REST API Fuzzing by Coverage Level Guided Blackbox Testing

Chung-Hsuan Tsai*, Shi-Chun Tsai*, and Shih-Kun Huang*†
*Department of Computer Science, †Information Technology Service Center
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
{zx.c, sctsai, skhuang}@nycu.edu.tw

Abstract—With the growth of web applications, REST APIs have become the primary communication method between services. In order to ensure system reliability and security, software quality can be assured by effective testing methods. Black box fuzz testing is one of the effective methods to perform tests on a large scale. However, conventional black box fuzz testing generates random data without judging the quality of the input.

We implement a black box fuzz testing method for REST APIs. It resolves the issues of blind mutations without knowing the effectiveness by Test Coverage Level feedback. We also enhance the mutation strategies by reducing the testing complexity for REST APIs, generating more appropriate test cases to cover possible paths.

We evaluate our method by testing two large open-source projects and 89 bugs are reported and confirmed. In addition, we find 351 bugs from 64 remote API services in APIs.guru.

The work is in https://github.com/iasthc/hsuan-fuzz.

Index Terms—OpenAPI, REST API, Test coverage level, Black-box testing, Fuzz testing, Software quality

I. Introduction

Due to the limitation to exhaust possible input values of the program, there still exists potential errors in normal operations, and even leads to security concerns in the program. Software testing can effectively reduce risks, but manual testing and writing test cases are time-consuming, so we need to use automated tools to resolve this problem. Fuzz testing is one of the automated testing methods.

Fuzzing is very effective for triggering software errors. It can find exceptions that the developer has not dealt with, and identify whether the software has security issues. Fuzzing is also a common method of finding software vulnerabilities. For test targets with fixed binary formats (such as video and sound), fuzz testing is effective. There are many studies on black box, white box, or grey box test methods [1] [2]. However, network and Web applications are with variants of formats, and it is more difficult to use the same techniques as been used in fuzzing binary programs.

Internet applications and related services are popular. REST APIs have gradually become the primary communication method between services, ranging from cloud SaaS (software as a service) to IoT (Internet of Things) devices. If it can be tested on a large scale, the quality of services will be improved. The emergence of the OpenAPI specification [3] defines a standardized interface for REST APIs, making automated testing of a large number of REST APIs easier.

The current grey box fuzz testing in the REST API will track code coverage through program instrumentation, and guide the fuzzer to maximize code coverage, but it can only be applied to the programming language supported by the tool, such as Ruby [4] or Python [5]. The black box fuzz testing will try to analyze and infer the dependencies between requests, and test by replacing fixed parameter values, as much as possible to combine the most API paths in each round of requests [6]. Random tests may be carried out with the goal of increasing the coverage of status codes [7].

It can be seen from the above that grey box fuzzing can guide the fuzzing process through the feedback of coverage information, but it is limited by the specific programming languages supported. In contrast, although black box fuzzing has the advantage of not being restricted by the programming language, blind, random, and divergent mutations will make some paths hard to be triggered. In other words, if the black box fuzz testing process can identify the effects of the mutation, it can retain the advantages of not being restricted by the programming language and perform more efficient testing.

We, therefore, propose to add "Test Coverage Level" [8] with feedback to the REST API black box fuzz testing to know whether the mutation effect and input are appropriate and to improve the shortcomings of the ordinary black box test that has no feedback from the testing results. Meanwhile, we resolve the issues that may be encountered in testing REST APIs and adopt different strategies to increase the speed and chance of triggering errors. Finally, a proof-of-concept tool is implemented that only requires OpenAPI specifications and path dependencies to perform automated testing, which can speed up the discovery of errors.

Our contributions:

- Propose a new strategy for black box fuzzing with estimated code coverage.
- Implement a new black box fuzzer.
- Add "Test Coverage Level" as feedback for fuzzing.
- Resolve the issues of "request sequence", "path dependency", "valid parameter" and "access token".
- Use "pairwise testing" to reduce the combinations of test parameters and speed up the testing process.
- Increase the chance of triggering errors through different "mutation strategies".

II. Background

We will explain "OpenAPI Specification", as well as "Fuzz Testing", "Black Box Testing", "Pairwise Testing" and "Test Coverage Level" respectively.

A. OpenAPI specification

REST (Representational State Transfer) is a software architecture style for network applications. REST style is usually used in HTTP, mainly because the characteristics of HTTP are consistent with the definition of REST style, but it is not actually bound to any communication protocol. The Web API defined in this style is called RESTful API (hereinafter referred to as REST API).

TABLE I
REST API and Non-REST API

Behavior	Request Methods	REST	Non-REST
Create	POST	/Users	/newUser
Read	GET	/Users/1	/getUser
Update	PUT	/Users/1	/updateUser
Delete	DELETE	/Users/1	/deleteUser

OpenAPI specification [3] (hereinafter referred to as OpenAPI) is an interface description language, which is written in YAML or JSON format, not limited to a specific programming language so that REST API has a standardized description method. By directly reading the document or using the visualization tool [9], you can learn the path, parameters, functions, and other information of the REST API, which is convenient for developers to implement and test according to the specifications.

B. Fuzz testing

Fuzz testing (Fuzzing) is a technique for automated software testing, which generates random and unexpected inputs repeatedly in the hope of triggering errors in the target program. It can effectively find program anomalies, logic errors, or developer design flaws, thereby improving program reliability and software quality. According to the obtained program information, testing can be divided into kinds of black box, white box, and grey box manner.

Black box testing usually refers to only understanding the program's input, output, and specifications, and not knowing the internal behavior of the program. Compared with black box testing, white box testing is with source code, which can generate better test cases through more complex techniques (such as symbolic execution), but it takes more time and cost.

