EVALUATION OF FUZZING ON WEB API FROM OFFENSIVE AND DEFENSIVE PERSPECTIVES

Danar Gumilang Putera
Department of Electrical Engineering, Faculty of Engineering, University of Indonesia
Depok, West Java, Indonesia
danar.gumilang@ui.ac.id

Ruki Harwahyu*
Department of Electrical Engineering, Faculty of Engineering, University of Indonesia
Depok, West Java, Indonesia
ruki.hwyu@gmail.com
*Corresponding Author

Abstract

Web API technology has been widely used in various application infrastructures because it allows different application services to interact and communicate via network platforms. Web API allows applications to share functionality and data with others, making it a preferred choice for integration across infrastructures. Despite the benefits of Web API, it is not without its security concerns. Many vulnerabilities arise due to misconfigurations or insufficient security mechanisms, which can be prevented by performing functionality testing. One of the critical functionality tests to do is fuzzing. Fuzzing is a testing method to identify vulnerabilities that emerge from flawed input and business logic validations. In this research, we performed fuzzing experiments from offensive and defensive perspectives. We compared several state-of-the-art fuzzing tools for the offensive approach, namely EvoMaster, Restler, and RestTestGen. For the defensive approach, we compare several state-of-the-art input validation libraries: Joi, Zod, Marshmallow, and Pydantic. The performance metrics used are the fuzzing tool's effectiveness in finding bugs/errors and the validation library's effectiveness in validating fuzzing payloads, which is measured by calculating the percentage of error reduction. Evaluation results show that Restler found the most bugs/errors compared to other fuzzing tools. Then, each validation library has the following effectiveness: Joi 98.34%, Zod 96.68%, Marshmallow 98.04%, and Pydantic 98.04%.

***Keywords*: REST; API; Testing; Fuzzing; Web Service; Input Validation.**

1. Introduction

Web API technology has been widely applied in various application infrastructures because it allows different application services to interact and communicate via network media or the internet [Subramanian and Raj (2019)]. Through Web API, an application can expose functionality or data that other applications can use. Web API has also become standard in application infrastructure implementing microservices architecture [Higginbotham (2021); Mateus-Coelho, Cruz-Cunha, and Ferreira (2021)]. The microservices architecture consists of many small services where these services can communicate with each other to exchange and synchronize data. Web API is also the technology most widely used as an integration platform between application infrastructures. For example, various Internet of Things (IoT) devices can send and receive data from a centralized database server via Web API, making it easier to collect and analyze data from distributed IoT devices. Several cloud services providers, such as Google Cloud Platform (GCP), Microsoft Azure, and Amazon Web Services (AWS), provide management interfaces that can be accessed via Web API so that their customers can manage their cloud resource usage easily [Bello et al. (2021)]. To interact via a Web API, both the provider application and the functionality/data user application must implement a particular communication architecture. Several communication architectures can be implemented in Web API, including RESTful API, GraphQL, and RPC. RESTful API is the most widely used communication architecture and has become the default standard in Web API technology [Ehsan et al. (2022)].

As with web application technology, Web API technology is also vulnerable to various cyber security attacks. Apart from that, the increasingly widespread use of Web API in various application infrastructures has also made threat actors increasingly focus their malicious activities on applications that expose Web API [Hussain et al. (2020)]. Some examples of security vulnerabilities related to Web API that have a significant impact are the CVE-2021-21972 vulnerability in vCenter products [CVE - CVE-2021-21972 (no date); Klyuchnikov (2021)] and the Optus data leak incident in 2022 [Cyber Security Hub (2023); UpGuard (no date)]. Based on examples of security vulnerabilities in vCenter software and Optus, these vulnerabilities are caused by misconfiguration and poor security mechanism implementation. In the case of vCenter, critical endpoints can be accessed without authentication and have insecure input validation mechanisms that allow remote code execution. The Optus customer data leak incident was also caused by the absence of authentication in one of its API services. Misconfiguration or poor implementation of security mechanisms often escapes application developers' attention [Assal and Chiasson (2019); Tabrizchi and Rafsanjani (2020)]. If we look at the latest version of the Open Web Application Security Project (OWASP) Top 10, vulnerabilities caused by poor implementation of security mechanisms still occupy the top positions. For example, vulnerabilities caused by insecure authentication and authorization rank first in the OWASP Top 10 (Broken Access Control) [A01 Broken Access Control - OWASP Top 10:2021 (no date)]. Then, vulnerabilities caused by insecure input validation/ sanitization mechanisms rank third in the OWASP Top 10 (Injection) [A03 Injection - OWASP Top 10:2021 (no date)].

Vulnerabilities caused by misconfiguration and poor implementation of security mechanisms can be prevented before the application is deployed, namely by testing the application's functionality and security. Good security testing should cover scenarios that can trigger unexpected errors in the application [Humayun et al. (2022)]. One of the critical functionality tests to do is fuzzing (fuzz testing). Fuzzing is a testing method to identify and prevent various security vulnerabilities caused by insecure input and business logic validation mechanisms. In fuzzing, the application will be given various data inputs in the form of random data input, data input that does not match with the specification, and data input that is embedded with malicious content to find out whether the application can prevent malicious data that could cause damage on application [Godefroid (2020)].

Besides testing input and business logic validation mechanisms, fuzzing can also be used to test the authentication implementation. For example, in a Web API application, we can send various fuzzing requests that do not have authentication credentials to several endpoints on the Web API that require authentication to be accessed. If the fuzzing request returns an authentication error or the Web API does not return data, then the application has implemented a proper authentication mechanism. On the other hand, if the fuzzing request does not cause an authentication error or the Web API still returns response data, then the application has not implemented a secure authentication mechanism. Fuzzing can be done using several methods, namely black-box and white-box methods [Beaman et al. (2022)]. Black-box testing is functionality testing that is limited to program specifications such as input and output formats and cannot detect the internal processes of the application. On the other hand, white-box testing is a testing method that is performed at the application's source code level to understand the application's internal processes and generate better test scenarios compared to the black-box method. Even though it generates more comprehensive test coverage, the white-box method requires more time and costs than the black-box method.

Regarding fuzzing in Web API applications, many previous studies have discussed this topic. Previous studies all focused on fuzzing experiments from the offensive side, namely developing effective fuzzing tools or fuzzing algorithms. There has yet to be any previous research that discusses fuzzing from the defensive side, namely, how to protect applications from fuzzing attacks because fuzzing is not only performed as legal functionality testing but is also used by threat actors to look for vulnerabilities in applications. Because there are still limitations in previous research, we performed fuzzing experiments from the offensive and defensive sides in this research. The main contributions of this research are as follows:

(1) We performed fuzzing experiments with offensive and defensive approaches, whereas, in previous research, no one had discussed fuzzing from a defensive perspective.
(2) For the defensive approach, we developed a Web API application that installed several state-of-the-art validation libraries. The validation mechanism is applied at different levels, namely without validation, partial validation, and full validation. Next, the effectiveness of each validation library in validating fuzzing payloads is measured.
(3) For the offensive approach, we compared several state-of-the-art fuzzing tools and measured their effectiveness in fuzzing the application we developed, which had different levels of validation mechanisms.

2. Related Works

Atlidakis, Godefroid, and Polishchuk (2019) developed a fuzzing tool called Restler, designed to perform black-box testing on Web API applications that use RESTful architecture. The main feature of Retler is testing the input and service dependency validation. Restler can generate various variations of fuzzing payload

automatically based on the data input examples provided. To test the effectiveness of Restler, the author performed a fuzzing experiment on Gitlab and Office365 services. The experimental results show that Restler succeeded in finding new bugs/errors in Gitlab and Office365 services, which were marked by response with status code 500 (Internal Server Error). A response with a status code of 500 indicates that the application cannot handle the error that occurred. Errors/bugs found in Gitlab and Office365 services are caused by poor input validation mechanisms and invalid resource dependencies, such as accessing data that no longer exists or has been deleted.

Arcuri (2019) developed a fuzzing tool called EvoMaster, designed to perform black-box and white-box testing on Web API applications that use RESTful architecture. EvoMaster can analyze application specifications written in OpenAPI format and application source code, which is then used to generate fuzzing test cases. The test cases generated by EvoMaster consist of valid input and invalid input payloads, as well as dependencies between endpoints in the application. To test the effectiveness of EvoMaster, the author performed a fuzzing experiment on five open-source Web API applications written in Java with a total of 171 endpoints. The experimental results show that EvoMaster succeeded in finding errors/bugs on 80 endpoints, which were marked by response with status code 500 (Internal Server Error).

