on `/api/task/logg/410` with *Beslutter* shows that such endpoint requires authentication/authorization, as a 401 status code is returned (Line 31). A second call on

same endpoint with *TaskRunner* gets a 200 status code (Line 40). This provides evidence that the credentials for *TaskRunner* are not wrongly set. However, a call with same user on `/api/ekstern/pensjon/hent-barnetrygd` returns a 401 (Line 59), which would imply the user is not recognized. Based on previous call, this is wrong. Note that, even if access tokens might have a short life-span, they are dynamically retrieved in the same test case, and, as such, after a few (milli)seconds they should still be valid. Interestingly, the error message (in Norwegian at Line 63) seems to support the fact that this is a bug (i.e., a 403 should had been returned), as it states, translated with Google Translate: “Pension service can only be called by pension or logged in user with ADMINISTRATOR role”.

scout-api provides another interesting example. Consider the value 0.1 in Table III for *Not-Recognized*. It would seems a fault was found only in 1 out of 10 runs of the experiments. As discussed in Section IV, to function correctly our novel oracles rely on the tests generated by the fuzzer. If no test case was generated that can execute all the endpoints with the APIs resulting in different returned status codes, then our novel oracles would have little to work on, and so they would be ineffective. On the other hand, this also means that improvements in fuzzing technologies, achieving higher code and schema coverage, could result in detecting new security vulnerabilities with our novel oracles.

Table III shows that, on the 3+10 analyzed APIs, in total our oracles detected 27 distinct faults (although not all were found in all runs, recall the discussion about *scout-api* for example). This provides practical value for our novel techniques. However, compared to the other existing oracles used in EVO MASTER, this is a drop in a bucket. More than 800 faults related to 500 HTTP status codes were identified, and more than 12 000 faults related to schema mismatches. This date requires some more discussion.

Given N distinct endpoints, data could be found to trigger 500 status codes in all of them, leading to detecting N

¹²<https://github.com/navikt/familie-ba-sak>

```

1 /**
2 * Calls:
3 * 1 - (401) GET:/api/task/logg/{id}
4 * 2 - (200) GET:/api/task/logg/{id}
5 * 3 - (401) POST:/api/ekstern/pensjon/hent-barnetrygd
6 * Found 1 potential fault of type-code 801
7 * Using 1 example:
8 * 2020-12-01
9 */
10 @Test @Timeout(60)
11 public void test_392() throws Exception {
12
13     final String token_Beslutter = "Bearer " + given()
14         .contentType("application/x-www-form-urlencoded")
15         .body("name=Beslutter&grant_type=client_credentials&code=foo&client_id=foo&client_secret=secret")
16         .post("http://localhost:10162/azuread/token")
17         .then().extract().response().path("access_token");
18
19     final String token_TaskRunner = "Bearer " + given()
20         .contentType("application/x-www-form-urlencoded")
21         .body("name=TaskRunner&grant_type=client_credentials&code=foo&client_id=foo&client_secret=secret")
22         .post("http://localhost:10162/azuread/token")
23         .then().extract().response().path("access_token");
24
25
26     given().accept("*/*")
27         .header("x-EMextraHeader123", "")
28         .header("Authorization", token_Beslutter) // Beslutter
29         .get(baseUrlOfSut + "/api/task/logg/410?page=281")
30         .then()
31         .statusCode(401)
32         .assertThat()
33         .contentType("text/html");
34
35     given().accept("*/*")
36         .header("x-EMextraHeader123", "42")
37         .header("Authorization", token_TaskRunner) // TaskRunner
38         .get(baseUrlOfSut + "/api/task/logg/62")
39         .then()
40         .statusCode(200)
41         .assertThat()
42         .contentType("application/json")
43         .body("'data'", nullableValue())
44         .body("'status'", containsString("FEILET"))
45         .body("'melding'", containsString("Henting av tasklogg feilet."))
46         .body("'frontendFeilmelding'", containsString("Henting av tasklogg feilet."));
47
48     // Fault801. Wrongly Not Recognized as Authenticated. POST:/api/ekstern/pensjon/hent-barnetrygd
49     given().accept("application/json")
50         .header("x-EMextraHeader123", "")
51         .header("Authorization", token_TaskRunner) // TaskRunner
52         .contentType("application/json")
53         .body("{ \" +
54             \" \"ident\": \"5sJKKQCh2\", \" +
55             \" \"fraDato\": \"2020-12-01\" \" +
56             \" } \"")
57         .post(baseUrlOfSut + "/api/ekstern/pensjon/hent-barnetrygd")
58         .then()
59         .statusCode(401)
60         .assertThat()
61         .contentType("application/json")
62         .body("'servlet'", containsString("dispatcherServlet"))
63         .body("'message'", containsString("Pensjon tjeneste kan kun kalles av pensjon eller innlogget bruker med FORVALTER rolle"))
64         .body("'url'", containsString("/api/ekstern/pensjon/hent-barnetrygd"))
65         .body("'status'", containsString("401"));
66 }