Grey box testing is to have information about the specifications and program runtime information, such as tracking the code coverage achieved by each input through program instrumentation, understanding the effect of each round of mutation, and maximizing the code coverage. Most grey box fuzz testing [10] [11] adopt this scheme, and use better

strategies such as seed selection and mutation methods to optimize fuzzing processes.

Code coverage is a software metric that indicates how much code is executed during the execution of a program. It provides an assessment and measure of the effectiveness of the tests. Compared to low code coverage, high code coverage has more chances for errors to occur.

C. Pairwise testing

Pairwise testing is combinatorial testing, dedicated to using fewer test cases to cover the paired combinations of multiple parameters. Since the program usually triggers the error [12] by the interaction of single or paired parameters, this method can effectively reduce the number of tests. The minimum number of tests will be the product of the two most characteristic parameters.

$$C = \max_{i=1}^{Q} B_i \times \max_{j=1, j \neq i}^{Q} B_j$$

For example, if there are three different parameters of A, B, and C, with two, three, and four attributes respectively, the most comprehensive test combination needs to be executed 24 times ($2 \times 3 \times 4$). If a paired test is used, Only need to execute 12 (3×4) times.

D. Test Coverage Level

Martin-Lopez [8] et al. proposed a model for comparing test technologies for REST API. By formulating ten test coverage criteria, and combining them into eight test coverage levels (hereinafter referred to as TCL) to overcome the inability to automatically measure the effectiveness of test cases. To achieve a specific TCL, all previous criteria must be met, which can be evaluated from different perspectives (such as API, path, or request method), and prove that TCL is correlated with code coverage and failure detection rate positively.

Table II is "Test Coverage Model" and the test coverage criteria that each TCL needs to include. We use TCL to be estimated code coverage and as feedback to the fuzzing process.

TABLE II
Test Coverage Model

TCL	Input Criteria	Output Criteria
0		
1	Paths	
2	Operations	
3	Content-type	Content-type
4	Parameters	Status code classes
5	Parameters	Status codes
6	Parameters	Response body properties
7	Operation flows	

III. Design and Implementation

This research proposes a new concept "REST API black box fuzz testing based on coverage level guidelines", which uses TCL as feedback and guides the black box fuzzers to improve the drawback of black box testing without knowing the mutation effects. We summarize the problems encountered when testing REST APIs, and design our method to resolve the issues, and then explain our fuzzer "HsuanFuzz".

A. Complexity of Testing

Functions in the application programs need to be called in the correct order, and REST API is no exception. Only by sending requests in the correct order can meaningful test cases be generated, thereby increasing code coverage. Test cases that are "deleted" and then "updated" are of little significance.

The highly structured REST API is also a problem. Paths are interdependent. The response of path "A" may be a parameter in the request of path "B". If the path dependency problem is not resolved, paths and methods cannot be tested completely, and "404 Not Found" responses will be received.

The format of parameters is also one of the issues to be resolved. Many parameter strings have specific formats, such as email, date or uuid, etc. Randomly generated values can easily lead to "400 Bad Request".

The last part is about authorization. API services obtain authorization methods in different ways, such as API key, Bearer token, or OAuth. If we do not obtain authorization correctly, we will get a response of "401 Unauthorized", and we will not be able to properly test the path that requires authentication.

Based on the above observations, four issues must be resolved: request sequence, path dependency, valid parameter, and access token.

B. Architecture

The purpose is to add a guideline method to the black box fuzzing to improve the disadvantage of not being able to determine the effect of the mutation. We input OpenAPI, path dependencies, and access information, and use TCL as feedback to automate the testing of REST API. We also try to resolve the issues mentioned in Section III-A.

"Grammar" phase in Fig. 1 analyzes the dependencies between OpenAPI and paths to ensure the order between paths and requests. To avoid incorrect parameter format, we use OpenAPI example values or pre-defined default values as initial values and store them in the corpus in the form of ProtoBuf.

After reading the seed, it needs to go through the "parser" phase to restore it to a grammar, replace the ID parameters to resolve the path dependence problem. The "mutation" phase mutates each request parameter in pairs. If authentication is required, we first obtain an access token and then send it along with the request in the "sender" phase. In addition, the response to a successful request is recorded for subsequent use.

In the "analysis" phase, the response of each path will be checked. If the TCL of any path increases, it will be added to the corpus. If the status code of any response is 500, an error will be recorded. The seeds are then read from the corpus and the steps above are repeated.
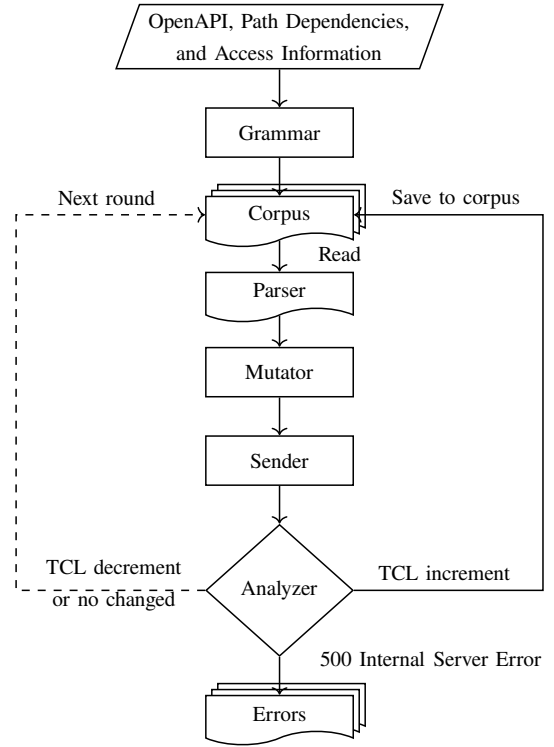

Fig. 1. The API Fuzzing Architecture

C. Implementation

We have implemented a proof-of-concept tool and named it "HsuanFuzz", which can perform black box fuzz testing based on coverage level guidelines on REST APIs. It can be executed by importing YAML files such as OpenAPI specifications, path dependencies, and access information (if required). We explain the details in the following.

