Scientific Journal of Informatics
Vol. 11, No. 2, May 2024
p-ISSN 2407-7658 https://journal.unnes.ac.id/journals/sji/index e-ISSN 2460-0040

OffensiveRezzer: A Novel Black-Box Fuzzing Tool for Web API

Danar Gumilang Putera[1], Ruki Harwahyu[2*]
[1,2]Department of Electrical Engineering, Faculty of Engineering, Universitas Indonesia, Indonesia

Abstract.
Purpose: The purpose of this study is to introduce OffensiveRezzer, a novel tool designed for black-box fuzzing on Web APIs, and to evaluate its effectiveness in detecting errors, particularly focusing on errors related to input validation implementation.
Methods: We introduced OffensiveRezzer and conducted a comparative analysis against existing fuzzing tools such as EvoMaster, Schemathesis, RestTestGen, Restler, and Tcases to assess its performance. Fuzzing experiments were carried out on a custom Web API application with different input validation levels, namely no input validation, partial input validation, and full input validation.
Result: OffensiveRezzer demonstrated superior performance compared to other fuzzing tools in identifying errors in Web APIs. It outperformed competitors by detecting the highest number of unique errors. The total number of errors found by OffensiveRezzer in the application without validation, the application with partial validation, and the application with full validation was 416, followed by Restler (240), RestTestGen (145), EvoMaster (138), Tcases (78), and Schemathesis (42).
Novelty: The study has presented OffensiveRezzer as a novel tool specifically designed for black-box fuzzing on Web APIs, with a primary focus on testing input validation implementation. This tool fills a gap in existing fuzzing tools and offers improved capabilities for detecting errors in Web APIs.

Keywords: Black-box testing, Fuzzing, REST, Web API
Received May 2024 / **Revised** May 2024 / **Accepted** May 2024

INTRODUCTION

Web API technology has found widespread implementation across diverse applications, enabling different application services to engage and exchange information over network platforms [1]. It enables applications to expose their functionalities for use by others, making it a prominent choice for application integration. For example, third-party applications like payment gateways or real-time notification services can efficiently send and receive data from corporation applications via Web API. Web API interaction requires provider and consumer applications to adhere to a specific communication architecture. Various communication architectures, such as REST, GraphQL, and RPC, can be implemented within Web API, with REST emerging as the most used architecture [2].

Like web applications, Web API is vulnerable to various cybersecurity threats. Furthermore, the rising prevalence of Web API across diverse application infrastructures has captured the interest of malicious actors, leading them to target vulnerabilities within applications that make use of Web API [3]. Notable security weaknesses associated with Web API include CVE-2021-21972 and CVE-2023-34048 in vCenter software, as well as data breaches at Optus and T-Mobile. Regarding vCenter software, critical endpoints lack appropriate authentication, and insecure input validation methods lead to remote code execution (RCE). Similarly, the security breaches at Optus and T-Mobile resulted from a lack of authentication in one of their API endpoints. These examples show how configuration and security implementation often evades the attention of application developers [4], [5].

Preventing vulnerabilities arising from misconfigurations and flawed security implementations can be achieved before application deployment through security testing. Good security testing must be able to cover all possible errors that could occur in an application [6]. Fuzzing, also referred to as fuzz testing, stands out as a pivotal method for functionality and security testing. It involves testing insecure input validations to uncover security vulnerabilities. During the fuzzing process, the tested application received

*Corresponding author.
Email addresses: danar.gumilang@ui.ac.id (Putera), ruki.hwyu@gmail.com (Harwahyu)*
DOI: 10.15294/sji.v11i2.4631

various data inputs, including random data, data not matching specifications, and data containing malicious content, to assess the application's defense against potential damage caused by malicious inputs [7]. Fuzzing uses various methodologies, including black-box and white-box approaches [8]. Fuzzing with a black-box approach usually generates test scenario coverage that is not too broad compared to fuzzing with a white-box approach. In the black-box approach, the generated fuzzing test scenarios are only based on general application specifications, such as data input specifications. This is different from the white-box approach, where apart from analyzing the general specifications of the application, white-box fuzzing also analyses the application source code so that it can understand the internal processes of the application and generate better test scenario coverage.

In the context of fuzzing for Web API, previous research has developed various fuzzing tools with different fuzzing algorithms. Atlidakis et al. [9], [10] and Godefroid et al. [11] developed a tool for performing black-box fuzzing on Web APIs called Restler. In fuzzing experiments carried out by the authors, Restler succeeded in finding new errors in Office365, Gitlab, and AzureDNS application services. Arcuri [12] developed a tool called EvoMaster, which can be used to perform black-box and white-box fuzzing on Web APIs. Through fuzzing experiments with EvoMaster, the author managed to find errors in several open-source-based Web APIs. Viglianisi et al. [13] and Corradini et al. [14] developed a tool for black-box fuzzing on Web APIs called RestTestGen. Through fuzzing experiments on several public Web API applications, RestTestGen succeeded in finding errors in several application endpoints. Laranjeiro et al. [15] also developed a black-box fuzzing tool called bBOXRT, which focuses on testing input validation mechanisms in applications. Hatfield-Dodds et al. [16] developed a tool called Schemathesis, which is designed to perform black-box fuzzing quickly with comprehensive findings. Alonso et al. [17] developed ARTE, a fuzzing algorithm designed to generate valid and invalid data inputs based on syntax and semantics. Lei et al. [18] developed a tool called Leif, which can be used to perform black-box fuzzing on Web APIs. In generating test scenarios, Leif uses application input specifications to generate test payloads that match the input type or format, such as string data input, which can have many formats and patterns.

For applications with complex data input structures, such as input consisting of many fields with different data types, current state-of-the-art fuzzing tools are still less effective in exploiting errors, as seen in the experimental results of this study. Apart from that, some state-of-the-art fuzzing tools tend not to exploit further errors when they have succeeded in finding errors or no longer find errors, so the error found is less comprehensive. Because there are still limitations in the current state-of-the-art fuzzing tools, we proposed OffensiveRezzer, a novel fuzzing tool specifically designed for black-box fuzzing on Web APIs. The primary contributions of this study are outlined below:

1) We developed OffensiveRezzer, a novel fuzzing tool for Web API designed to exploit errors effectively and efficiently in each input field so that it can find all possible errors related to data input.
2) We developed a custom Web API application specifically designed as a benchmark to measure the performance of fuzzing tools. This application has complex data input structures and configurations to set the input validation level. The input validation levels that can be set in this application are no validation, partial validation, and full validation. By implementing different levels of validation, we can find out the actual performance of a fuzzing tool.

OffensiveRezzer (short for Offensive RESTful Fuzzer) is a tool we developed to perform black-box fuzzing on Web API applications that implement REST architecture. The main focus of OffensiveRezzer is to test the implementation of input validation mechanisms in the application. In testing the validation mechanism, OffensiveRezzer tries to exploit types of errors related to data input, namely missing required, invalid type, and constraint violation. For missing required errors, OffensiveRezzer will delete the fields in the data input structure individually. For invalid type errors, OffensiveRezzer will change the data type of the input field. For example, suppose the application expects an input field to have a string data type. In that case, OffensiveRezzer will mutate the input field with another data type, for example, an integer, object, or array. For constraint violation errors, OffensiveRezzer will send input with the expected data type but with a value that does not match the specified minimum or maximum constraint.