```

Fig. 4. Example of generated test case in which a *Not-Recognized* (Section III-A) fault is identified in *familie-ba-sak*.

distinct faults. However, in the same endpoint, there could be several distinct faults, all leading to a 500 response. To detect the differences among them, EVOMASTER in *white-box* mode uses a further heuristic, which is to consider the last executed line of code in the business logic of the API when a 500 is returned. Often, this identifies the line of code where an exception was thrown. EVOMASTER then uses the combination of endpoint and last-executed-line to identify distinct faults. As such, the number of detected faults could be higher than N .

However, a 500 status does not really mean that the API crashed. If it crashed, then it would not be able to send back a HTTP response with status 500, and the TCP socket would have been closed. Typically, when an exception is thrown, then the framework (e.g., Spring for Java) running the API catches the exception, and sends back an error response. The API will keep running. If the crash is due to invalid input, then the fault is related to improper input validation, and rather a 4xx status error (indicating an user error) should have been returned. A typical example is assuming a requested resource to exist, and crashing on a null-pointer exception instead of returning a 404 not found user error status [27]. This is still a fault that needs to be fixed, but it is not as critical as it might look at a first sight.

Similarly, schemas are often considered as documentation. Documentation are often incomplete, especially if they are automatically generated from the source with some tools (e.g., SpringDoc). For example, for each endpoint, a schema should specify all possible expected type of responses (e.g., status codes), not only for successful responses. In the case of a returned 404 status code, it would be flagged as a schema fault for any endpoint for which 404 is not declared as possible response. Even in the case of errors, a response could contain a body payload describing the error (e.g., see example in Figure 4). To be able to be processed by the client user, the structure of such error messages should be specified in the schema.

Another potential issue is that, by default, all types defined in OpenAPI schemas are not-nullable, unless explicitly marked with a `nullable:true` setting. Returning a body payload with some null fields will result in a schema fault in such cases, *for each* field. And so on. Here, the fix could be needed in the schema or in the API, depending on whether such fields are supposed to be nullable or not. If a schema is only used for documentation for human users, it is easy to miss many misconfigurations and misalignment between the schema and the API, unless one is an expert in OpenAPI schemas. However, as soon as an automated tool can check these properties dynamically (e.g., a fuzzer), then thousands of those issues can be found, especially in APIs with a large number of endpoints.

Not all faults have the same severity. In this paper, we have proposed 3 novel oracles, but we cannot say how critical they are for certain, unless empirical studies or interviews/surveys with practitioners were carried out to assess their assumed importance. However, we argue that a security vulnerability

that leads to unauthorized access of resources (e.g., *Missed-Check*) is likely much more important than wrongly returning a 500 status code instead of a 404.

RQ3: *Our novel security oracles were effective at finding new faults in existing real-world APIs.*

VI. THREATS TO VALIDITY

Our experiments rely on randomized algorithms. To take their randomness into account, experiments were repeated 10 times.