Atlidakis, Godefroid, and Polishchuk (2020) redeveloped Restler by implementing a new fuzzing algorithm and security rule algorithm so that the coverage of fuzzing scenarios generated by Restler is more comprehensive and can find more errors. To test the effectiveness of Restler's new algorithms, the author performed a fuzzing experiment on Azure and Office365 services. The experimental results show that Restler succeeded in finding new errors in Azure and Office365 services. These errors are marked by responses with status code 500 (Internal Server Error), which indicates that the application cannot handle the error that occurred. Server errors can cause the service to crash and become inaccessible. The errors that occur in Azure and Office365 services are caused by improper security rule validation mechanisms.

Godefroid, Huang, and Polishchuk (2020) developed a fuzzing schema that can be used to generate more complex fuzzing payloads so that the generated test cases can cover all possible errors that can occur in a Web API application. The fuzzing schema focuses on modifying the data input structure, which consists of deleting one field (Drop), selecting one field and deleting another field (Select), duplicating one field (Copy), and changing the field's data type (Type). This fuzzing schema is then integrated into the Restler fuzzing tool. To test the effectiveness of the fuzzing schema, the author performed a fuzzing experiment on the Azure DNS service. Experimental results show that Restler combined with the Drop|Select|Copy|Type fuzzing schema found new errors in the Azure DNS service, marked by responses with status code 500 (Internal Server Error). These errors are caused by poor input and business logic validation implementation in the Azure DNS service.

Viglianisi, Dallago, and Ceccato (2020) developed a fuzzing tool called RestTestGen, designed to perform black-box testing on Web API applications that use RESTful architecture. RestTestGen can read application specifications and generate fuzzing test cases based on that specification. The test cases generated by RestTestGen are in the form of valid and invalid input payloads. RestTestGen will modify the input payloads based on the response received from the application. Apart from that, RestTestGen can also generate test cases to test dependencies between endpoints in the application. To test the effectiveness of RestTestGen, the author performed a fuzzing experiment on 87 public Web APIs with a total of 2617 endpoints. The experimental results showed that from 2617 endpoints, RestTestGen found 151 endpoints that responded with status code 500 (Internal Server Error).

Kim et al. (2022) performed an experiment in the form of a effectiveness comparison between 10 RESTful API fuzzing tools. All these fuzzing tools generally have the same features, namely being able to read application specifications written in swagger/OpenAPI format and generate fuzzing payloads and test cases based on the specification. The fuzzing experiment was performed on 20 open-source Web API applications written in Java/Kotlin. The performance metrics used in this experiment are the number of errors found in the application, which are indicated by responses with status code 500 (Internal Server Error), and the percentage of code coverage.

Zhang and Arcuri (2023) performed an experiment in the form of a effectiveness comparison between 7 RESTful API fuzzing tools. The experiment was performed using the black-box method. Performance metrics used in this research are the percentage of code coverage and the number of error types found. The fuzzing experiment was performed on 20 open-source Web API applications written in Java/Kotlin and NodeJs.

Lei et al. (2023) developed a fuzzing tool called Leif, which is designed to test input validation functionality in Web API applications that use RESTful architecture. Leif uses an algorithm called Format-encoded Type or FET to generate fuzzing payloads. The main feature of the FET algorithm is to try as best as possible to translate the type and format of an input parameter, especially string input parameters. To test the effectiveness of Leif and the FET algorithm, the author performed fuzzing experiments on 27 commercial Web API applications. The author also performed effectiveness comparison between Leif and several state-of-the-art fuzzing tools, namely BurpSuite, Fuzzapi, and NaiveFuzzer. The experimental results show that Leif succeeded in finding the most bugs/errors (responses with status code 500) compared to other fuzzing tools.

3. Methodology

The purpose of the experiment performed in this research is to answer the following questions:

(1) What is the effectiveness of fuzzing tools in detecting errors in Web API applications that do not have validation mechanism?
(2) What is the effectiveness of fuzzing tools in detecting errors in Web API applications with partial validation and full validation mechanisms?
(3) What is the effectiveness of input validation libraries in validating fuzzing payloads?

3.1. *Tested Web API Application*

To be able to answer the research questions mentioned previously, we first built a custom Web API application using JavaScript/NodeJs and Python, where both applications have the same features or endpoints. The selection of JavaScript and Python was based on a survey from Stack Overflow in 2023, where, based on the survey result, JavaScript and Python were the most popular or widely used technologies. For the JavaScript/NodeJs-based application, we use Express as the HTTP framework and Sequelize as a mediator between the application server and the database. We use Flask as the HTTP framework and SQLAlchemy as a mediator between the application server and the database for the Python-based application.

We also use several state-of-the-art validation libraries to implement validation mechanisms in the application. The validation mechanism is implemented using two modes, namely partial validation and full validation. In partial validation, the validation only checks the data type and does not check the size constraints of the data. For example, suppose the expected input is an integer data type with a minimum value of 0 and a maximum value of 100. In that case, the partial validation mode only checks whether the data input is integer or not and does not check the minimum or maximum size of the data. For full validation mode, validation is also performed on the minimum/maximum size of the data, in addition to validating the data type.

If an error occurs in the application, such as an unexpected or database error, the error will be recorded in the system log, and then the application will respond with status code 500 (Internal Server Error). The source code for the applications we develop is publicly available and accessible via GitHub repositories [a]. The application is a simple e-commerce application that has seven endpoints. Details of application endpoints can be seen in Table 1.

Table 1. List of Resource Endpoints from Tested Application.

Method	Endpoint (Path)	Description
POST	/product	Create a new Product
POST	/product-tag-category	Create a new Product along with its Tags and Category
POST	/product-tag-category-coupon	Create a new Product along with its Tags, Categories, and Coupons
POST	/user	Create a new User
POST	/user-address	Create a new User along with its Address
POST	/user-address-product	Create a new User along with its Addresses and its sold Product
POST	/user-address-product-shipping	Create a new User along with its Addresses and its sold Product (along with the Product's Shipping method)

3.2. *Fuzzing Tools Used in the Research*

RESTler [Atlidakis, Godefroid, and Polishchuk (2019); microsoft/restler-fuzzer (no date)] is designed to perform black-box fuzzing on Web API applications that implement RESTful architecture. RESTler is designed to find various errors and bugs in applications caused by invalid parameters and request bodies, resources that can still be accessed even though they have been deleted or protected with authentication/authorization, and invalid dependency between endpoints. Before performing fuzzing, RESTler needs to compile the application's OpenAPI specification to create a grammar base and data dictionary for the fuzzing process. The data dictionary is a collection of request parameters and payloads with valid and invalid values. RESTler will first use the parameters/payload in the data dictionary when performing the fuzzing process. Then, in the subsequent fuzzing request, RESTler will modify the request parameters. Parameter modification includes combining several invalid inputs, changing data types, removing or adding input that does not match specifications, and passing parameters that do not match input constraints. The test report generated by RESTler includes the number of 500 unique errors found, examples of parameters/payloads that can trigger errors, and scripts to replicate error scenarios in the application.

EvoMaster [Arcuri (2019); EMResearch/EvoMaster (no date)] is a tool for fuzzing Web API applications using black-box and white-box methods. EvoMaster can be used to perform fuzzing on Web API applications that implement RESTful, GraphQL, and RPC architectures. EvoMaster is a fuzzing tool based on Artificial Intelligence (AI), which uses evolutionary algorithms and dynamic program analysis to generate effective test scenarios. EvoMaster will continue to evolve input parameters based on specific criteria until it finds parameter

[a] https://github.com/bungdanar/nodejs-rest-fuzzing ; https://github.com/bungdanar/python-rest-fuzzing

variations with a high probability of triggering errors in the application. To speed up the evolution process in producing effective fuzzing scenarios, EvoMaster also observes the application response, whether the response is an error response or not. The main goal of EvoMaster is to find various potential errors in the application while reducing the time needed for fuzzing. Currently, EvoMaster supports white-box testing for Web API applications written in Java/Kotlin, JavaScript, and C#. In white-box testing, EvoMaster can also intercept and analyze application communication with the SQL database to broaden the generated test scenarios and include tests specific to SQL transactions. The test report generated by EvoMaster includes the number of error endpoints in the application and unit test scripts that can be run individually to replicate test scenarios that could trigger errors in the application.