Missing required, invalid type, and constraint violation mutations will be applied to each input field so that the fuzzing payloads generated by OffensiveRezzer can find hidden errors/bugs in the application. OffensiveRezzer relies heavily on the application specification written in OpenAPI version 3 to create fuzzing payloads. The more complete the specification information, the more comprehensive the fuzzing payload coverage generated by OffensiveRezzer. At present, OffensiveRezzer remains in its initial

prototype phase and is undergoing continuous development. At this stage, OffensiveRezzer can dispatch fuzzing payloads to application endpoints through various HTTP methods such as GET, POST, PATCH, PUT, and DELETE. Fuzzing payloads generated by OffensiveRezzer apply to data input in the request body, query parameters, and path parameters. One of OffensiveRezzer's limitations is that OffensiveRezzer still cannot perform fuzzing on application endpoints that require authentication to be accessed. The source code and how to use OffensiveRezzer can be seen in our GitHub repository [19].

In the context of the OpenAPI specification, input types are generally divided into two main categories, namely primitive types and complex types. Primitive types consist of string, number, integer, and boolean data types, while complex types consist of object and array data types. The object data type consists of several attributes or fields, each with its own value and data type. An attribute in an object can have a value with a primitive or complex data type. For example, if an attribute in an object has a value with the object data type, then the entire object will form a nested object. Then, the array data type contains a collection of elements with the same data type. An array can contain a collection of values with primitive or complex types. For example, if an array contains a collection of values with the object data type where the object has the same structure for each element, then the entire array will form an array of objects. Especially for the string data type, the expected value can contain a specific format or pattern. For example, the string data type may include additional format information with the datetime value indicating that the given input string value must be a valid datetime format. The string data type can also include certain pattern information, which is usually denoted with regular expression syntax.

In creating variations of fuzzing payloads, OffensiveRezzer will first create valid data input. Creating valid data input is an essential initial process, and this valid input will be used as a basis for performing input mutations. For integer and number data types, OffensiveRezzer will generate two value variations, namely value 1 (default) and minimum value (optional), if the specification contains information regarding the minimum value of an input field. For the boolean data type, OffensiveRezzer will randomly generate true or false values. For the string data type, OffensiveRezzer will first generate two value variations, namely 'fuzz' (default) and the character 'f' multiplied by the number of minimum character lengths if the specification contains information regarding the minimum character length. Giving default values and values matching the minimum constraint ensures the generated input is valid. Because the string data type can contain information about formats or patterns, OffensiveRezzer will perform further analysis to be able to generate string values that match specific formats or patterns. OffensiveRezzer will use a regular expression to generate a string value that matches the pattern if there is information about a pattern. If there is information regarding the format and the format is a datetime/date format, OffensiveRezzer will then create a current datetime/date value, which is then converted into a string value with a valid datetime/date format. For the object data type, OffensiveRezzer will create an object that contains various fields and their values according to the data type of each field. For the array data type, OffensiveRezzer will create a data array containing one element with a value according to the specification. Figure 1 shows an example of valid data input generated by OffensiveRezzer based on the definitions in the application specification.

After generating valid data input, OffensiveRezzer will perform various value mutations based on that data input. To be able to mutate values, OffensiveRezzer requires a valid input example along with input specification information. The first mutation variation performed by OffensiveRezzer is the missing required mutation. In performing the missing required mutation, OffensiveRezzer will first copy the generated valid input. From the copy of the valid data, OffensiveRezzer will delete the input fields one by one. For example, suppose an input is an object type consisting of fields A and B. In that case, OffensiveRezzer will generate two fuzzing payloads: a data object that only consists of field B and a data object that only contains field A. Especially for input with the array data type, apart from eliminating the input field, OffensiverRezzer will also mutate the value by assigning a value in the form of an empty array. An empty array is an array that does not have a single element. Figure 2 shows an example of a missing required mutation generated by OffensiverRezzer based on the application specification and the valid input example.

Figure 1. Example of valid input generated by OffensiveRezzer.

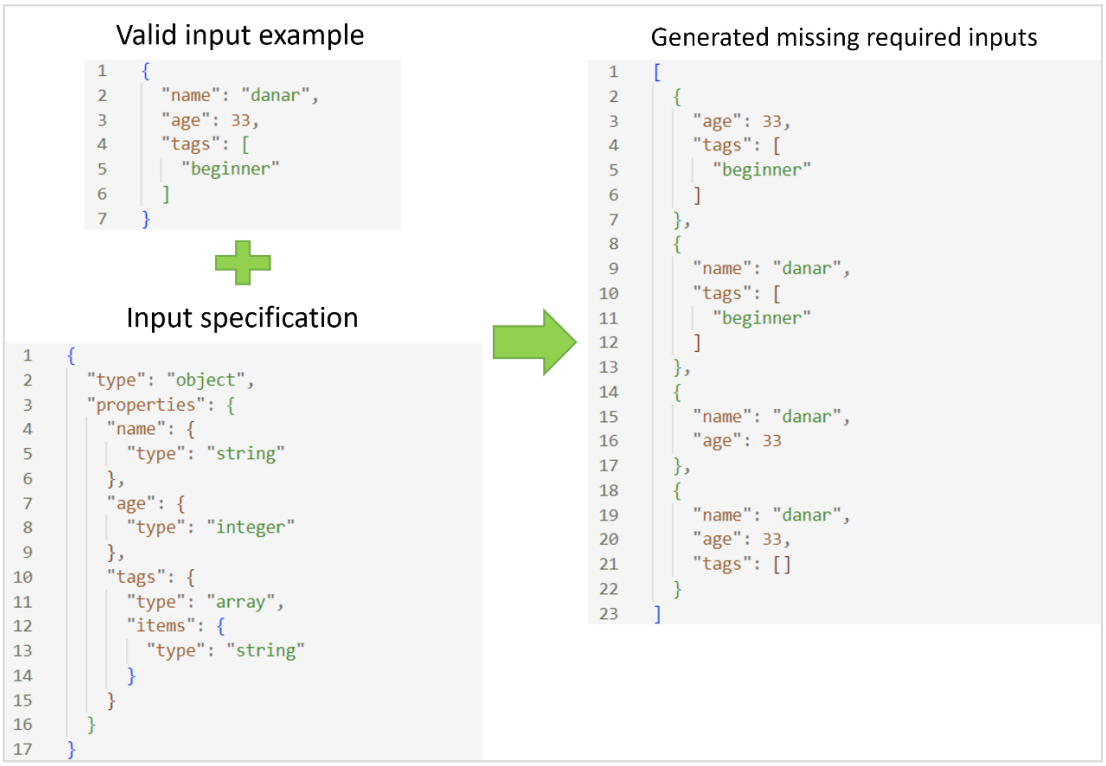

Figure 2. Example of missing field input generated by OffensiveRezzer.