1) Setup of Initialization: In order to make the fuzzer work correctly and efficiently, we need to enter some relevant information manually.

a) Entering related information: "Path dependencies" requires the tester to list the parameter dependencies of all paths, fill in the ID parameter and source of the request, and predefine the TCL operation flows, as shown in Listing 1.

In addition, the tester also needs to fill in the "access information" to obtain the token for REST API, or directly write the authorization key into the YAML file, as shown in Listing 2.

b) Defining Corpus and Error: We use a map to create storage areas to avoid repeated storage of the same grammar or errors. The key is a hash of the path and request parameters, and the value is the serialized grammar. This allows us to restore the seed to grammar by deserialization.

```
paths:
  /articles/{slug}:
    items:
    - key: "slug"
      source:
        path: "/articles"
        key: "article[{slug}]"
posts:
  /articles:
    flow:
    - method: GET
      path: "/articles/{slug}"
    - method: PUT
      path: "/articles/{slug}"
```

Listing 1. Path Dependencies

```
url: 'https://localhost/token'
method: POST
content_type: application/json
body: '{"username": "user", "password": "12345"}'
name: access_token
in: header
key: 'bearer'
token: ''
hardcode: false
```

Listing 2. Token Authorization

2) Producing Seed Input: This phase has two parts. The first part is to generate the grammar, analyze the OpenAPI and solve the "request sequence" and "path dependency" problems. The second part is to assign parameter values, using OpenAPI examples or pre-defined default values.

a) Generating Grammar: The first execution will generate a grammar based on the defined data structure, and serialize it as a seed for the subsequent generation of more appropriate test cases. In order to analyze and calculate the TCL, we minimize the test cases, and each request will send the "number of response status code types" times.

```
syntax = "proto3";

package base;
import "google/protobuf/struct.proto";

message Info {
    repeated Node nodes = 1;
}

message Node {
    uint32 group = 1;
    string path = 2;
    string method = 3;
    repeated Request requests = 4;
}

message Request {
    string type = 1;
    google.protobuf.Struct value = 2;
}
```

Listing 3. Seed in ProtoBuf format

b) Solving dependencies: The path itself has dependencies. It cannot be correctly deleted without creating an object. Paths are also dependent on each other. The value in the request parameter of path "B" needs to be obtained through the response of path "A". We have tried automated analysis, but found that the efficiency is low, because developers have different methods for defining parameters, and the user's ID parameter may be defined as a different name or even a typo, such as user_id, userId, or usersId. Manual input combined with depth-first search can effectively improve the accuracy of dependencies. For example, if one wishes to test the path "/articles/{slug}/comments", the order of the requests will be the same as in Table III, which can resolve the "request order" issue and overcome the "path dependency" problem.

TABLE III
REQUEST ORDER OF DEPENDENT PATH

*	Method	Path	Description
1	POST	/articles	New Article
2	PUT	/articles/{slug}	Update an Article
3	GET	/articles/{slug}	Read an Article
4	POST	/articles/{slug}/comments	New Comment
5	GET	/articles/{slug}/comments	Read All Comments
6	DELETE	/articles/{slug}/comments/{id}	Delete a Comment
7	DELETE	/articles/{slug}	Delete an Article

3) Giving Values to Parameters: OpenAPI strings have many different formats. Randomly generating strings that do not conform to the format will result in "400 Bad Request". A good OpenAPI usually adds "examples" to the parameter attributes, so we can generate a valid initial value based on this information. If there is no example in OpenAPI, the initial value is generated according to the defined rules, such as Table IV.

TABLE IV
DEFAULT VALUES OF PARAMETER FORMAT

Format	Default Values
date	2021–05–28
date-time	2021–05–28T10:00:00+08:00
time	10:00:00+08:00
email	user@example.com
hostname	localhost
ipv4	127.0.0.1
ipv6	0:0:0:0:0:0:0:1
uri	https://tools.ietf.org/html/rfc3986
uuid	5bcafcb2–a669–11eb-bcbc–0242ac130002

D. Input Mutation

The mutation phase will give priority to the path with the ID parameter, and then mutate according to the strategy. In order to avoid that the ID field can never be mutated, we put the mutation strategy behind it, that is, the ID parameters also have a certain chance to mutate to increase the chance of triggering errors.

1) Handling ID Parameter: Correctly handling the ID parameters can greatly increase the number of successfully explored paths. We take the value from the response body through a defined method. Take Listing 1 as an example. "key" represents the ID parameter, and the "path" and "key" in the "source" are the data source of the ID parameter. For example, "data[relationships{id}]" represents the "id" field of the "relationships" object in the "data" array.

2) Processing Mutation Strategy: We use some methods of Go-fuzz [13] to modify the string, such as "insert, duplicate, remove a range of bytes", "set, add, subtract a byte", "exchange two bytes", "replace a multi-byte ASCII number" and "bit flip". In addition, to modify the string directly, we added our own experience of writing REST API to the mutation strategy. If the exception is not handled properly, giving the wrong type or removing the necessary fields can easily cause the service to trigger an error. Therefore, according to the characteristics of the REST API, "replace data type", "remove fields" and "modify string" are randomly adopted.