RestTestGen [Viglianisi, Dallago, and Ceccato (2020); SeUniVr/RestTestGen (no date)] is a tool and framework designed to perform black-box fuzzing on Web API applications that implement RESTful architecture. RestTestGen combines several testing strategies to find bugs and vulnerabilities in Web API applications. RestTestGen also provides options for implementing custom testing strategies so researchers and application developers can perform fuzzing using specific parameters or requirements according to application testing needs. RestTestGen will create input parameters randomly based on default values or input examples in the application specification. Next, RestTestGen will modify the input parameters by providing invalid input, such as changing the data type, sending values that do not match the constraints, and eliminating several required input fields. RestTestGen also modifies input parameters based on application responses to generate input parameters that are better at triggering errors in the application. RestTestGen can also utilize Natural Language Processing (NLP) technology to translate additional information in the application specification to generate more varied test input parameters. The test report generated by RestTestGen includes the number of errors marked by application responses with status code 500, examples of input parameters that trigger errors, and test scripts that can be run to replicate error scenarios.

3.3. *Input Validation Libraries Used in the Research*

3.3.1. Joi

Joi [Joi.dev (no date)] is a validation library in the JavaScript/NodeJs ecosystem designed to validate and ensure the integrity of data entering the application. Joi can be easily integrated into several web frameworks, such as Express or Hapi, so it can be used to validate incoming HTTP requests. Joi uses the concept of schema, which is a representation of validation rules for a particular data structure. Validation schema can be a simple schema, such as a schema for validating strings, or a complex schema used to validate an object that consists of many fields.

Application developers can define validation rules for data types such as string, number, and date. For example, application developers can determine the maximum character length for string data, minimum and maximum values for number data, and specific formats for string data, such as email format. The Joi validation schema will validate data structure and additional rules that have been previously defined. If a validation error occurs in the data input, Joi will return an error containing information about any fields that do not match the validation schema. In the context of a web API application, the validation error message can be returned to the client as a response with status code 400 (Bad Request), which means there is an error from the client side.

```
19  private static plainProductCreatePartialJoiValidation: Partial<
20    Record<keyof ProductCreatePayload, Joi.Schema>
21  > = {
22    name: Joi.string().required(),
23    sku: Joi.string().required(),
24    regular_price: Joi.number().required(),
25    discount_price: Joi.number().required(),
26    quantity: Joi.number().integer().required(),
27    description: Joi.string().required(),
28    weight: Joi.number().required(),
29    note: Joi.string().required(),
30    published: Joi.boolean(),
31  }
```

Fig 1. Partial Validation Schema with Joi for Product Data.

```
85  private static plainProductCreateFullJoiValidation: Partial<
86    Record<keyof ProductCreatePayload, Joi.Schema>
87  > = {
88    name: Joi.string().min(3).max(255).required(),
89    sku: Joi.string().min(3).max(255).required(),
90    regular_price: Joi.number().precision(4).min(0).required(),
91    discount_price: Joi.number()
92      .precision(4)
93      .min(0)
94      .max(Joi.ref('regular_price'))
95      .required(),
96    quantity: Joi.number().integer().min(0).max(9999).required(),
97    description: Joi.string().min(3).max(1000).required(),
98    weight: Joi.number().precision(4).min(0).max(1000).required(),
99    note: Joi.string().min(3).max(255).required(),
100   published: Joi.boolean(),
101 }
```

Fig 2. Full Validation Schema with Joi for Product Data.

Figures 1 and 2 show examples of partial and full validation implementations for the Product entity using Joi. In the partial validation schema, to create a new product, the client must send data containing the quantity field with the definition "Joi.number().integer().required()", which means the quantity field must exist and must be in the form of number without decimal value. Then, in the full validation schema, the definition of the quantity field becomes stricter, namely "Joi.number().integer().min(0).max(9999).required()", which means that the quantity field must exist, must be in the form of number without decimal value, and has a minimum value of 0 and maximum value of 9999.

3.3.2. Zod

Zod [zod.dev (no date)] is another input validation library in the JavaScript/NodeJs ecosystem written entirely using TypeScript. Zod also uses the schema concept to create various validation rules. Zod is a library designed to validate data input from the outside application while ensuring the code base remains strongly typed. Zod uses the static type system from TypeScript to run validation schema during compilation (compile-time) and reduce unexpected errors at runtime. Zod has a declarative syntax so that application developers can easily define validation schema, starting from a simple schema, such as a schema for validating individual strings/numbers, to a complex schema, such as a schema for validating objects, arrays, and nested objects. The Zod schema that has been defined can also be combined with other schemas, making it very easy for application developers to create complex schemas by utilizing existing schemas.

```
13  private static couponCreatePartialZodValidationSchema = z.object({
14    code: z.string(),
15    description: z.string(),
16    discount_value: z.coerce.number(),
17    discount_type: z.string(),
18    times_used: z.coerce.number().int().optional(),
19    max_usage: z.coerce.number().int(),
20    start_date: z.coerce.date(),
21    end_date: z.coerce.date(),
22  })
```

Fig 3. Partial Validation Schema with Zod for Coupon Data.

```
33  private static couponCreateFullZodValidationSchema = z
34    .object({
35      code: z.string().min(3).max(255),
36      description: z.string().min(3).max(1000),
37      discount_value: z.coerce.number().nonnegative().lte(100),
38      discount_type: z.string().min(3).max(255),
39      times_used: z.coerce.number().int().nonnegative().optional(),
40      max_usage: z.coerce.number().int().nonnegative(),
41      start_date: z.coerce.date(),
42      end_date: z.coerce.date(),
43    })
44    .refine(
45      (data) => {
46        if (data.times_used) {
47          return data.times_used <= data.max_usage
48        } else {
49          return true
50        }
51      },
52      {
53        path: ['times_used'],
54        message: 'Must be less than or equal to max_usage',
55      }
56    )
57    .refine((data) => data.end_date >= data.start_date, {
58      path: ['end_date'],
59      message: 'Must be greater than or equal to start_date',
60    })
```

Fig 4. Full Validation Schema with Zod for Coupon Data.

The Zod schema will validate the input's data type and additional constraints such as character length for string data, minimum and maximum values for data numbers, or specific patterns for string data. If the data input received does not match the validation schema, Zod will return an error containing information about any fields that violate validation rules. In the context of a web API application, the error message from Zod can be returned to the client as a response with status code 400 (Bad Request), which indicates an error in the data input provided by the client. Figures 3 and 4 show examples of partial and full validation implementations for the Coupon entity using Zod. In partial validation schema, to create a new coupon, the client must send data that contains the times_used field with the definition "z.coerce.number().int().optional()", which means the times_used field does not have to be present or optional and must be a number without decimal value. Then, in the full validation schema, the definition of the times_used field becomes stricter, namely "z.coerce.number().int().nonnegative().optional()", which means that the times_used field does not have to be present or optional, it must be in the form number without decimal value, and cannot be negative or in other words have a minimum value of 0. In Figure 4, we can also see the Zod schema's refinement process, characterized by the "refine" method. The refinement process in the Zod schema is usually performed to add or customize business logic validation. For example, this refinement has a validation implementation for the times_used field, which cannot exceed the value of the max_usage field.

3.3.3. Marshmallow

Marshmallow [marshmallow (no date)] is one of the libraries in the Python ecosystem that focuses on serialization, deserialization, and data validation. Marshmallow can be used in web applications to handle incoming HTTP requests and outgoing HTTP responses. Marshmallow uses a declarative implementation in defining the data schema. The Marshmallow data schema includes data structure, type, and additional constraints, such as minimum and maximum values. Marshmallow schema will parse or deserialize data from the outside application and attempt to convert it into a Python object. After the deserialization process succeeds, Marshmallow will validate the constraints on the data object. If an error occurs in the deserialization or validation process, Marshmallow will return an error containing information about any fields or properties that do not match the validation schema. In the context of a web API application, error information can be returned to the client as a response with status code 400 (Bad Request), which indicates an error in the data input provided by the client.

```
131  class UserCreatePartialMaValidation(Schema):
132      first_name = fields.Str(required=True)
133      last_name = fields.Str(required=True)
134      email = fields.Str(required=True)
135      phone_code = fields.Str(required=True)
136      phone_number = fields.Str(required=True)
137
```

Fig 5. Partial Validation Schema with Marshmallow for User Data.

```
169  class UserCreateFullMaValidation(Schema):
170      first_name = fields.Str(
171          required=True, validate=validate.Length(min=3, max=255))
172      last_name = fields.Str(
173          required=True, validate=validate.Length(min=3, max=255))
174      email = fields.Email(required=True, validate=validate.Length(max=255))
175      phone_code = fields.Str(
176          required=True, validate=validate.Regexp('^[0-9]{1,3}$'))
177      phone_number = fields.Str(
178          required=True, validate=validate.Regexp('^[0-9]{4,12}$'))
179
```

Fig 6. Full Validation Schema with Marshmallow for User Data.