After performing the missing required mutation, the following mutation variation performed by OffensiveRezzer is the invalid type mutation. In performing the invalid type mutation, OffensiveRezzer will also first copy the generated valid input. From the copy of the valid data, OffensiveRezzer will assign values with inappropriate data types to each input field in turn. The invalid type mutation algorithm used by OffensiveRezzer is quite simple. For each input field that has a primitive data type (string, integer, number, and boolean), it will be given a value with the object data type. In contrast, input fields with

complex data types (object and array) will be given a value with the string data type. Each input field with a primitive data type will have its value mutated to "{fuzz: 'fuzz'}" while the input field with a complex data type will have its value mutated to the string "fuzz". Especially for input with an array data type, apart from assigning values with a primitive data type, OffensiveRezzer will also perform invalid type mutations on the elements in the array. For instance, suppose an input has an array type containing string elements. In that case, OffensiveRezzer will perform an invalid type mutation by assigning a value in the form of a data array containing object elements. On the other hand, arrays with complex elements will get an invalid type mutation to an array with primitive elements. Figure 3 shows an example of an invalid type mutation generated by OffensiverRezzer based on the application specification and the valid input example.

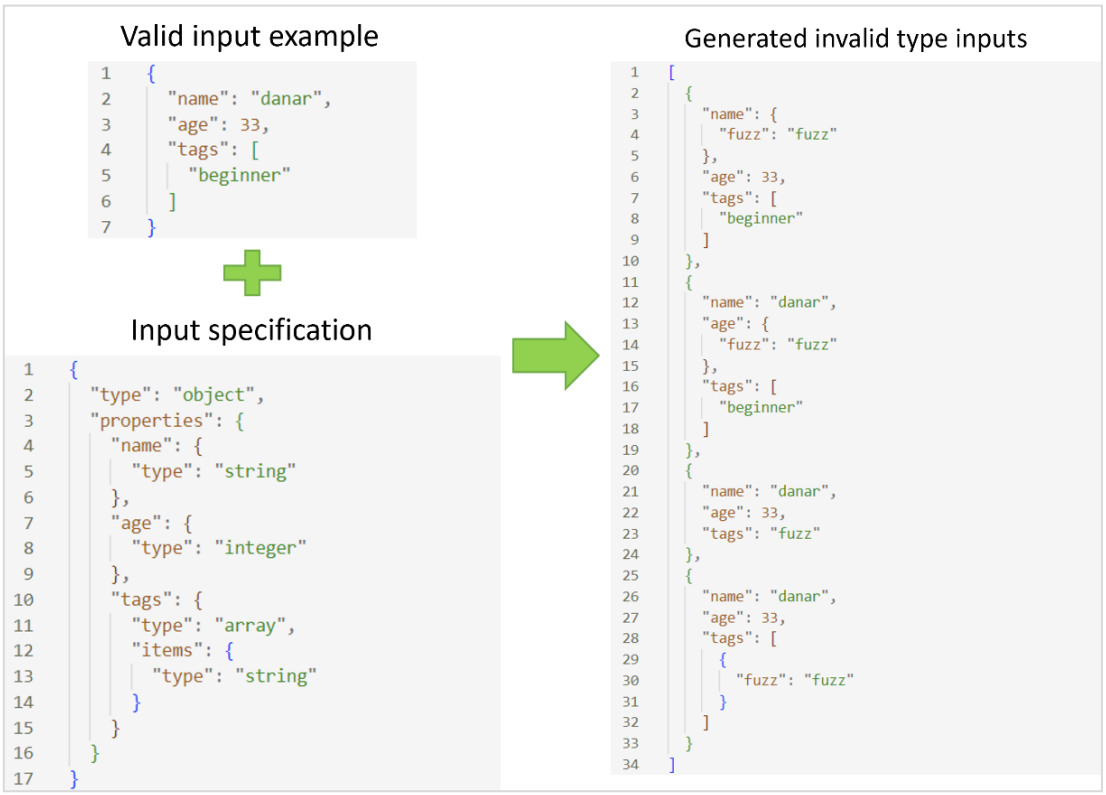

Figure 3. Example of invalid type input generated by OffensiveRezzer.

After performing the invalid type mutation, the final mutation variation performed by OffensiveRezzer is the constraint violation mutation. As the name suggests, the constraint violation mutation will generate input values that do not match the minimum/maximum constraints in the application specification. The constraint violation mutation performed by OffensiveRezzer only applies to string, integer, and number data types. Before performing the constraint violation mutation, OffensiveRezzer will first copy the generated valid data input. From the copy of the valid data, OffensiveRezzer will assign values that do not match the constraints to each string, number, or integer input field in turn. Especially for input with the array data type, if the array elements have the string, number, or integer data type, OffensiveRezzer will also perform the constraint violation mutation on the array elements. The constraint violation mutation algorithm used by OffensiveRezzer is quite simple, namely by assigning extreme values to each input field. For example, for input with a number or integer data type, OffensiveRezzer will generate a huge value like 10^{1000}. Suppose the application specification contains information regarding the minimum/maximum value. In that case, OffensiveRezzer will also assign an input value that is less than the minimum requirement and greater than the maximum requirement. For example, we subtract 10^9 from the minimum allowed value, and add 10^9 to the maximum allowed value. Another example is if an input has a string data type with a datetime format, then OffensiveRezzer will perform a mutation by subtracting and adding 10^5 years to the valid datetime value. Figure 4 shows an example of the constraint violation mutation generated by OffensiverRezzer based on the application specification and the valid input example.

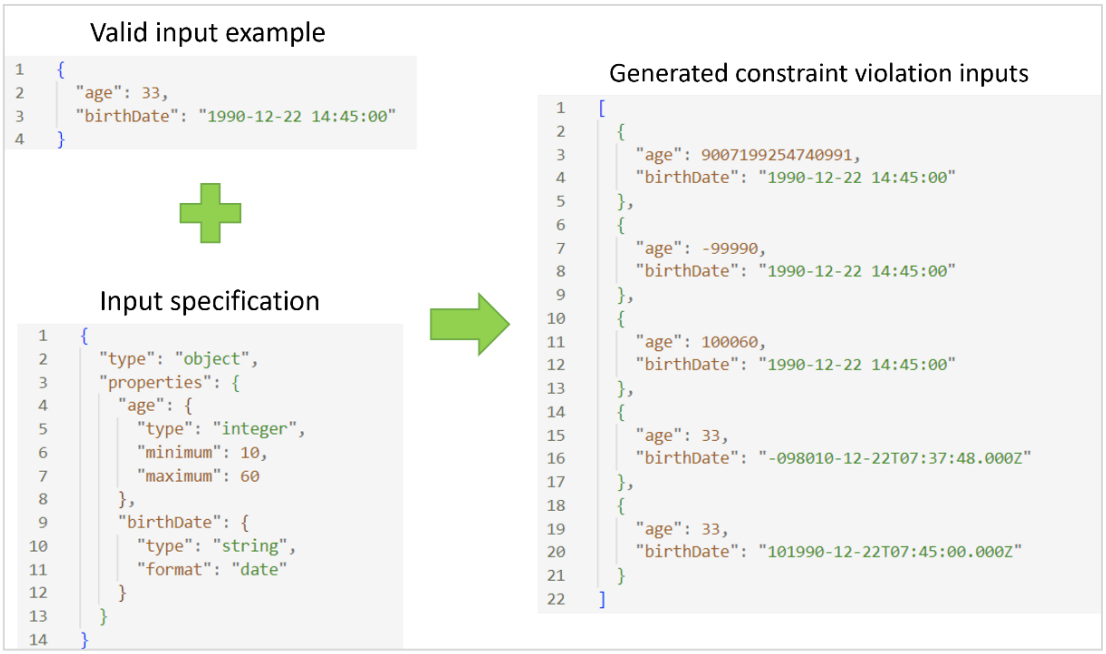

Figure 4. Example of constraint violation input generated by OffensiveRezzer.