However, mutating all parameters at the same time will easily lead to "400 Bad Request". Combining different numbers of parameters will cause the path to explode. Therefore, we adopt the method of pairwise testing. In each round of fuzzing, only two parameters of each request are changed to increase the chance and speed of triggering errors.

E. Transferring Request

After completing the pre-work, information on paths, methods, and parameters can be obtained from the mutated grammar. We set them to the correct fields, decode the string, remove control characters, and send the request after obtaining the access token. Lastly, we also record the response body of a successful request for using in the following requests.

F. Analyzing Response

In order to evaluate the effectiveness of the test case, we use the TCL mentioned in Section II-D as a feedback method and use the input and output criteria to calculate the TCL of each path. If the TCL of any path increases, it is defined as "interesting" and stored in the corpus for use in the next round of fuzzing. In addition, if the first digit of the response status code is five, it means that an error has been triggered, and will be recorded for later reproduction.

The maximum TCL of our implementation is 6 in general, unless there is a pre-defined operation flows "Read Single", "Read All", "Update", "Delete" in the input path dependencies (Listing1) and meet the behavior of Table III to reach TCL 7.

IV. RESULTS AND EVALUATIONS

We have implemented the fuzzer called HsuanFuzz. In order to be able to evaluate HsuanFuzz, based on the two different aspects of "code coverage" and "error finding ability", we discuss whether the research objectives are achieved in the form of research questions, and evaluate and present the results of this research.

- RQ 1: Is it effective to add TCL (test coverage level) guidelines to the REST API black box fuzz testing?
- RQ 2: Is it better than existing REST API black box fuzzers?
- RQ 3: What other advantages does black box fuzzing have?

Experimental Environment

Our devices are based on Ubuntu version 20.04 with Intel Core i7–10700 processor and 16 GB memory. We use "HsuanFuzz based on black box coverage level guidelines", "HsuanFuzz without black box coverage level guidelines" and black box fuzzer "RESTler version 7.3.0" [14] as comparisons. It is evaluated whether it achieves better results in different aspects according to the given time budget.

RQ 1: Guiding by TCL (Test Coverage Level)

In order to verify RQ 1, we selected the "RealWorld" program of Go language implementation [15] as the test target. We execute each fuzzer for 200 seconds and measure the code coverage with the number of lines of code executed as a unit.

TABLE V
REALWORLD PROGRAMS WITH LINES AND COVERAGE

Components	Lines of Code	H[a]	H[b]	R[c]
User	319	172	163	116
Article	544	438	413	36
Total	863	610	576	152

RealWorld [16] is a demonstration application that provides a complete API specification and can use different front-end and back-end implementations. There are two components: "User" and "Article", which can perform functions such as register, login, post articles, and comment. There are 11 paths and a total of 19 request methods.

We use OpenAPI provided by RealWorld, and test in the default method of each fuzzer. Special attention is paid to the fact that OpenAPI does not provide parameter examples. HsuanFuzz can execute about 26 rounds during 200 seconds, which means that there are 26 chances to generate new seeds, while RESTler cannot define the number of requests for one round.

[a] HsuanFuzz based on black box coverage level guidelines
[b] HsuanFuzz without black box coverage level guidelines
[c] RESTler

Fig. 2 shows that the code coverage of "HsuanFuzz based on black box coverage level guidance" has only been executed for about 26 rounds, and the code coverage rate increases significantly faster than "HsuanFuzz without black box coverage level guidance". As to RESTler, after we adjusted its "fuzzing_mode" setting, whether it was bfs, bfs-cheap or random-walk, it was difficult to increase its coverage quickly.
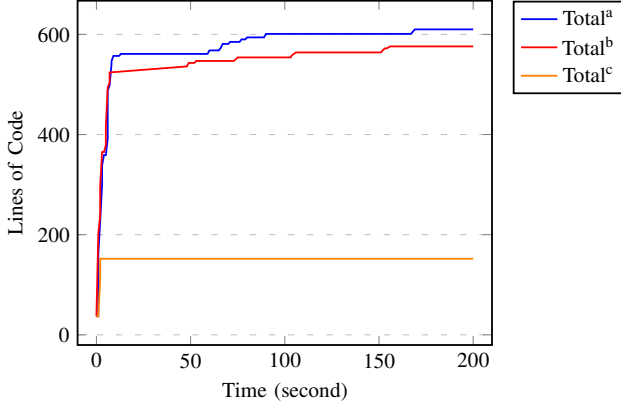

Fig. 2. RealWorld Code Coverage Trend

Compared with "HsuanFuzz without black box coverage level guidelines", "User" in Fig. 3 can still increase code coverage after a period of execution, while "Article" can increase code coverage at a faster speed. It proves that black box coverage level can guide the fuzzer effectively, converge the direction of mutation, and mutate values of parameters more quickly and properly to increase code coverage.
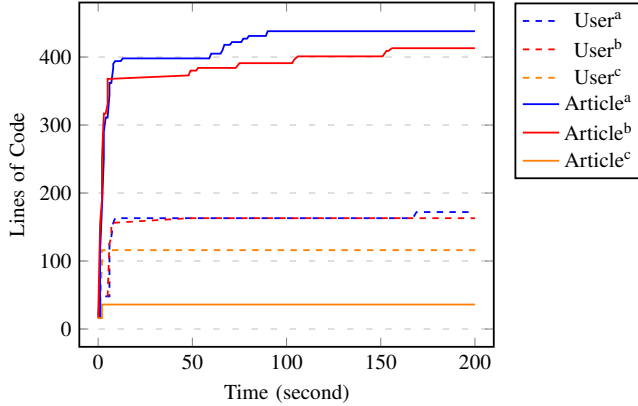