Figures 5 and 6 show examples of partial and full validation implementations for the User entity using Marshmallow. In partial validation schema, to create a new user, the client must send data containing the email field with the definition "fields.Str(required=True)", meaning the email field must exist and be a string. Then, in the full validation schema, the definition of the email field becomes stricter, namely "fields.Email(required=True,validate=validate.Length(max=255))", which means that the email field must exist, it must be in the form of a string with valid email pattern, and the maximum allowed character length is 255.

3.3.4. Pydantic

Pydantic [Pydantic (no date)] is another Python library besides Marshmallow, designed to handle serialization, deserialization, and data validation. Pydantic also uses the concept of schema to define data structure and validation. The main difference and advantage of Pydantic compared to other libraries is that Pydantic utilizes the Type Hint feature, a built-in Python feature to define validation schema. Using Type Hint, writing the Pydantic schema becomes more straightforward, concise, and readable. Pydantic can also be integrated with various web frameworks in the Python ecosystem, such as Flask and Django, so it can be used to handle the deserialization and validation of incoming HTTP requests. Pydantic will parse the external data input into a Python object and then validate that object. If an error occurs during the deserialization or validation process, Pydantic will return an error containing information about any fields that do not match the validation schema. The error message returned by Pydantic can be forwarded to the client as a response with status code 400 (Bad Request).

```
144  class ShippingCreatePartialPydanticValidation(BaseModel):
145      model_config = ConfigDict(extra='forbid')
146
147      description: str
148      charge: Decimal
149      free: bool = False
150      estimated_days: int
151
```

Fig 7. Partial Validation Schema with Pydantic for Shipping Data.

```
189  class ShippingCreateFullPydanticValidation(BaseModel):
190      model_config = ConfigDict(extra='forbid')
191
192      description: str = Field(min_length=3, max_length=1000)
193      charge: Decimal = Field(ge=0, decimal_places=4)
194      free: bool = False
195      estimated_days: int = Field(ge=0, le=8)
196
```

Fig 8. Full Validation Schema with Pydantic for Shipping Data.

Figures 7 and 8 show examples of partial and full validation implementations for the Shipping entity using Pydantic. In the partial validation schema, to create new shipping data, the client must send data containing the charge field with the definition "charge: Decimal", which means the charge data must exist and must be in the form of a number either without decimal value or with decimal value. Then, in the full validation schema, the definition of the charge field becomes stricter, namely "charge: Decimal = Field(ge=0, decimal_places=4)", which means that the charge field must exist, it must be in the form of a number either without decimal value or with decimal value, must be greater than or equal to 0, and can store numbers with decimal values up to 4 digits.

3.4. *Experiment Setup*

To perform the experiment in this research, we used a computer with the following specifications: Windows 11 operating system, inter-core i7 1.7GHz 12 cores processor, and 32GB memory. We also use MySQL v8.1.0 as the database server. The application and database servers are run using Docker Container v4.15.

To answer **Research Question 1 (RQ 1)**, we used three fuzzing tools, namely EvoMaster v1.6.1, Restler v9.0.1, and RestTestGen v23.9. The fuzzing experiment will be performed using the black-box method. Each of the fuzzing tools will run using its default configuration. EvoMaster and Restler will run for 1 hour. Because RestTestGen does not have a time-based fuzzing feature, we run RestTestGen 5 times to get consistent results. Next, the number of unique errors marked by application response with status code 500 (Internal Server Error) discovered by each fuzzing tool will be measured.

To answer **RQ 2** and **RQ 3**, we use Joi v17.9.2 and Zod v3.20 libraries for the JavaScript/NodeJs-based application. In contrast, we use Marshmallow v3.20.1 and Pydantic v2.3 libraries for the Python-based application. Fuzzing will be performed again on the four input validation libraries in partial and full validation mode. After the fuzzing process is complete, the number of unique errors (status code 500) found by each fuzzing tool for each input validation library will be measured again. Then, to measure the effectiveness of each input validation library, we measure the percentage of error reduction in the application after the application is implemented with the validation mechanism.

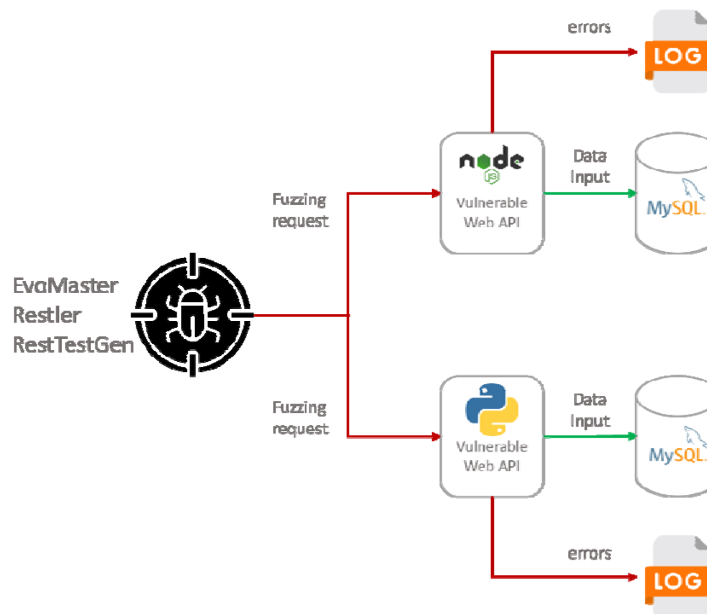

Fig 9. Fuzzing Scenario in Applications without Validation Mechanism.

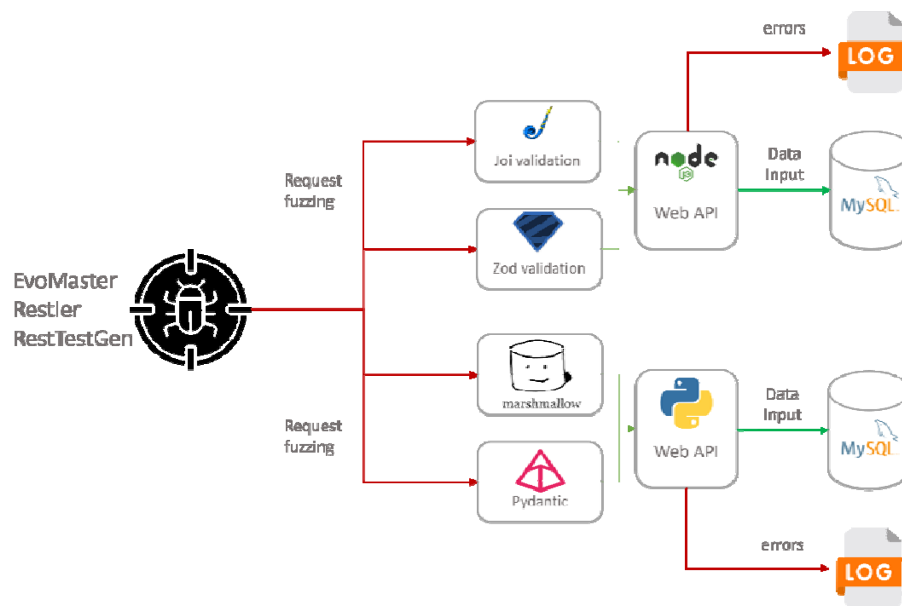

Fig 10. Fuzzing Scenario in Applications with Validation Mechanism.

Figures 9 And 10 show fuzzing scenarios that are performed in this research. Figure 9 shows that the fuzzing experiment is performed on applications configured not to have a validation mechanism. The application is connected to a database server, which stores client data input. We use EvoMaster, Restler, and RestTestGen to send fuzzing requests to the application. To measure the effectiveness of each fuzzing tool, we measure the number of unique errors obtained from the error log file generated by the application. The error log contains information about endpoint path, response status code, and error message. Figure 10 shows the fuzzing experiment on applications configured to have a validation mechanism using Joi and Zod libraries for the NodeJs-based application and Marshmallow and Pydantic libraries for the Python-based application. After the fuzzing process is complete, the effectiveness of the fuzzing tools is measured again through the number of unique errors detected by each fuzzing tool for each input validation library. Apart from the effectiveness of the fuzzing tools, we also measure the effectiveness of each input validation library in preventing fuzzing payloads.