METHODS

Tested web API application

The fuzzing target in this experiment was a custom Web API application that we developed ourselves, specifically designed as a benchmark for measuring the performance of fuzzing tools. This application is built with Python, the Flask HTTP framework, and the Pydantic input validation library. This application has configurations to set the level of input validation that will be applied to the application. The input validation levels that can be set are no validation, partial validation, and full validation. In the application with partial validation, it will only check the input type, while full validation will check the data type and constraints of the input. For example, the application has an input field with a string data type, a minimum character length of 3, and a maximum character length of 1000. In this example, partial validation will only check whether the data input received by the application is string-type input without checking the character length of the data input, while full validation will also check whether the received input value matches constraint specifications.

The tested application is an e-commerce Web API that has 7 endpoints that can be accessed via the POST method. Details regarding all application endpoints and application source code can be seen in our GitHub repository [20]. Each endpoint has a different input structure or specification. The endpoint URL name represents the composition of the data input that must be sent. For example, at the */product* endpoint, the client must send data input containing fields related to the *Product* entity. Figure 5 shows an example of data input for the */product* endpoint. Another example is the */product-tag-category* endpoint, which has more complex input specifications than the */product* endpoint. Apart from sending data input containing fields related to the *Product* entity, the client must also send input fields related to the *Tag* and *Category* entities. Figure 6 shows an example of data input for the */product-tag-category* endpoint.

We did not use existing benchmark applications or open-source/real-world applications because we wanted to perform fuzzing experiments on applications with complex input structures and applications with different input validation level implementations. It will be easier for us to build our applications rather than looking for real-world applications that can meet the needs of this research experiment. Performing fuzzing on applications with complex input structures and different validation levels is very important because it can measure the actual performance of fuzzing tools in detecting errors. Another reason is that OffensiveRezzer still has limited features such as not being able to perform fuzzing on endpoints that require authentication. Real-world applications usually have endpoints that require authentication to be accessed.

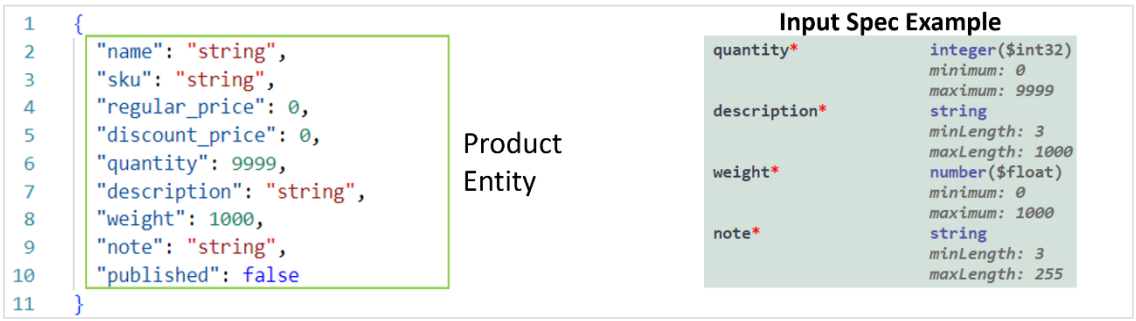

Figure 5. Example and input specifications for endpoint */product*.

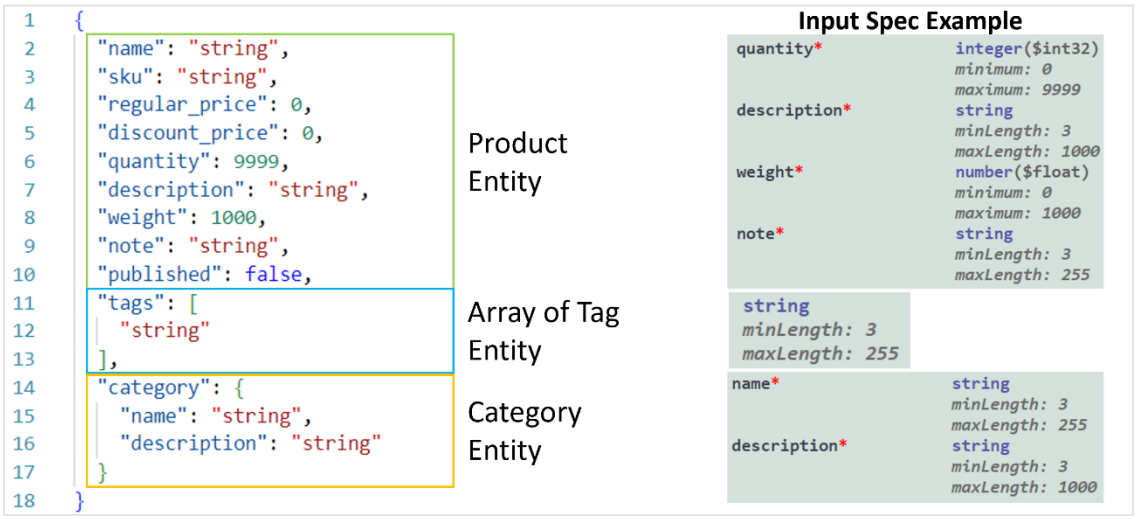

Figure 6. Example and input specifications for endpoint */product-tag-category*.

Fuzzing tools used in the study

To conduct the fuzzing experiment in this study, we employed OffensiveRezzer and several state-of-the-art fuzzing tools, namely Restler v9.0.1 [9], [21], EvoMaster v1.6.1 [12], [22], RestTestGen v23.9 [13], [23], Tcases v4.0.2 [24], and Schemathesis v3.22.1 [16], [25]. Each fuzzing tool will run using its default configuration. For time-based fuzzing tools such as EvoMaster and Restler, the fuzzing process will run for 1 hour. For fuzzing tools that are not time-based, the fuzzing process will be performed 5 times to get consistent results.

Performance metric

The performance metric used in this study was the fuzzing tools' effectiveness level, which was measured by counting the number of distinct or unique errors found in the application. An error condition in the application is identified by an application response that has a status code of 500 or Internal Server Error. Status code 500 indicates the application encountered an unexpected or unhandled error condition. Status code 500 can also indicate that the application is in a crashed condition and could experience Denial of Service if it keeps encountering unhandled errors.