Fig. 3. RealWorld Individual Code Coverage

RQ 2: Service Comparisons

We chose two open-source projects of e-commerce platform, Spree Commerce [17] and Magento Community [18], as our testee and defined status code 500 as an error to evaluate the ability to find errors. The former is written in Ruby, the latter in PHP, and both are great projects on GitHub with over 9,700 stars and a combined total of over 2,300 contributors. We built Spree on version 4.2.1 and Magento on version 2.4.2 respectively in the experimental environment and entered two YAML files (path dependencies and access information) for a 24-hour test.

1) Spree Commerce: Spree has 31 paths and a total of 35 request methods. Table VI lists these three fuzzers with errors found in 24 hours.

TABLE VI
ERRORS OF SPREE COMMERCE IN 24 HOURS

Method	Path	H[a]	H[b]	R[c]
GET	/order_status/{number}	✓		
GET	/products/{id}	✓		
POST	/account	✓	✓	✓
POST	/account/addresses	✓	✓	✓
POST	/cart/add_item	✓	✓	✓
PATCH	/account	✓	✓	✓
PATCH	/account/addresses/{id}	✓	✓	
PATCH	/cart/set_quantity	✓	✓	✓
PATCH	/checkout	✓	✓	✓
DELETE	[d]		✓	✓
DELETE	[e]		✓	✓

[d] /cart/remove_coupon_code/{coupon_code}
[e] /cart/remove_line_item/{line_item_id}

HsuanFuzz does not include repeated errors in its design and finds a total of 11 errors with different paths and parameters. RESTler found 55 errors. After removing duplicate errors based on the path, method, and parameters, there were a total of 6 errors.

Table VI shows that with a small number of paths and request methods and a long execution time, both HsuanFuzz can find a similar number of errors, but not the same number of errors, presumably due to the inability to converge the direction of mutation. RESTler could not find the error of "/account/addresses/{id}" path. We concluded that HsuanFuzz was able to find this error because it had issued the path dependency problem and adopted a good mutation strategy.

We also executed HsuanFuzz to find some additional errors. A total of 14 errors were reported for two versions of Spree [19] [20], which were confirmed to be bugs by the project author. In addition, an error in the YAML file of OpenAPI specification was reported [21].

2) Magento Community: Magento is a very large-scale service. There are 312 paths and 395 request methods in Magento Admin REST endpoints. A 24-hour test was also performed to list the errors found in these three fuzzers.

Table VII reveals that HsuanFuzz can find more errors than RESTler at the same time. However, RESTler cannot find an error with the request method being GET. We judged that it is because the id parameter of paths will not be mutated. The number of errors found by HsuanFuzz with or without black box coverage level guidance is similar because Magento errors

are mostly caused by data type changes or lack of fields, which can be triggered after fuzzing has been conducted for a certain period of time. It will be explained and compared in more detail later.

We also reported the errors we found to the project author. A total of 75 errors were reported in two versions [22] [23], all of which were confirmed as bugs.

3) Analysis: After 24 hours of testing, HsuanFuzz and RESTler were able to find 11 and 6 errors out of 14 known errors for Spree, and 71 and 3 errors out of 75 known errors for Magento, proving that HsuanFuzz is superior to RESTler in terms of error finding ability.

If the comparison is made from the perspective of whether "black box coverage level guidelines", both HsuanFuzz can find a similar number of errors in a long enough time. Therefore, we use Fig. 4 and Fig. 5 to analyze the cutting time period.

In the early stage of fuzzing, HsuanFuzz will continue to add seeds to the corpus, because there are constantly ways to improve TCL, but after a period of time, it will stop adding seeds because it cannot improve TCL. However, we found that HsuanFuzz can still find new interesting seeds 2.5 hours after starting to test Spree.

Also in Fig. 4, we found that it can continue to find new bugs after a long period of operation. It means that our black box coverage level guidelines are effective.
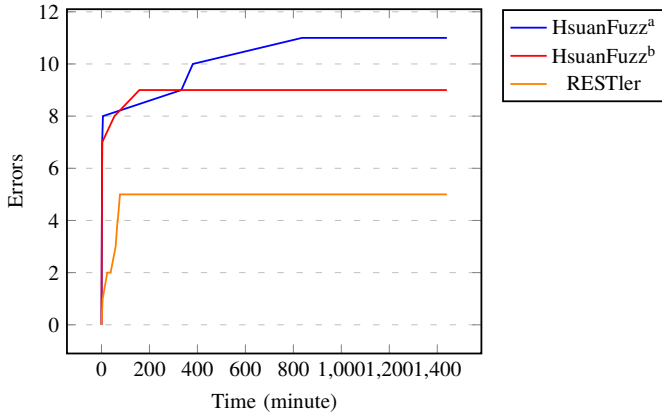

Fig. 4. Spree Commerce Errors Trend

Both types of HsuanFuzz ended up with similar error results in Magento. We found that we were unable to find new errors after about two hours, and most of the error types were data type changes or missing fields, which is exactly the strategy we have defined for the REST API based on our experience. In order to give RESTler more time to find the error, we also test the same as Spree for 24 hours.