3.5. *Performance Metrics*

To measure the effectiveness of the fuzzing tools, we use performance metrics in the form of the number of unique errors found by each fuzzing tool. To measure the effectiveness of the input validation libraries, we use performance metrics in the form of the percentage of error reduction in the application after the application is implemented with the validation mechanism. The percentage of error reduction is measured for each validation mode (partial and full).

To get information about the number of unique errors, we do not use the fuzzing report generated by each fuzzing tool because each tool has a different format and calculation of the number of errors. For example, EvoMaster only counts the number of endpoints with an error, while Restler counts the number of errors based on the unique fuzzing payload. On the other hand, RestTestGen counts the number of all requests that generate errors without considering the uniqueness of the fuzzing payload. Because these three fuzzing tools generate different information, we use another approach to count the number of unique errors to get accurate and objective information. We implemented a log system in the application to record every unexpected error.

If one type of error occurs on two different endpoints, it will be counted as two errors. For example, if the endpoints "/product" and "/product-tag-category" have the same error message, "Out of range value for column 'regular_price' at row 1", then it will be counted as two errors in the application. We also clean error messages that contain input parameter information. If two error messages have the same context but contain different input parameters, they will be counted as one error. For example, suppose on the endpoint "/product-tag-category" there are error messages "Incorrect datetime value: 'xxxxxx' for column 'start_date' at row 1" and "Incorrect datetime value: 'yyyyyy' for column 'start_date' at row 1". In that case, they will be counted as one error in the application.

To calculate the percentage of error reduction in the application after the application is implemented with validation mechanisms, we combined the error findings from all fuzzing tools and then cleaned the combined errors to get unique errors. The error reduction percentage is calculated for each input validation library in partial and full validation modes. We compare the number of errors in the application that does not have the validation mechanism to the number of errors in the application that has the validation mechanism.

As a note, in full validation mode, we intentionally did not implement validation in specific fields, namely regular_price (Product), max_usage (Coupon), and charge (Shipping). So technically, in the experiment performed in this research, we would not find an input validation library with an error reduction percentage of 100%. For example, the Coupon entity has a max_usage field with an integer data type. In full validation mode, we only implement validation for minimum value constraint, which is greater than or equal to 0, and do not implement validation for maximum value constraint. If the client sends max_usage data that exceeds the maximum constraint of the MySQL integer field (4294967295), it will trigger an error in the application. This aims to determine whether the fuzzing tools can trigger these remaining errors.

The performance metrics used in this research were calculated using Python scripts. The collection of Python scripts we use can be accessed via the GitHub repository [b].

4. Result and Discussions

4.1. *Fuzzing Tools Performance*

4.1.1. Fuzzing Tools Performance in Applications without Validation Mechanisms

Figure 11 shows the number of unique errors EvoMaster, Restler, and RestTestGen found in Web API applications without validation mechanisms. Because the applications do not have validation mechanisms, the error types that can be found in these applications are errors due to the absence of required input fields (missing required), errors due to data type that does not match the specification (invalid type), and errors due to violation of constraint and business logic (constraint violation).

After the fuzzing process for 1 hour, EvoMaster succeeded in sending fuzzing requests to all 7 application endpoints; in other words, EvoMaster had an endpoint coverage of 100%. In contrast, Restler only succeeded in sending fuzzing requests to 5 of 7 endpoints in the application (71.43% endpoint coverage). For RestTestGen, which does not have a time-based fuzzing feature, RestTestGen also succeeded in achieving an endpoint coverage test of 100%. Even though Restler has the lowest endpoint coverage, Restler managed to find the highest number of unique errors compared to other fuzzing tools. Restler found 140 errors in the NodeJs-based application and 164 errors in the Python-based application. The second position is achieved by RestTestGen, which found 115 errors in the NodeJs application and 108 errors in the Python application. EvoMaster occupies the last position in finding unique errors, namely 70 errors in the NodeJs and Python applications.

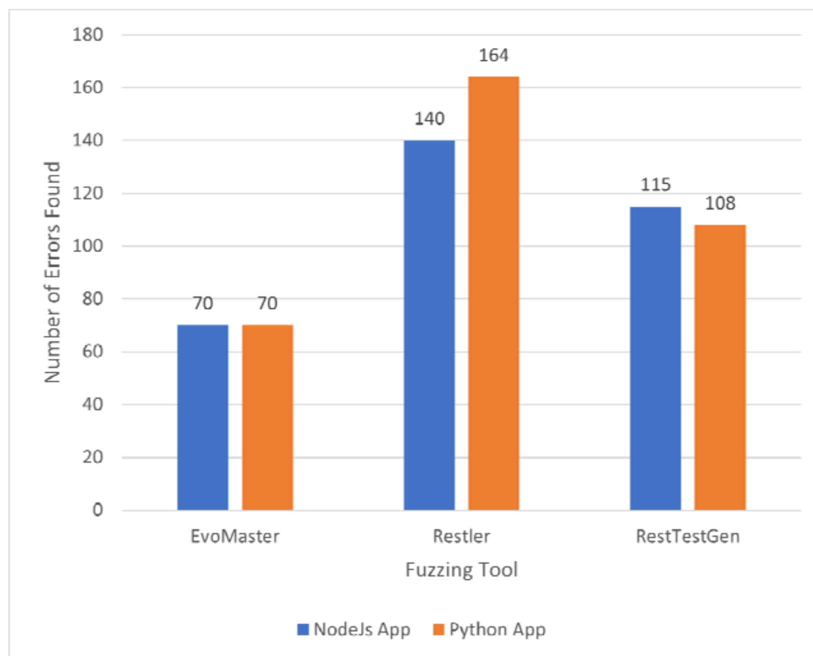

Fig 11. Comparison of the Number of Errors in Applications without Validation found by the Fuzzing Tools.

From the error log file analysis, most errors found by EvoMaster were constraint violation errors, such as the value of the quantity and discount_price fields in the Product entity, which exceeded the maximum value. EvoMaster does not exploit the missing required and invalid type errors, even though these two error types should be easy to find in an application that does not have a validation mechanism. EvoMaster focuses on

[b] https://github.com/bungdanar/data-processing

finding errors in all endpoints as quickly as possible to achieve 100% endpoint coverage in less time. When it successfully finds an error on an endpoint, EvoMaster does not perform further exploitation to find other error types and immediately moves to another endpoint. This behavior is in line with the report generated by EvoMaster, which only shows information on the number of endpoints that have errors.

RestTestGen also achieved 100% endpoint coverage with more errors found than EvoMaster. The errors found by RestTestGen are errors due to missing required, invalid type, and constraint violation. Hence, the number of unique errors found is higher than EvoMaster, which only focuses on constraint violation errors. In contrast to EvoMaster and RestTestGen, which managed to achieve 100% endpoint coverage, Restler only achieved endpoint coverage of 71.43% or 5 of 7 endpoints in the application. Even though it has the lowest endpoint coverage, Restler found the highest number of unique errors among other fuzzing tools. This finding indicates that Restler performed in-depth specification analysis to create various fuzzing payloads that could trigger errors in the application. Restler's fuzzing process, which focuses on finding as many errors as possible on an endpoint, results in low endpoint coverage for a certain period. In the context of the experiment performed in this research, we are confident that Restler can achieve 100% endpoint coverage if the fuzzing duration is increased to 1.5 or 2 hours.

4.1.2. Fuzzing Tools Performance in Applications with Partial Validation Mechanisms

Figure 12 shows the number of unique errors EvoMaster, Restler, and RestTestGen found in Web API applications with partial validation mechanisms. Because the applications have partial validation mechanisms, the only error type found in these applications is errors caused by a violation of constraint and business logic (constraint violation). There are no other errors caused by missing required fields or errors caused by data types that do not match specifications (invalid type).

After the fuzzing process for 1 hour, EvoMaster succeeded in achieving 100% endpoint coverage or, in other words, successfully sent the fuzzing requests to all seven endpoints in the application. RestTestGen also successfully achieved 100% endpoint coverage. Unlike EvoMaster and RestTestGen, Restler only achieved endpoint coverage of 71.43% or 5 out of 7 endpoints. Even though Restler has the lowest endpoint coverage, Restler managed to find the highest number of unique errors compared to other fuzzing tools. From Figure 12, it can be seen that Restler found 72 errors in the application with Joi and Pydantic, 74 in the application with Zod, and 71 in the application with Marshmallow. The second position is achieved by EvoMaster, which found the same number of errors in all applications (Joi, Zod, Marshmallow, and Pydantic), namely 68 errors. RestTestGen was in the last position in finding errors, namely 20 errors in the application with Joi, 29 in the application with Zod, 28 in the application with Marshmallow, and 34 in the application with Pydantic.