To ascertain the count of distinct errors, we opted not to rely on the fuzzing report produced by individual tools, given their varying formats and methods of error calculation. For instance, EvoMaster solely records the number of endpoints with errors, meaning that if there are multiple distinct error messages within a single endpoint, they are consistently tallied as a single error in the application. Another example is that RestTestGen will report errors as many as the number of fuzzing requests that successfully trigger an error without paying attention to the unique mutation of fuzzing payloads. So, the reported number of errors can be inaccurate and tends to be greater than the actual number of unique errors. In contrast to EvoMaster and RestTestGen, Restler can calculate the number of errors better by paying attention to the unique mutations of fuzzing payloads. OffensiveRezzer itself still doesn't have an accurate error calculation system, and the way the error calculation works is similar to that of RestTestGen.

For precise data regarding the count of distinct errors, we conducted error calculations by analyzing the error log files produced by the target application. The application we developed has a log system that will record every unhandled error and information on the endpoint where the error occurred. The approach of counting distinct errors via error log file analysis was also performed by [16] and [26]. If same error arises at two distinct endpoints, it will be registered as two distinct errors. For instance, if the endpoints */user* and */user-address* display the identical error message, "*Column 'first_name' cannot be null*", it will be considered as two distinct errors within the application. Moreover, we standardized error messages that included input parameter details from the client. If two error messages share the same context but encompass different input parameters, they will be tallied as a single error. For instance, if the */user-address-product* endpoint presents the error messages "*Incorrect decimal value: 'abcdef' for column 'discount_price'* " and "*Incorrect decimal value: '{fuzz: fuzz}' for column 'discount_price'* ", they will be treated as one error. We used Python scripts to count the number of unique errors found by the fuzzing tools. The Python scripts can be viewed in our GitHub repository [27].

Experiment model

Figure 7 shows the experiment model carried out in this research. Each fuzzing tool will perform fuzzing in turn on the target applications. For fuzzing tools that are not time-based, namely OffensiveRezzer, RestTestGen, Tcases, and Schemathesis, the fuzzing process will be carried out in 5 iterations. For time-based fuzzing tools, namely Restler and EvoMaster, the fuzzing process will be carried out for 1 hour. Fuzzing experiments will be carried out in turns on the application without validation, the application with partial validation, and the application with full validation. The target applications are equipped with a log system to record any server errors that occur. The log system will record the error message along with the endpoint location where the error occurred. After all the fuzzing processes are complete, an analysis of the error log file is carried out using Python scripts. This analysis was performed to count the number of unique errors and classify the error types found by each fuzzing tool.

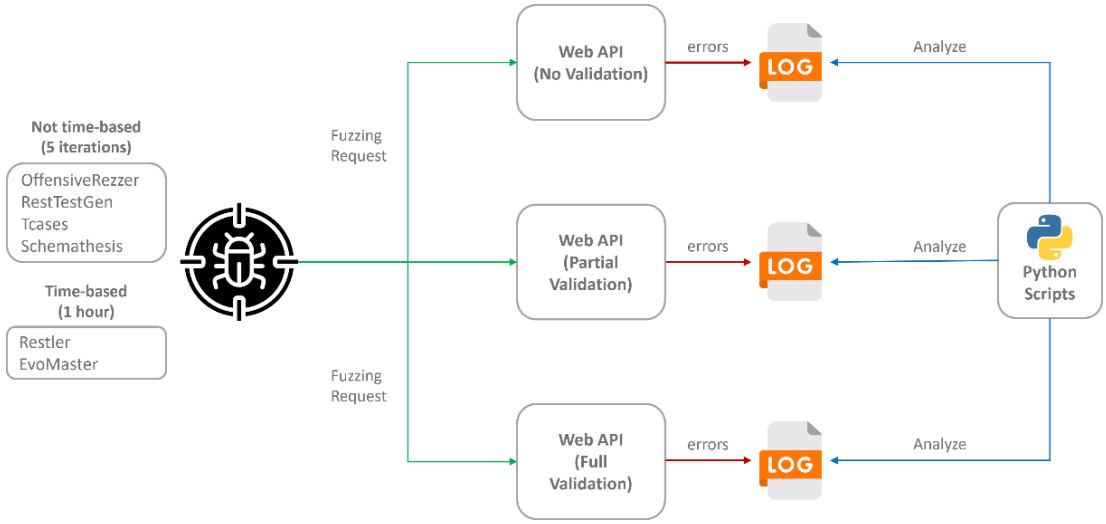

Figure 7. The experiment model carried out in the research.

RESULTS AND DISCUSSIONS

Fuzzing tools performance on application without validation

Figure 8 shows the comparison of distinct error counts detected by fuzzing tools in the application without input validation. In the application without input validation, there are several types of errors that can be found, namely errors caused by the minimum/maximum input length not being in accordance with specifications (constraint violation), errors caused by the missing input fields that must be sent (missing fields), and errors caused by data input types that do not comply with specifications (invalid type). OffensiveRezzer outperformed other fuzzing tools by detecting the highest number of unique errors, followed by Restler, RestTestGen, Tcases, EvoMaster, and Schemathesis. OffensiveRezzer managed to find 55.49% more errors compared to Restler, 136.11% more than RestTestGen, 254.17% more than

Tcases, 264.29% more than EvoMaster, and 537.50% more than Schemathesis. This result showed that OffensiveRezzer most effectively exploited possible errors in each input field.