Fig. 5 shows the number of Magento errors found in the first 15 minutes. About 3 minutes earlier, both could find a similar number of errors, and then "HsuanFuzz based on black box coverage level guidance" clearly outperformed "HsuanFuzz without black box coverage level guidance". This shows that the feedback and guidance we added and the idea of reusing

the seeds of interest are effective and that the black box fuzz testing with guidance can find errors more quickly.
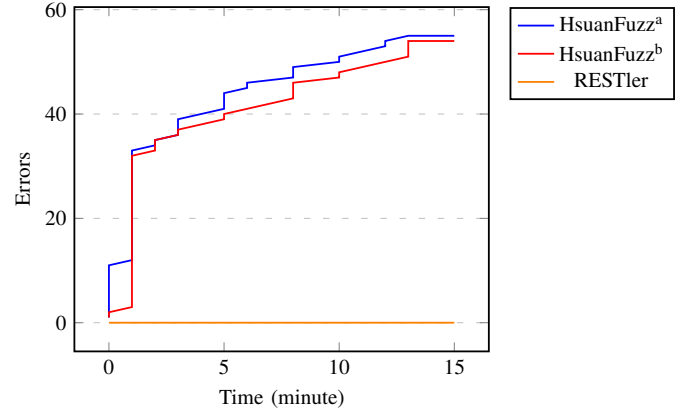

Fig. 5. Magento Community Errors Trend

According to Fig. 6, HsuanFuzz has a good error finding ability, while Fig. 4 and Fig. 5 show that the coverage level guide is effective in finding new interesting seeds after a certain period of time. Finally, Table VIII shows the type and number of errors we reported.
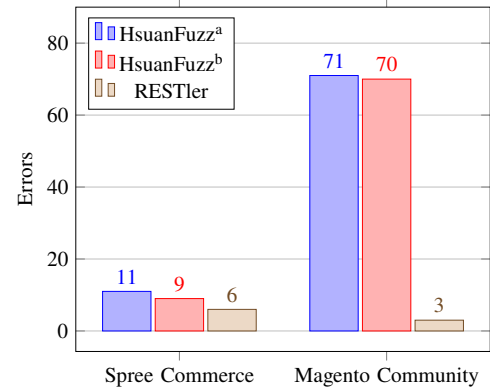

Fig. 6. Errors in 24 hours

RQ 3: Other Advantages

The grey box fuzzer requires program instrumentation and has the drawback of programming language restrictions. For black box testing, it does not need to understand the internal behavior and the programming language used. Therefore, black box fuzzing can be performed on a large scale, quickly and effectively without having to set up a Web API server and without having background knowledge of the target service's programming language.

APIs.guru [24] is committed to becoming a Web API Wikipedia, and only public, persistent and useful Web APIs will be included. It provides many different types of services, most of which can be accessed directly, and some can be browsed only with subscription or authorization. With the specifications of Swagger and OpenAPI versions, it can provide a perfect entry point for black box testing.

TABLE VII
ERRORS OF MAGENTO COMMUNITY IN 24 HOURS

Method	Path	Hᵃ	Hᵇ	Rᶜ
GET	/...ᵈ	✓	✓	
POST	/ˢ	✓	✓	✓
POST	ᵉ/ʷ			✓
POST	ᶠ/ᵗ	✓	✓	
POST	ᶠ/ʲ	✓	✓	
POST	ᶠ/items	✓	✓	
POST	ᶠ/ᵏ	✓	✓	
POST	ᶠ/ˡ	✓	✓	
POST	ᶠ/ᵐ	✓	✓	
POST	ᶠ/ⁿ	✓	✓	
POST	ᵍ/ʲ	✓	✓	
POST	ʰ/items	✓	✓	
POST	/categories	✓	✓	
POST	/coupons	✓	✓	

Method	Path	Hᵃ	Hᵇ	Rᶜ
POST	/guest-ˢ	✓	✓	
POST	ⁱ/ᵗ	✓	✓	
POST	ⁱ/ʲ	✓	✓	
POST	ⁱ/items	✓	✓	
POST	ⁱ/ᵏ	✓	✓	
POST	ⁱ/ˡ	✓	✓	
POST	ⁱ/ᵐ	✓	✓	
POST	ⁱ/ⁿ	✓	✓	
POST	/inventory/ᵒ	✓	✓	
POST	/invoices/	✓	✓	
POST	/orders	✓	✓	
POST	/products	✓	✓	
POST	/salesRules	✓	✓	
POST	/taxRates	✓		

Method	Path	Hᵃ	Hᵇ	Rᶜ
POST	/ᵖ-validation/ᵖ	✓	✓	
PUT	ᵘ/q	✓	✓	✓
PUT	ᵛ/q	✓	✓	✓
PUT	ᵉ/options/{optionId}			✓
PUT	ᵉ/ʷ			✓
PUT	/carts/mine			✓
PUT	ᶠ/items/{itemId}	✓	✓	
PUT	ᵍ/items/{itemId}	✓	✓	
PUT	/categories/{id}	✓	✓	
PUT	ʳ/variation	✓	✓	
PUT	ⁱ/items/{itemId}	✓	✓	
PUT	/orders/create	✓	✓	
PUT	/products/{sku}	✓	✓	

ᵈ A total of 35 paths.
ᵉ /bundle-products
ᶠ /carts/mine
ᵍ /carts/{cartId}
ʰ /carts/{quoteId}

ⁱ /guest-carts/{cartId}
ʲ estimate-shipping-methods
ᵏ payment-information
ˡ set-payment-information
ᵐ shipping-information
ⁿ totals-information

ᵒ source-selection-algorithm-result
ᵖ vertex-address
q {amazonOrderReferenceId}
ʳ /configurable-products
ˢ address/cleanse
ᵗ billing-address
ᵘ /amazon-billing-address
ᵛ /amazon-shipping-address
ʷ {sku}/links/{id}

We selected 936 YAML files with OpenAPI, a total of 768 services, and tested them without entering path dependencies and access information. Finally, 465 responses with a status code greater than "500" were found in 64 services, including 351 with the status code of "500 Internal Server Error" in 47 services.