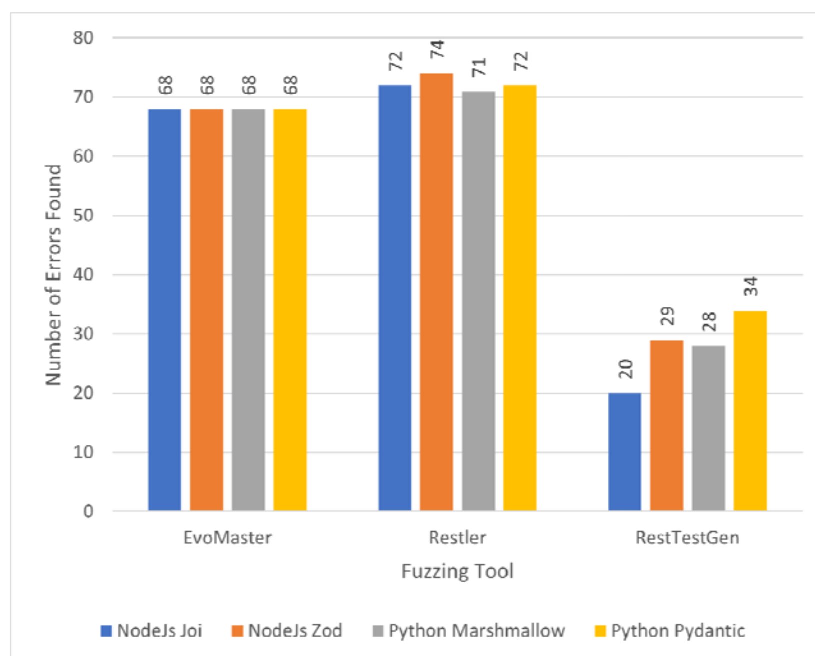

Fig 12. Comparison of the Number of Errors in Applications with Partial Validation found by the Fuzzing Tools.

From the error log file analysis, EvoMaster is quite good at exploiting constraint violation errors in all application endpoints. On endpoints that have complex input specifications, such as the endpoints "/product-tag-category-coupon" and "/user-address-product-shipping", EvoMaster performs tests on all entities or object components in the data input structure. For example, the endpoint "/user-address-product-shipping" has an input

specification consisting of four entities: User, Address, Product, and Shipping. In this case, EvoMaster tests several fields in these entities so that it can trigger more errors in the application.

On the other hand, Restler, which only achieved endpoint coverage of 71.43%, found a higher number of unique errors than EvoMaster. Like EvoMaster, Restler is good at exploiting constraint violation errors in application endpoints. The difference is that Restler tries to test all fields in the data input structure to detect more errors, even though it impacts a lower endpoint coverage. This differs from how EvoMaster works, which tends to find errors as quickly as possible and achieve a high level of endpoint coverage. When it successfully finds an error on an endpoint, EvoMaster immediately moves to test another endpoint and does not exploit further errors on the previous endpoint. This results in fewer errors than Restler, although the positive side is that it can achieve 100% endpoint coverage. Regarding Restler's endpoint coverage level, which is the lowest among other fuzzing tools, we are confident that if the fuzzing duration is increased to 1.5 or 2 hours, Restler will be able to achieve 100% endpoint coverage and be able to detect more errors.

Apart from EvoMaster, RestTestGen also achieved 100% endpoint coverage. Regarding fuzzing in the application with partial validation mechanisms, RestTestGen is in the last position in finding the number of errors. From the error log file analysis, it is known that RestTestGen is not good at exploiting errors on an endpoint. For example, on the endpoint "/user-address-product-shipping", which has a data input structure consisting of 4 entities (User, Address, Product, and Shipping), RestTestGen does not test the input fields for all these entities or, in this example, only test the input fields in the User and Address entities. Therefore, the number of errors detected by RestTestGen is less than that detected by EvoMaster and Restler, where EvoMaster and Restler are better at exploiting errors on an endpoint.

4.1.3. Fuzzing Tools Performance in Applications with Full Validation Mechanisms

Figure 13 shows the number of unique errors EvoMaster, Restler, and RestTestGen found in Web API applications with full validation mechanisms. Because the applications have full validation mechanisms, technically, no more errors will be found, whether errors caused by missing required fields, errors caused by inappropriate data types (invalid type), or errors caused by violation of constraints and business logic (constraint violation). However, as previously mentioned in the performance metrics subsection, we intentionally did not implement full validation for the maximum value constraint in the regular_price (Product), max_usage (Coupon), and charge (Shipping) fields. Therefore, in fuzzing experiments on applications with full validation mechanisms, constraint violation errors can still be found at these 5 endpoints "/product", "/product-tag-category", "/product-tag-category-coupon", "/ user-address-product", and "/user-address-product-shipping".

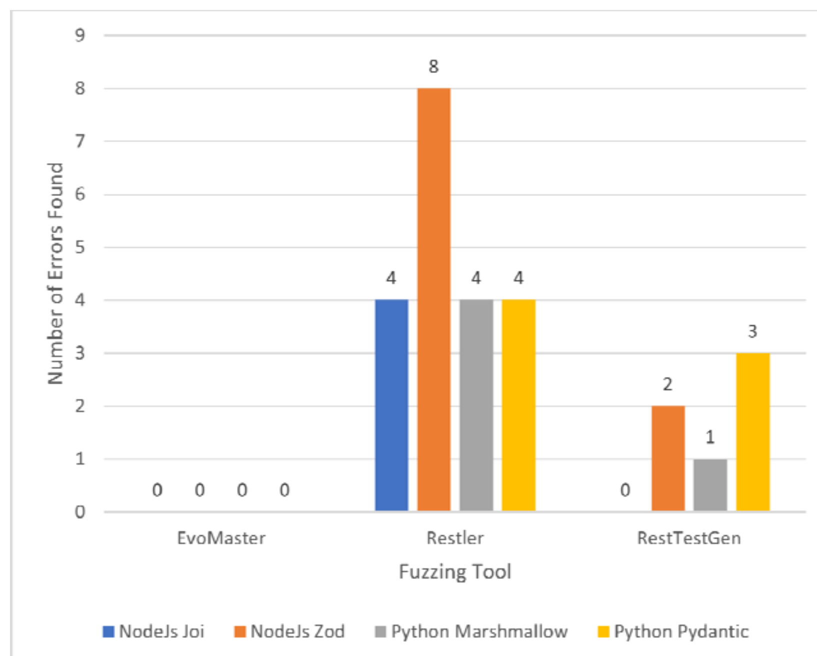

Fig 13. Comparison of the Number of Errors in Applications with Full Validation found by the Fuzzing Tools.

After the fuzzing process for 1 hour, EvoMaster successfully sent the fuzzing request to all seven endpoints in the application (100% endpoint coverage). RestTestGen also successfully achieved 100% endpoint coverage. Unlike EvoMaster and RestTestGen, Restler only achieved endpoint coverage of 71.43% or 5 out of 7 endpoints. Even though Restler has the lowest endpoint coverage, Restler managed to find the highest number of unique errors compared to other fuzzing tools. From Figure 13, it can be seen that Restler found 8 errors in

the application with Zod and 4 in the application with Joi, Marshmallow, and Pydantic. The second position is achieved by RestTestGen, which found 2 errors in the application with Zod, 1 in the application with Marshmallow, and 3 in the application with Pydantic. In contrast, in the application with Joi, RestTestGen failed to find any error. The last position is occupied by EvoMaster, which failed to find a single error in all applications.

From the analysis of error log files, it is known that Restler is very good at exploiting errors in the regular_price field (Product) and max_usage field (Coupon) in several application endpoints such as the "/product", "/product-tag-category", and "/ product-tag-category-coupon" endpoints. Restler has not successfully triggered an error for the charge field (Shipping) in the "/user-address-product-shipping" endpoint. This is caused by Restler's limited fuzzing duration, so Restler has not had enough time to test all endpoints in the application. We are confident that if Restler's fuzzing duration is increased to 1.5 or 2 hours, Restler will be able to achieve 100% endpoint coverage and detect more errors in all application endpoints.