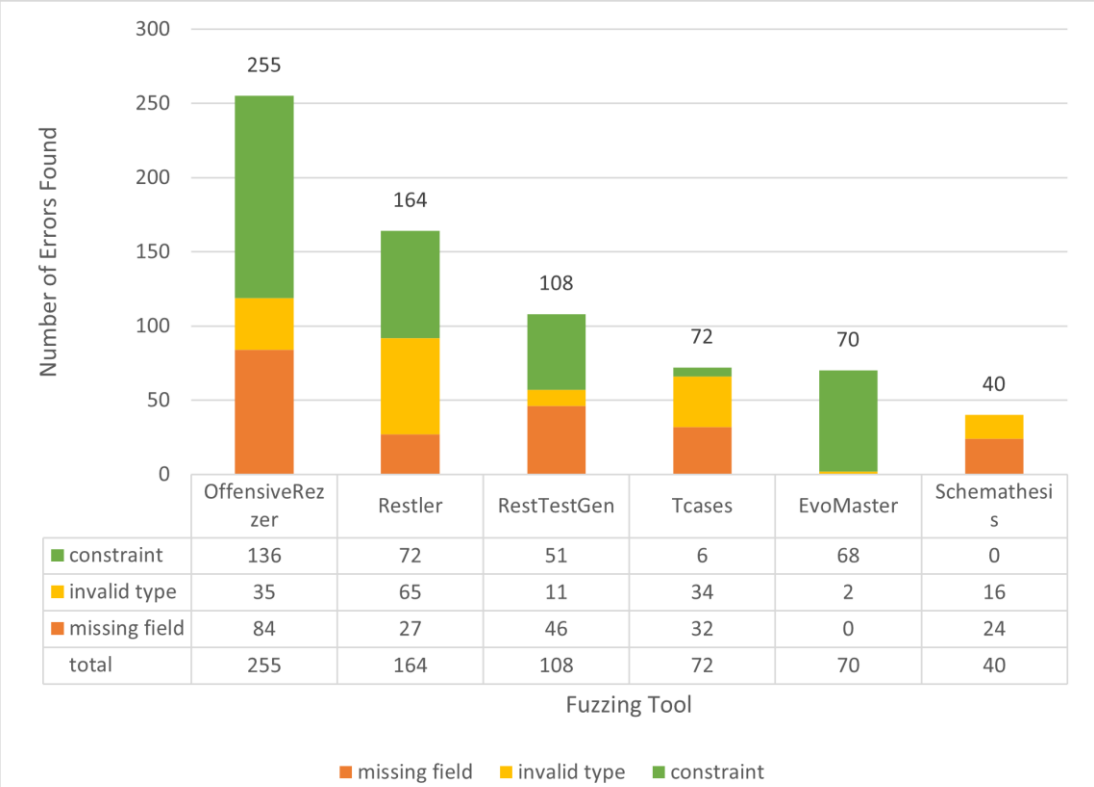

Figure 8. Unique error counts and error types detected by fuzzing tools in the application that has no input validation.

Figure 8 also shows the composition of the error types found by each fuzzing tool so that we can identify the tendency of each fuzzing tool to exploit the error types. For example, OffensiveRezzer and RestTestGen exploited more missing field and constraint violation errors and less exploited invalid type errors. Then, Restler, which was in the second position, exploited more constraint violations and invalid type errors. When compared with OffensiveRezzer, Restler was better at exploiting invalid type errors. Tcases and Schemathesis had the same error type composition, namely exploiting more invalid type and missing field errors and minimal exploiting constraint violation errors. EvoMaster, which was in the fifth position, tended to focus more on exploiting constraint violation errors and did not exploit invalid type and missing field errors at all, whereas, in the application without validation, the invalid type and missing field errors should be straightforward to find.

Fuzzing tool performance on application with partial validation

Figure 9 shows the comparison of distinct error counts detected by fuzzing tools in the application with partial input validation. In the application with partial input validation, the only type of error that can be found is constraint violation error, namely errors caused by the minimum/maximum length of data input that does not comply with specifications. OffensiveRezzer outperformed other fuzzing tools by detecting the highest number of unique errors, followed by Restler, EvoMaster, RestTestGen, Tcases, and Schemathesis.

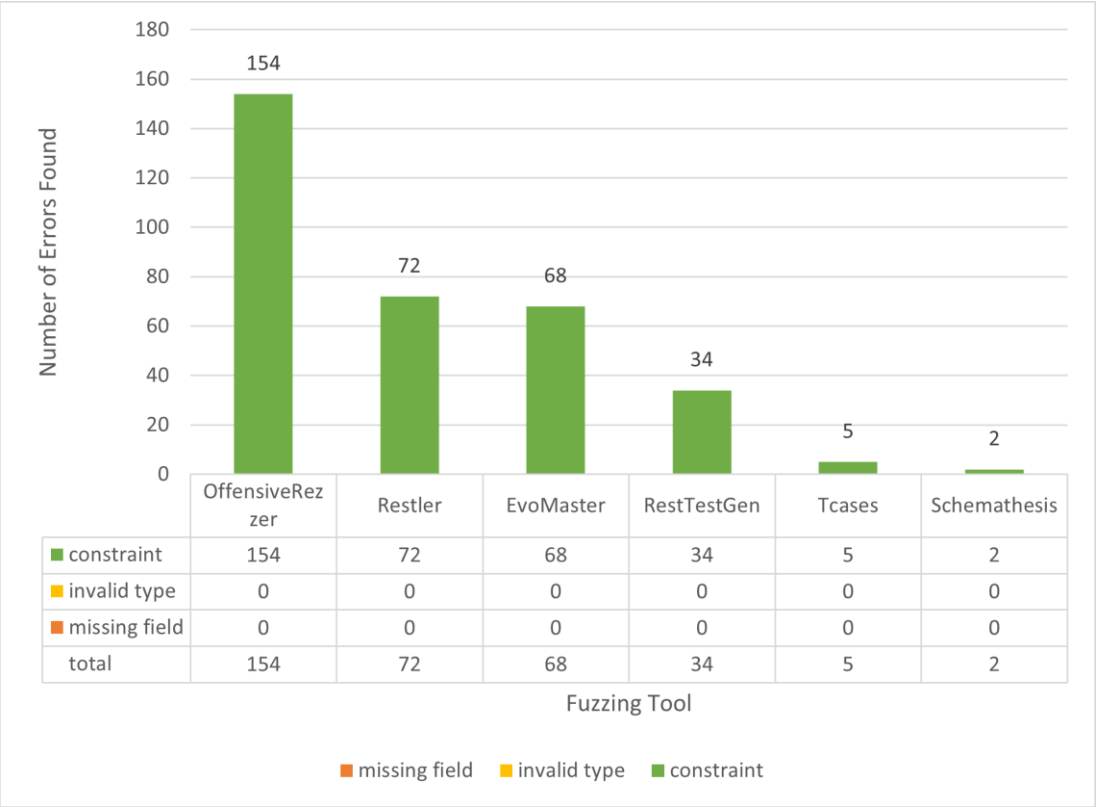

Figure 9. Unique error counts and error types detected by fuzzing tools in the application that has partial input validation.

OffensiveRezzer had the highest number of unique error findings because it exploited constraint violation errors across all fields within the input structure, no matter how complex the structure is. For instance, the */user-address-product-shipping* endpoint has a complex data input structure consisting of fields related to the *User*, *Address*, *Product*, and *Shipping* entities. On this endpoint, OffensiveRezzer successfully applied constraint violation for each field in the *User*, *Address*, *Product*, and *Shipping* entities. On the other hand, Restler, which was in second place, did not exploit the */user-address-product-shipping* endpoint. Restler is a time-based fuzzing tool, and the 1-hour fuzzing time used in this experiment was not enough to perform fuzzing on all application endpoints. Of the total of 7 endpoints, Restler only succeeded in fuzzing 5 endpoints, while the other fuzzing tools all succeeded in fuzzing 7 endpoints. Despite having the lowest test coverage percentage, Restler identified the second-largest number of unique errors after OffensiveRezzer. Restler was also effective in applying constraint violations to each input field. For example, the */product-tag-category-coupon* endpoint is another endpoint that has a complex input structure, as does the */user-address-product-shipping* endpoint. The input structure at the */product-tag-category-coupon* endpoint consists of fields related to the *Product*, *Tag*, *Category*, and *Coupon* entities. At this endpoint, Restler performed fuzzing in the form of constraint violation for each input field in the *Product*, *Tag*, *Category*, and *Coupon* entities. The main limitation of Restler is that it required a long fuzzing duration to be able to fuzz every input field in all application endpoints.