We succeeded in finding the error of the remote target service without knowing the API programming language. By collecting relevant information in the response, it can be found that the target service uses programming languages such as Ruby, PHP, C#, and Java, which also proves that black box testing has the advantage of not being restricted by languages.

Threats To Validity

Greybox fuzz testing has been proved to be effective in discovering security bugs. However, programs either in source or binary form need to be instrumented for coverage feedback. We propose to take advantage of TCL (test coverage level) in blackbox testing to be as a replacement of conventional code coverage. It is the primary threat to our work. Since TCL is only an approximation of test coverage, it relies on the response and request through the OpenAPI specifications. Currently, there are only seven levels of TCL and it is too coarse to be as precise as the code coverage.

V. RELATED WORK

RESTler [6] is currently a well-known REST API black box fuzzer, proposed by Atlidakis et al. and as Microsoft's open-source project [14]. RESTler is a generation-based fuzzer that uses a predefined dictionary to replace parameter values in grammar and analyzes OpenAPI to infer dependencies between paths statically. It is to lengthen the test case as much as possible with most paths for a single round of requests. In contrast, our research uses manual input to achieve a more accurate path dependency. We also add TCL feedback to guide the fuzzer and increase the chance of triggering errors through mutation-based fuzzing and other strategies.

Pythia [4] is based on grey box fuzzing, also proposed by Atlidakis et al., using the built-in functions of Ruby to get code coverage. The initial seed is the execution result of RESTler, and the mutation stage is improved by the machine learning method, and the validity of the grammar is still maintained after some noise is injected. Coverage-guided fuzzing in Python-based web server [5] is modified from Python-AFL, which is also one of the grey box fuzzing methods, which can obtain code coverage from Flask. The grey box fuzzing methods mentioned above have limitations, and the test targets are limited by the programming language of the fuzzer.

EvoMaster has two methods: white box fuzzing [25] and black box fuzzing [7]. The former needs to read the document

Target	Type	Number
Spree	No Method Error	7
Spree	SQL Exception	2
Spree	Invalid URI	2
Spree	Unknown Attribute	1
Spree	Page Incalculable	1
Spree	Type Error	1
Spree	Column not found	27
Magento	Property A does not have accessor method B	26
Magento	Missing required argument A	7
Magento	Failed to parse time string	3
Magento	Amazon Pay could not process your request	2
Magento	Request name A doesn't exist	2
Magento	A must be of type string	2
Magento	Undefined offset N	2
Magento	Invalid argument supplied for foreach ()	1
Magento	Expects parameter N to be array	1
Magento	Class A does not exist	1
Magento	Invalid scope type A	1

and write part of Java code for the test target in order to generate a test case. It requires programming ability. The latter is tested randomly, to maximize types of HTTP status codes, and does not have an appropriate feedback mechanism.

QuickREST [26] is a property-based black box testing method that can quickly verify and test OpenAPI but does not send a request to the test target, so it may be different from the developer's implementation result. RestTestGen [27] is also a black box test method, after executing an effective test, try to trigger an error by violating restrictions and other methods. In addition, customized rules such as renaming fields or stemming are used to analyze the interdependence of paths. In our research, the target is continuously tested, and the strategy of adding variant strings increases the chance of triggering errors, and the accuracy of path dependencies is improved through manual input.

VI. CONCLUSIONS

This research has developed a proof-of-concept tool, a REST API black box fuzzer based on coverage level guidelines. The purpose is to produce more appropriate test cases for black box fuzzing. In the case of avoiding incorrect requests, meaningfully mutations are conducted to generate new values and test every path as much as possible.

We add test coverage levels feedback to guide the fuzzer, and resolves the drawback of black box testing that cannot know the effect of the mutation. We also resolve the issues of testing complexity for REST APIs and strengthen mutation strategy to increase the chance of triggering errors.

Our research is inspired by AFL which provides feedback based on code coverage. We uses input and output of specifications to approximate code coverage by test coverage levels. We enter the appropriate OpenAPI, path dependencies, and authorization information to automatically generate test cases and perform black box fuzzing on the test target API. To our best knowledge, this is the first research to apply test coverage levels for REST API black box fuzzing.

It is important to note that this research is not a grey box test, and does not require instrumentation of program or server. One of the benefits of using black box testing is that it is not restricted by any programming language and does not need to understand the internal behavior of the program. Through the OpenAPI that clearly defines the input and output, we can test the remote target API more easily. Black box fuzzing with feedback guidance currently seems to be the best way to test a large number of REST APIs.

This study proves that the coverage level guidelines are effective in the estimated form of code coverage, and also proves that both self-built and remote test targets can effectively find errors. Finally, in two large-scale open-source projects, a total of 89 errors were found and reported, and 351 errors were also found in 64 remote services provided by APIs.guru.

Future Work

1) Increasing Coverage Level: There are a total of 8 test coverage levels used in this research, ranging from level 0 to 7, which can be measured from multiple aspects of the API, and the relative relationship with code coverage can be obtained. If more levels are added, whether the test coverage can be more accurate and easier to find errors, is worth further research and discussion.

2) Manual Analysis and Automatic Identification: When recording errors of the remote test target, we found that some servers would return version information, such as "Tomcat 8.5.60" and "Tomcat 9.0.39". However, these versions have CVEs (CVE-2021–25122, CVE-2021–25122, and CVE-2021–25329).