In our empirical study, 16 APIs were used (3 toy examples, 3 examples for security research, and 10 real-world). A larger case study would be needed to be able to generalize our results, especially for APIs developed in industry.

Our novel oracles are not specific nor tailored to any fuzzer. In our experiments, we used EVOMASTER, but in theory any other state-of-the-art fuzzer could have been used. How easy or difficult would it be for our novel techniques to be integrated in other fuzzers is a matter that would require further investigation.

VII. CONCLUSION

In this paper, we have presented 3 novel oracles aimed at detecting authorization-related faults in REST APIs. These oracles were used to detect 27 new faults in an empirical study consisting of 13 real-world APIs.

Our novel techniques were implemented as an extension of EVOMASTER, a state-of-the-art fuzzer. Once our novel techniques detect new faults, executable test cases are generated to help practitioners in industry to reproduce the faults and debug them.

Future work will aim at designing more automated oracles to identify different kinds of security faults in REST APIs, and carry out empirical studies in industry.

Our extension to EVOMASTER is released as open-source at www.evomaster.org.

ACKNOWLEDGMENT

This work is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EAST project, grant agreement No. 864972). Man Zhang is supported by the State Key Laboratory of Complex & Critical Software Environment (SKLCCSE, grant No. CCSE-2024ZX-01).

REFERENCES

- [1] A. Golmohammadi, M. Zhang, and A. Arcuri, "Testing restful apis: A survey," *ACM Transactions on Software Engineering and Methodology*, aug. 2023. [Online]. Available: <https://doi.org/10.1145/3617175>
- [2] A. Arcuri, M. Zhang, S. Seran, J. P. Galeotti, A. Golmohammadi, O. Duman, A. Aldasoro, and H. Ghianni, "Tool report: Evomaster—black and white box search-based fuzzing for rest, graphql and rpc apis," *Automated Software Engineering*, vol. 32, no. 1, pp. 1–11, 2025.
- [3] A. Arcuri, M. Zhang, A. Belhadi, B. Marculescu, A. Golmohammadi, J. P. Galeotti, and S. Seran, "Building an open-source system test generation tool: lessons learned and empirical analyses with evomaster," *Software Quality Journal*, pp. 1–44, 2023.
- [4] A. Arcuri, "RESTful API Automated Test Case Generation," in *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 9–20.