EvoMaster, which has 100% endpoint coverage, failed to find a single error. As previously explained, EvoMaster tends to try to find errors as quickly as possible and achieve a high level of endpoint coverage. When it does not find an error on an endpoint, EvoMaster immediately moves to test another endpoint and does not exploit further errors on the previous endpoint. The lack of error exploitation performed by EvoMaster resulted in no error being discovered even though the endpoint coverage had reached 100%. This is different from the way Restler works, which tries to exploit further errors on an endpoint so that it succeeds in finding remaining errors even though it impacts a lower level of endpoint coverage. Apart from EvoMaster, RestTestGen also achieved 100% endpoint coverage. Regarding fuzzing in the application with full validation mechanisms, RestTestGen only succeeded in finding errors in the application with Zod, Marshmallow, and Pydantic. From the analysis of the error log file, it is known that RestTestGen succeeded in exploiting an error in the regular_price field (Product) in the endpoints "/product", "/product-tag-category", and "/product-tag-category-coupon". RestTestGen has yet to succeed in finding errors in the max_usage field (Coupon) and charge field (Shipping), even though it has reached 100% endpoint coverage.

4.2. *Input Validation Libraries Performance*

Figure 14 shows the number of errors in the NodeJs application when receiving fuzzing requests from fuzzing tools. Figure 14 compares the number of errors between applications without validation mechanisms and those with validation mechanisms using Joi and Zod libraries. From Figure 14, it can be seen that there is a decrease in the number of errors in the application after the application has validation mechanisms. The number of errors shown in Figure 14 is the aggregate number found by EvoMaster, Restler, and RestTestGen, which are then filtered to obtain unique errors. Before having validation mechanisms, there were 241 errors in the application. After the partial validation mechanism was implemented in the application, the number of errors decreased to 118 (51.04% decrease) in the application with Joi and 121 (49.79% decrease) in the application with Zod.

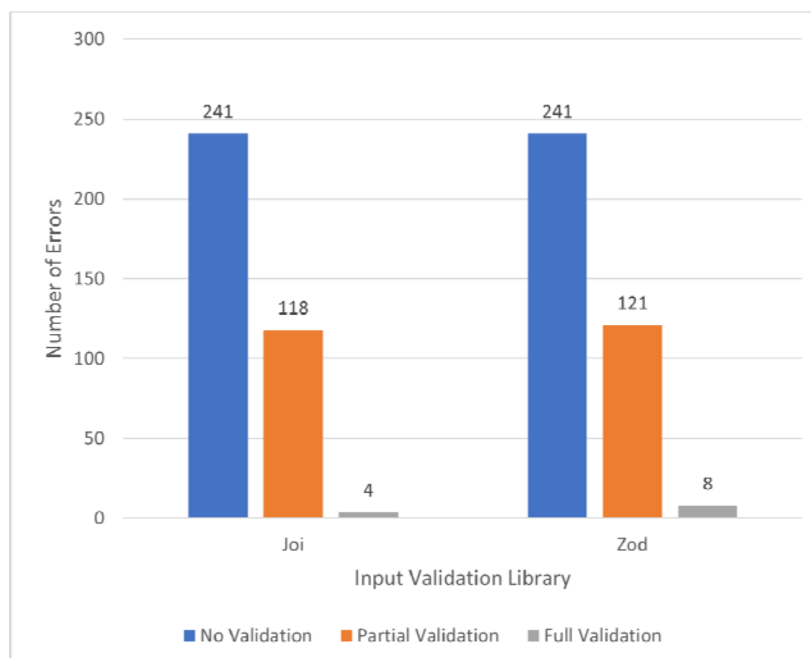

Fig 14. Comparison of Error Reduction in NodeJs Application Before and After Validation Implementation.

From the analysis of the error log files, errors that still occur in the application with partial validation mechanisms are errors caused by violation of constraint and business logic in data input (constraint violation), such as the weight field in the Product entity which receives input that exceeds its maximum limit. Technically, there should be no more errors caused by invalid data types in partial validation mode. However, based on the error log file analysis, invalid type errors were still found in the application with Zod.

The invalid type errors found are related to the start_date and end_date fields in the Coupun entity, where both fields have the datetime data type in the MySQL database. Regarding the datetime field, we apply ISO 8601 format validation for every data input related to datetime, where one of the specifications is that the year value must be in the range of 0 - 9999. On Joi, ISO 8601 format validation can be done using the "Joi.date().iso()" method, while on Zod, it can be done using the "z.string().datetime()" method. For example, the fuzzing tool sends input with the value "202364-01-01 00:00:00" for the start_date field in the Coupon entity. This value is considered valid by Zod and is passed to the database, which ultimately causes an error.

After the application had full validation mechanisms, the number of errors decreased to 4 errors (98.34% decrease from 241) in the application with Joi and 8 errors (96.68% decrease from 241) in the application with Zod. As previously mentioned in the Performance Metrics section, in full validation mode, we intentionally did not implement full validation in several input fields so that technically, in the experiment performed in this research, we would not find an input validation library that had an error reduction rate of up to 100%.

The remaining errors still exist in the application with Joi are errors in the regular_price field (Product) and the max_usage field (Coupon), for which we did not implement validation for the maximum value. In the application with Zod, the remaining errors were also related to the regular_price and max_usage fields, plus an invalid type error related to the datetime field. Therefore, for the fuzzing scenario in the application with full validation mechanisms, the number of errors in the application with Zod is more than in the application with Joi, as seen in Figure 14. This residual error can be easily eliminated by adding validation for the minimum or maximum value constraints. For example, for the regular_price field, which has the data type decimal(19, 4) in the MySQL database, we can add Joi and Zod validation using the following method:

(1) Joi: Joi.number().precision(4).min(0).less(1e15).required()
(2) Zod: z.coerce.number().nonnegative().lt(1e15)

Figure 15 shows the number of errors in the Python application when receiving fuzzing requests from fuzzing tools. Figure 15 compares the number of errors between applications without validation mechanisms and applications with validation mechanisms using Marshmallow and Pydantic libraries. From Figure 15, it can be seen that there is a decrease in the number of errors in the application with validation mechanisms. The number of errors shown in Figure 15 is the aggregate number found by EvoMaster, Restler, and RestTestGen, which are then filtered to obtain unique errors. In applications without validation mechanisms, 255 unique errors were successfully detected by the fuzzing tools. After implementing the partial validation mechanism in the application, the number of errors decreased by 51.76%, or from 255 errors to 123 errors, in the application with Marshmallow, and decreased by 53.33%, or from 255 errors to 119 errors in the application with Pydantic.

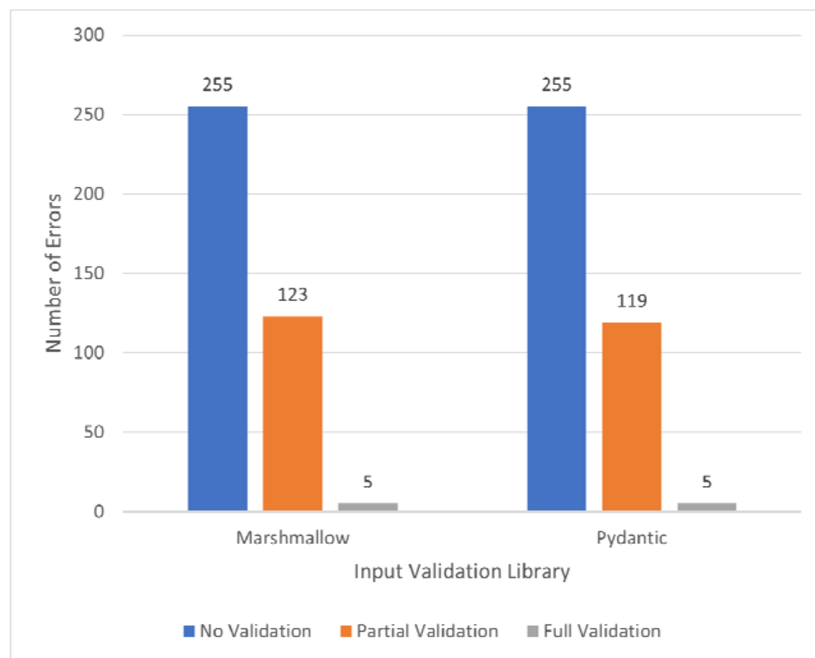

Fig 15. Comparison of Error Reduction in Python Application Before and After Validation Implementation.

From the error log file analysis, it is known that errors that still occur in the application with partial validation mechanism are errors caused by violations of constraint and business logic in data input (constraint violation), such as the max_days field in the Shipping entity which receives input that exceeds its maximum limit and the email field in the User entity that receives input in the form of a string that does not match valid email pattern. In the case of partial validation using Marshmallow or Pydantic, no errors were found due to invalid data type.