We can use this research as the first stage to conduct large-scale testing. After finding out the possible weaknesses or vulnerabilities in the system, the second stage of manual analysis or automatic identification can be carried out to reduce the cost of manually verifying the test targets one by one.

3) Supporting IoT and CoAP: CoAP has a REST style and has the advantages of low power consumption. It is designed to be used on devices with relatively insufficient resources to meet the needs of the Internet of Things and the requirements for the use of networked devices.

Since it is similar to HTTP, it is worth further study by modifying the test coverage standard and using it on CoAP. We expect to extend this research to CGI, IoT communication protocol to identify weaknesses and try to find out the vulnerabilities of firmware components.

TABLE IX
TEST RESULTS OF REMOTE APIS

Test Target	500s	Test Target	500s	Test Target	500s
rest.zuora.com	3	apps.nrs.gov.bc.ca/gwells/api/v1	1	randomlovecraft.com/api	2
api.logoraisr.com/rest-v1	3	oralquestionsandmotions-api.parliament.uk	1	connect.squareup.com	5
apirest.isendpro.com/cgi-bin	1	api.interzoid.com	1	api.stoplight.io/v1	3
rest.ably.io	3	go.netlicensing.io/core/v2/rest	6	api.parliament.uk/search	2
ibl.api.bbci.co.uk/ibl/v1	1	marketcheck-prod.apigee.net/v2	51	api.up.com.au/api/v1	4
apps.gov.bc.ca/pub/bcgnws	1	nsidc.org/api/dataset/2	3	api.icons8.com	1
api.beezup.com	2	v2.namsor.com/NamSorAPIv2	2	api.sandbox.velopayments.com	4
bikewise.org/api	3	ntp1node.nebl.io	17	vocadb.net	18
www.bungie.net/Platform	110	www.neowsapp.com	7	worldtimeapi.org/api	2
rest-api.d7networks.com/secure	2	api.oceandrivers.com	5	ws.api.video	1
dev.to/api	5	osdb.openlinksw.com/osdb	1	api.ideaconsult.net/nanoreg1	3
api.presalytics.io/doc-converter	1	demo.orthanc-server.com	10	geodesystems.com	2
api.open511.gov.bc.ca	1	api.paylocity.com/api	23	api.cloudmersive.com	12
exude-api.herokuapp.com	1	peertube2.cpy.re/api/v1	1	smart-me.com	1
api.figshare.com/v2	10	phantauth.net	5	api.spoonacular.com	1
apps.gov.bc.ca/pub/geomark	4	api.pocketsmith.com/v2	5		

REFERENCES

[1] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT)*, Aug. 2020.

[2] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, " redqueen: Fuzzing with input-to-state correspondence," in *26th Annual Network and Distributed System Security Symposium (NDSS)*, vol. 19, Jan. 2019, pp. 1–15.

[3] The Linux Foundation. Openapi specification. [Online]. Available: https://spec.openapis.org/oas/v3.0.3

[4] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, "Pythia: grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations," *CoRR*, vol. abs/2005.11498, 2020.

[5] S. T. Liu, "Coverage guided fuzzing in python-based web server," M.S. thesis, National Chiao Tung University, Hsinchu, Taiwan, 2019.

[6] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Aug. 2019, pp. 748–758.

[7] A. Arcuri, "Automated black- and white-box testing of restful apis with evomaster," *IEEE Software*, vol. 38, no. 3, pp. 72–78, 2021.

[8] 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*, Aug. 2019, p. 15–21.

[9] Redocly LLC. Redoc. [Online]. Available: https://redocly.github.io/redoc

[10] Google LLC. American fuzzy lop. [Online]. Available: https://github.com/google/AFL

[11] LLVM. libfuzzer. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[12] R. Black, *Pragmatic software testing: Becoming an effective and efficient test professional.* John Wiley & Sons, 2016.

[13] D. Vyukov. Randomized testing for go. [Online]. Available: https://github.com/dvyukov/go-fuzz

[14] Microsoft Corporation. Restler. [Online]. Available: https://github.com/microsoft/restler-fuzzer

[15] Thinkster. Realworld with golang & gin. [Online]. Available: https://github.com/gothinkster/golang-gin-realworld-example-app

[16] ——. Realworld. [Online]. Available: https://github.com/gothinkster/realworld

[17] Spree. Spree commerce. [Online]. Available: https://github.com/spree/spree

[18] Magento. Magento community. [Online]. Available: https://github.com/magento/magento2

[19] Chung-Hsuan Tsai. Issue #10647, spree. [Online]. Available: https://github.com/spree/spree/issues/10647

[20] ——. Issue #10971, spree. [Online]. Available: https://github.com/spree/spree/issues/10971

[21] ——. Pull request #10626, spree. [Online]. Available: https://github.com/spree/spree/pull/10626

[22] ——. Issue #31551, magento. [Online]. Available: https://github.com/magento/magento2/issues/31551

[23] ——. Issue #32784, magento. [Online]. Available: https://github.com/magento/magento2/issues/32784

[24] APIs.guru. Openapi directory. [Online]. Available: https://github.com/APIs-guru/openapi-directory

[25] A. Arcuri, "Restful api automated test case generation," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Aug. 2017, pp. 9–20.

[26] 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)*, Oct. 2020, pp. 131–141.

[27] E. Viglianisi, M. Dallago, and M. Ceccato, "Resttestgen: Automated black-box testing of restful apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, Oct. 2020, pp. pp. 142–152.