EvoMaster is another fuzzing tool besides Restler which is also time-based. In contrast to Restler, with a fuzzing duration of 1 hour, EvoMaster successfully fuzzed all endpoints in the application. Even though it had 100% endpoint coverage, the number of error findings from EvoMaster was still less than Restler's finding. This showed that EvoMaster did not exploit errors in every field. For instance, on the */user-address-product-shipping* endpoint, EvoMaster only applied constraint violations to several fields in the data input structure. EvoMaster only applied constraint violation to the *first_name* and *last_name* fields in the *User* entity. Meanwhile, the *User* entity has 3 other fields, namely *email*, *phone_code*, and *phone_number*, which are also vulnerable to constraint violations. Further experiments need to be conducted to determine whether the number of error findings from EvoMaster can increase by increasing the fuzzing duration.

RestTestGen is a fuzzing tool that is not time-based, such as OffensiveRezzer, Tcases, and Schemathesis. In this experiment, RestTestGen could only find a small number of constraint violation errors. The small number of error findings showed that RestTestGen also did not exploit errors in every input field. For instance, on the */user-address-product-shipping* endpoint, RestTestGen only applied constraint violations for the *User* entity fields and did not apply constraint violations for other fields in the *Address*, *Product*, and *Shipping* entities. Tcases and Schemathesis are the fuzzing tools that found the fewest constraint violation errors. Tcases only found 5 errors, while Schemathesis found 2 errors. This error finding aligns with the result of fuzzing experiments on the application without validations, where Tcases and Schemathesis focused more on exploiting missing fields and invalid type errors than constraint violation errors.

Fuzzing tool performance on application with full validation

Figure 10 shows the comparison of distinct error counts detected by fuzzing tools in the application with full input validation. In the application with full input validation, errors should no longer be found. However, to test the effectiveness of the fuzzing tool, we deliberately did not apply full validation to some input fields. Some of these input fields are the *charge* field in the *Shipping* entity, the *regular_price* field in the *Product* entity, and the *max_usage* field in the *Coupon* entity. With such a configuration, there are still constraint violation errors that can be found in the application with full input validation, namely at the endpoints */user-address-product-shipping*, */user-address-product*, */product-tag-category-coupon*, */product*, and */product-tag-category*. OffensiveRezzer outperformed other fuzzing tools by detecting the highest number of unique errors, followed by Restler, RestTestGen, Tcases, EvoMaster, and Schemathesis.

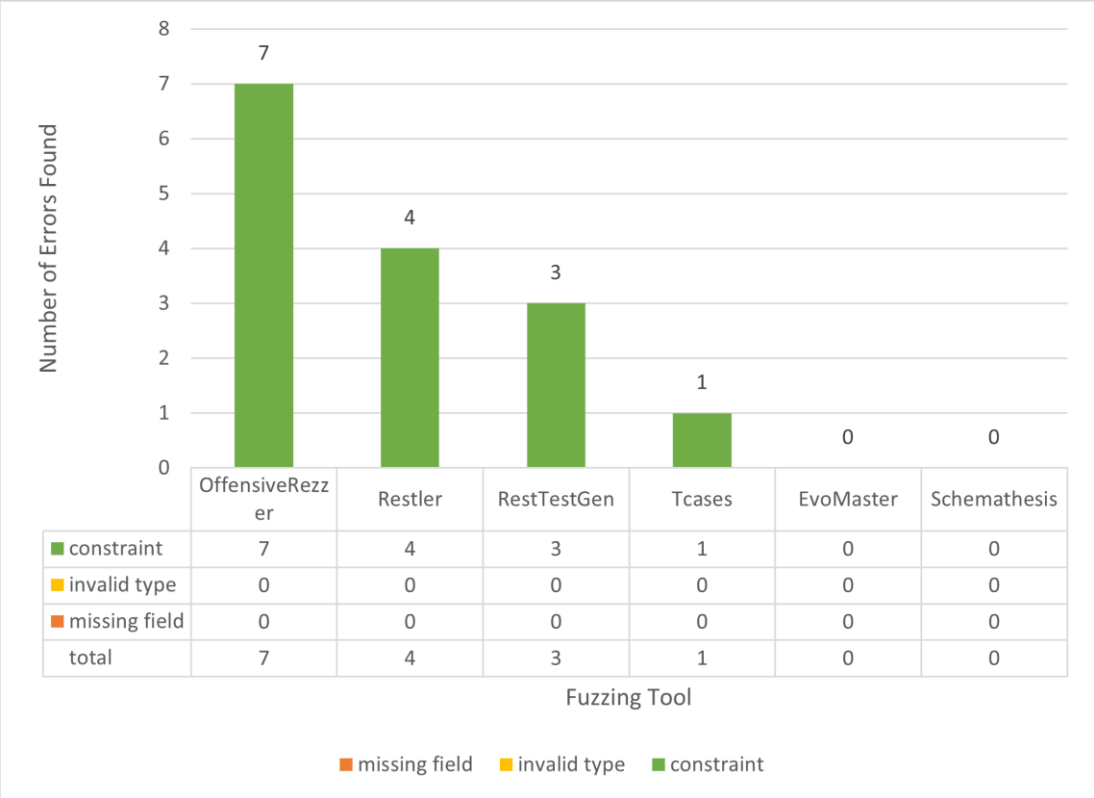

Figure 10. Unique error counts and error types detected by fuzzing tools in the application that has full input validation.

OffensiveRezzer succeeded in finding all remaining errors in the five endpoints related to the *charge* field in the *Shipping* entity, the *regular_price* field in the *Product* entity, and the *max_usage* field in the *Coupon* entity. Restler, who was in second place, only found constraint violation errors for the *regular_price* and *max_usage* fields in the endpoints */product-tag-category-coupon*, */product*, and */product-tag-category*. Restler still failed to find the remaining errors in the endpoints */user-address-product-shipping* and */user-address-product*. As previously explained, the main limitation of Restler is that it required a longer fuzzing

duration to fuzz all application endpoints and find more errors. The third position is occupied by RestTestGen, which only found constraint violation errors for the *regular_price* field (*Product* entity) in the endpoints */product-tag-category-coupon*, */product*, and */product-tag-category*.

Tcases, which is in the fourth position, only found 1 constraint violation error for the *regular_price* field (*Product* entity) in the endpoint */product-tag-category*. The last position was occupied by EvoMaster and Schemathesis, where these two fuzzing tools failed to find any errors. This showed that EvoMaster and Schemathesis did not exploit errors in every input field. Another possible cause is that EvoMaster and Schemathesis did not perform further fuzzing processes when the application kept returning positive responses, so these two fuzzing tools assumed no more errors can be exploited. As a result, these two fuzzing tools failed to find any remaining errors. This is the opposite of how OffensiveRezzer worked. OffensiveRezzer did not analyze the responses received from the application. OffensiveRezzer will keep performing fuzzing based on the number of possible errors that can occur in an input field, where information on possible errors is obtained through analysis of the application specification. Even though it does not perform application response analysis and only relies on specification analysis, OffensiveRezzer succeeded in being the most effective fuzzing tool in this experiment, compared to other fuzzing tools such as EvoMaster, Schemathesis, RestTestGen, and Restler, which use a combination of specification and application response analysis.