- [5] M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated Test Generation for REST APIs: No Time to Rest Yet," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 289–301. [Online]. Available: <https://doi.org/10.1145/3533767.3534401>
- [6] M. Zhang and A. Arcuri, "Open Problems in Fuzzing RESTful APIs: A Comparison of Tools," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, may 2023. [Online]. Available: <https://doi.org/10.1145/3597205>
- [7] H. Sartaj, S. Ali, and J. M. Gjøby, "Rest api testing in devops: A study on an evolving healthcare iot application," 2024. [Online]. Available: <https://arxiv.org/abs/2410.12547>
- [8] M. Zhang, A. Arcuri, Y. Li, Y. Liu, and K. Xue, "White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–38, 2023.
- [9] M. Zhang, A. Arcuri, P. Teng, K. Xue, and W. Wang, "Seeding and mocking in white-box fuzzing enterprise rpc apis: An industrial case study," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2024–2034.
- [10] A. Poth, O. Rjollli, and A. Arcuri, "Technology adoption performance evaluation applied to testing industrial rest apis," *Automated Software Engineering*, vol. 32, no. 1, p. 5, 2025.
- [11] A. Arcuri, A. Poth, and O. Rjollli, "Introducing black-box fuzz testing for rest apis in industry: Challenges and solutions," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2025.
- [12] M. Foley and S. Maffei, "Apirl: Deep reinforcement learning for rest api fuzzing," in *Thirty-ninth Conference on Artificial Intelligence (AAAI 2025)*, 2025.
- [13] Y. Wang and Y. Xu, "Beyond rest: Introducing apif for comprehensive api vulnerability fuzzing," in *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, 2024, pp. 435–449.
- [14] M. Kim, S. Sinha, and A. Orso, "Adaptive rest api testing with reinforcement learning," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 446–458.
- [15] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A black box tool for robustness testing of rest services," *IEEE Access*, vol. 9, pp. 24 738–24 754, 2021.
- [16] D. Corradini, Z. Montolli, M. Pasqua, and M. Ceccato, "Deeprest: Automated test case generation for rest apis exploiting deep reinforcement learning," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1383–1394.
- [17] M. Kim, S. Sinha, and A. Orso, "Llamaresttest: Effective rest api testing with small language models," in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2025.
- [18] C. Lyu, J. Xu, S. Ji, X. Zhang, Q. Wang, B. Zhao, G. Pan, W. Cao, P. Chen, and R. Beyah, "Miner: A hybrid data-driven approach for rest api fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4517–4534.
- [19] Y. Liu, Y. Li, G. Deng, Y. Liu, R. Wan, R. Wu, D. Ji, S. Xu, and M. Bao, "Morest: Model-based restful api testing with execution feedback," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2022.
- [20] G. Deng, Z. Zhang, Y. Li, Y. Liu, T. Zhang, Y. Liu, G. Yu, and D. Wang, "{NAUTILUS}: Automated {RESTful}{API} vulnerability detection," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5593–5609.
- [21] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Automated Black-Box Testing of RESTful Web APIs," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2021, pp. 682–685.
- [22] H. Wu, L. Xu, X. Niu, and C. Nie, "Combinatorial testing of restful apis," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2022.
- [23] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful REST API fuzzing," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2019, p. 748–758.
- [24] E. Viglianisi, M. Dallago, and M. Ceccato, "Resttestgen: Automated black-box testing of restful apis," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2020.
- [25] Z. Hatfield-Dodds and D. Dygalo, "Deriving semantics-aware fuzzers from web api schemas," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2022, pp. 345–346.
- [26] W. Du, J. Li, Y. Wang, L. Chen, R. Zhao, J. Zhu, Z. Han, Y. Wang, and Z. Xue, "Vulnerability-oriented testing for restful apis," in *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024, pp. 739–755.
- [27] B. Marculescu, M. Zhang, and A. Arcuri, "On the faults found in rest apis by automated test generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–43, 2022.
- [28] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Checking security properties of cloud service rest apis," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2020, pp. 387–397.
- [29] D. Corradini, M. Pasqua, and M. Ceccato, "Automated black-box testing of mass assignment vulnerabilities in restful apis," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2553–2564.
- [30] E. M. Pasca, D. Delinschi, R. Erdei, and O. Matei, "Llm-driven, self-improving framework for security test automation: Leveraging karate dsl for augmented api resilience," *IEEE Access*, 2025.
- [31] A. Santos Filho, R. J. Rodríguez, and E. L. Feitosa, "Automated broken object-level authorization attack detection in rest apis through openapi to colored petri nets transformation," *International Journal of Information Security*, vol. 24, no. 2, p. 83, 2025.
- [32] A. Arcuri and J. P. Galeotti, "Handling SQL databases in automated system test generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–31, 2020.
- [33] E. Martin, T. Xie, and T. Yu, "Defining and measuring policy coverage in testing access control policies," in *Information and Communications Security: 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006. Proceedings 8*. Springer, 2006, pp. 139–158.
- [34] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 5, pp. 507–525, 2015.
- [35] 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*, 2019, pp. 15–21.
- [36] 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.
- [37] A. Arcuri, M. Zhang, A. Golmohammadi, A. Belhadi, J. P. Galeotti, B. Marculescu, and S. Seran, "EMB: A curated corpus of web/enterprise applications and library support for software testing research," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 433–442.