In this research, fuzzing experiments were performed on the custom Web API application that we developed ourselves so that the results of this research, namely the effectiveness of fuzzing tools, only applied to the case of this custom Web API application. The effectiveness of fuzzing tools could be different from the results of this study if fuzzing experiments were performed on other applications. Then, in measuring the fuzzing tools' effectiveness level, we only used the unique error counts caused by invalid data input. We did not include other error aspects, such as response data schemas that did not comply with application specifications or invalid dependencies between endpoints. This is because OffensiveRezzer still has limited features and cannot detect these other error aspects.

In this research, OffensiveRezzer became the most effective fuzzing tool compared to state-of-the-art fuzzing tools. Because fuzzing experiments were performed on an application we developed ourselves, this will undoubtedly threaten the validity of this research's results. There may be a bias or suspicion that our application is designed to benefit OffensiveRezzer. To overcome this validity threat, we ensured that all source codes used in this research, namely the OffensiveRezzer source code, the target application, and the error calculation script, were all publicly available and accessible via GitHub. Anyone can check the source codes to determine whether malicious code is implemented to benefit OffensiveRezzer. The application we developed is just a simple application that performs the CREATE operation in the database. In the *Tested Web API Application* subsection, we had also explained why we developed our applications for fuzzing targets and did not use existing applications or benchmarks.

CONCLUSION

In this research, we proposed OffensiveRezzer, a novel black-box fuzzing tool for Web API applications that implement REST architecture. We additionally assessed OffensiveRezzer's effectiveness level against various fuzzing tools, including Restler, Tcases, RestTestGen, Schemathesis, and EvoMaster. The research evaluated the effectiveness of these tools by counting the unique errors detected by each. In the application without input validation mechanisms, OffensiveRezzer emerged as the most effective tool, namely by finding the highest unique errors, trailed by Restler, RestTestGen, Tcases, EvoMaster, and Schemathesis. For fuzzing experiments on the application that has partial input validation, OffensiveRezzer was again the most effective tool, namely by finding the highest unique errors, trailed by Restler, EvoMaster, RestTestGen, Tcases, and Schemathesis. For fuzzing experiments on the application with full input validation, OffensiveRezzer was also the most effective tool. Some tools, such as Schemathesis and EvoMaster, failed to find even one error.

This research showed that despite its limited features, OffensiveRezzer has quite promising performance in exploiting potential errors that may arise in application data input. For future work, we will develop OffensiveRezzer with comprehensive fuzzing features comparable to current state-of-the-art fuzzing tools. We will assess OffensiveRezzer's effectiveness again using real-world applications and other existing benchmark applications.

REFERENCES

[1] H. Subramanian and P. Raj, Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs. Packt Publishing Ltd, 2019.

[2] S. A. Bello et al., "Cloud computing in construction industry: Use cases, benefits and challenges," Automation in Construction, vol. 122, p. 103441, 2021.

[3] A. Ehsan, M. A. M. E. Abuhaliqa, C. Catal, and D. Mishra, "RESTful API testing methodologies: Rationale, challenges, and solution directions," Applied Sciences, vol. 12, no. 9, p. 4369, 2022.

[4] H. Tabrizchi and M. Kuchaki Rafsanjani, "A survey on security challenges in cloud computing: issues, threats, and solutions," The journal of supercomputing, vol. 76, no. 12, pp. 9493–9532, 2020.

[5] H. Assal and S. Chiasson, "'Think secure from the beginning' A Survey with Software Developers," in Proceedings of the 2019 CHI conference on human factors in computing systems, 2019, pp. 1–13.

[6] M. Humayun, N. Jhanjhi, M. F. Almufareh, and M. I. Khalil, "Security threat and vulnerability assessment and measurement in secure software development," Comput. Mater. Contin, vol. 71, pp. 5039–5059, 2022.

[7] P. Godefroid, "Fuzzing: Hack, art, and science," Communications of the ACM, vol. 63, no. 2, pp. 70–76, 2020.

[8] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, "Fuzzing vulnerability discovery techniques: Survey, challenges and future directions," Computers & Security, vol. 120, p. 102813, 2022.

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

[10] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Checking security properties of cloud service REST APIs," in 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 2020, pp. 387–397.

[11] P. Godefroid, B.-Y. Huang, and M. Polishchuk, "Intelligent REST API data fuzzing," in Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, 2020, pp. 725–736.

[12] A. Arcuri, "RESTful API automated test case generation with EvoMaster," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 28, no. 1, pp. 1–37, 2019.

[13] 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), 2020, pp. 142–152.

[14] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato, "Automated black-box testing of nominal and error scenarios in RESTful APIs," Software Testing, Verification and Reliability, vol. 32, no. 5, p. e1808, 2022.

[15] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A black box tool for robustness testing of REST services," IEEE Access, vol. 9, pp. 24738–24754, 2021.

[16] Z. Hatfield-Dodds and D. Dygalo, "Deriving semantics-aware fuzzers from web api schemas," in Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, 2022, pp. 345–346.

[17] J. C. Alonso, A. Martin-Lopez, S. Segura, J. M. García, and A. Ruiz-Cortés, "ARTE: Automated Generation of Realistic Test Inputs for Web APIs," IEEE Transactions on Software Engineering, vol. 49, no. 1, pp. 348–363, 2023, doi: 10.1109/TSE.2022.3150618.

[18] Z. Lei, Y. Chen, Y. Yang, M. Xia, and Z. Qi, "Bootstrapping Automated Testing for RESTful Web Services," IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 1561–1579, 2023, doi: 10.1109/TSE.2022.3182663.

[19] D. G. Putera, "OffensiveRezzer," bungdanar/offensive-rezzer, https://github.com/bungdanar/offensive-rezzer (accessed May 8, 2024).

[20] D. G. Putera, "Python REST Fuzzing," bungdanar/python-rest-fuzzing, https://github.com/bungdanar/python-rest-fuzzing (accessed May 8, 2023).

[21] "RESTler," GitHub, https://github.com/microsoft/restler-fuzzer (accessed May 8, 2024).

[22] "EvoMaster: A Tool For Automatically Generating System-Level Test Cases," GitHub, https://github.com/EMResearch/EvoMaster (accessed May 8, 2024).

[23] "RestTestGen," GitHub, https://github.com/SeUniVr/RestTestGen (accessed May 8,

2023).
[24] "Cornutum/tcases," GitHub, https://github.com/Cornutum/tcases (accessed May 8, 2024).
[25] Z. Hatfield-Dodds and D. Dygalo, "Deriving Semantics-Aware Fuzzers from Web API Schemas," GitHub, Dec. 01, 2021. https://github.com/schemathesis/schemathesis (accessed May 8, 2024).
[26] 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, 2022, pp. 289–301.
[27] D. G. Putera, "Data Processing", bungdanar/data-processing, https://github.com/bungdanar/data-processing (accessed May 8, 2024).