After the application had full validation mechanisms, the number of errors decreased to 5 errors (98.04% decrease from 255) in the application with Marshmallow and Pydantic. The remaining errors are errors in the regular_price field (Product) and the max_usage field (Coupon), for which we did not implement validation for the maximum value constraint. This result shows that Marshmallow and Pydantic have the same level of effectiveness in performing full validation on input provided by the client. This residual error can be easily eliminated by adding validation for the minimum or maximum value constraints. For example, for the regular_price field, which has the data type decimal(19, 4) in the MySQL database, we can add Marshmallow and Pydantic validation using the following method:

(1) Marshmallow: fields.Decimal(required=True, places=4, validate=validate.Range(min=0, max=999999999999999.9999))
(2) Pydantic: Field(ge=0, max_digits=19, decimal_places=4)

5. Conclusion and Future Work

In this research, we compared the effectiveness of several state-of-the-art fuzzing tools, namely EvoMaster, Restler, and RestTestGen. We also combine fuzzing experiments with several state-of-the-art input validation libraries: Joi, Zod, Marshmallow, and Pydantic. The fuzzing experiment was performed on Web API applications that we developed using NodeJs and Python. The Web API application can be run using three modes: without validation, partial validation, and full validation. The performance metrics used in this research are the effectiveness of the fuzzing tool, which is measured by calculating the number of errors that are found, and the effectiveness of the input validation library, which is measured by calculating the percentage reduction in the number of errors before and after the application has validation mechanisms.

In the fuzzing experiment on applications without validation, Restler found the highest average number of errors with 152 errors, followed by RestTestGen with 111 errors and EvoMaster with 70 errors. Then, Restler found the highest average number of errors in the fuzzing experiment on applications with partial validation, namely 72 errors, followed by EvoMaster with 68 errors and RestTestGen with 27 errors. In the fuzzing experiment on applications with full validation, Restler found the highest average number of errors with 5 errors, followed by RestTestGen with 1 error and EvoMaster with 0 error.

We also evaluated the effectiveness of each input validation library in reducing errors. In NodeJs applications with partial validation, Joi and Zod reduced errors by 51.04% and 49.79%, respectively. Furthermore, Joi and Zod reduced errors by 98.34% and 96.68%, respectively, in applications with full validation. In Python applications with partial validation, Marshmallow and Pydantic reduced errors by 51.76% and 53.33%, respectively. Furthermore, Marshmallow and Pydantic reduced errors by 98.04% in applications with full validation.

For future work, we plan to conduct a fuzzing experiment using other input validation libraries, such as Fluent Validation for the C# programming language and Validator for the Go programming language.

Acknowledgments

This work was fully funded by the Ministry of Communication and Information Technology, Indonesia – Domestic Masters Scholarship Program.

Conflict of Interest

The authors have no conflicts of interest to declare in this study.

References

[1] A01 Broken Access Control - OWASP Top 10:2021 (no date). https://owasp.org/Top10/A01_2021-Broken_Access_Control/ (Accessed: 26 July 2023).
[2] A03 Injection - OWASP Top 10:2021 (no date). https://owasp.org/Top10/A03_2021-Injection/ (Accessed: 26 July 2023).
[3] Arcuri, A., 2019. RESTful API automated test case generation with EvoMaster. ACM Transactions on Software Engineering and Methodology (TOSEM), 28(1), pp.1-37.
[4] Assal, H. and Chiasson, S., 2019, May. 'Think secure from the beginning' A Survey with Software Developers. In Proceedings of the 2019 CHI conference on human factors in computing systems (pp. 1-13).
[5] Atlidakis, V., Godefroid, P. and Polishchuk, M., 2019, May. Restler: Stateful rest api fuzzing. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) (pp. 748-758). IEEE.
[6] Atlidakis, V., Godefroid, P. and Polishchuk, M., 2020, October. Checking security properties of cloud service REST APIs. In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST) (pp. 387-397). IEEE.
[7] Beaman, C., Redbourne, M., Mummery, J.D. and Hakak, S., 2022. Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. Computers & Security, 120, p.102813.

[8] Bello, S.A., Oyedele, L.O., Akinade, O.O., Bilal, M., Delgado, J.M.D., Akanbi, L.A., Ajayi, A.O. and Owolabi, H.A., 2021. Cloud computing in construction industry: Use cases, benefits and challenges. Automation in Construction, 122, p.103441.
[9] CVE - CVE-2021-21972 (no date). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-21972 (Accessed: 26 July 2023).
[10] Cyber Security Hub (2023) IOTW: Everything we know about the Optus data breach. https://www.cshub.com/attacks/news/iotw-everything-we-know-about-the-optus-data-breach (Accessed: 26 July 2023).
[11] Ehsan, A., Abuhaliqa, M.A.M., Catal, C. and Mishra, D., 2022. RESTful API testing methodologies: Rationale, challenges, and solution directions. Applied Sciences, 12(9), p.4369.
[12] EMResearch/EvoMaster (no date) GitHub - EMResearch/EvoMaster: The first open-source AI-driven tool for automatically generating system-level test cases (also known as fuzzing) for web/enterprise applications. Currently targeting whitebox and blackbox testing of Web APIs, like REST, GraphQL and RPC (e.g., gRPC and Thrift). https://github.com/EMResearch/EvoMaster (Accessed: 26 July 2023).
[13] Godefroid, P., 2020. Fuzzing: Hack, art, and science. Communications of the ACM, 63(2), pp.70-76.
[14] Godefroid, P., Huang, B.Y. and Polishchuk, M., 2020, November. 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 (pp. 725-736).
[15] Higginbotham, J., 2021. Principles of Web API Design: Delivering Value with APIs and Microservices. Addison-Wesley Professional.
[16] Humayun, M., Jhanjhi, N., Almufareh, M.F. and Khalil, M.I., 2022. Security threat and vulnerability assessment and measurement in secure software development. Comput. Mater. Contin, 71, pp.5039-5059.
[17] Hussain, F., Hussain, R., Noye, B. and Sharieh, S., 2020. Enterprise API security and GDPR compliance: Design and implementation perspective. IT Professional, 22(5), pp.81-89.
[18] Joi.dev (no date). https://joi.dev/ (Accessed: 26 July 2023).
[19] Kim, M., Xin, Q., Sinha, S. and Orso, A., 2022, July. 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 (pp. 289-301).
[20] Klyuchnikov, M. (2021) Unauthorized RCE in VMware VCenter. https://swarm.ptsecurity.com/unauth-rce-vmware/ (Accessed: 26 July 2023).
[21] Lei, Z., Chen, Y., Yang, Y., Xia, M., and Qi, Z. 2023. Bootstrapping Automated Testing for RESTful Web Services. IEEE Transactions on Software Engineering, 49(4), p.1561-1579.
[22] marshmallow (no date) simplified object serialization — marshmallow 3.21.0 documentation. https://marshmallow.readthedocs.io/en/stable/ (Accessed: 26 July 2023).
[23] Mateus-Coelho, N., Cruz-Cunha, M. and Ferreira, L.G., 2021. Security in microservices architectures. Procedia Computer Science, 181, pp.1225-1236.
[24] microsoft/restler-fuzzer (no date) GitHub - microsoft/restler-fuzzer: RESTler is the first stateful REST API fuzzing tool for automatically testing cloud services through their REST APIs and finding security and reliability bugs in these services. https://github.com/microsoft/restler-fuzzer (Accessed: 26 July 2023).
[25] Pydantic (no date) Welcome to Pydantic. https://docs.pydantic.dev/latest/ (Accessed: 26 July 2023).
[26] SeUniVr/RestTestGen (no date) GitHub - SeUniVr/RestTestGen: A framework for automated black-box testing of RESTful APIs. https://github.com/SeUniVr/RestTestGen (Accessed: 26 July 2023).
[27] Subramanian, H. and Raj, P., 2019. Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs. Packt Publishing Ltd.
[28] Tabrizchi, H. and Kuchaki Rafsanjani, M., 2020. A survey on security challenges in cloud computing: issues, threats, and solutions. The journal of supercomputing, 76(12), pp.9493-9532.
[29] UpGuard (no date) How did the Optus data breach happen?. https://www.upguard.com/blog/how-did-the-optus-data-breach-happen (Accessed: 26 July 2023).
[30] Viglianisi, E., Dallago, M. and Ceccato, M., 2020, October. Resttestgen: automated black-box testing of restful apis. In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST) (pp. 142-152). IEEE.
[31] Zhang, M. and Arcuri, A., 2023. Open problems in fuzzing restful apis: A comparison of tools. ACM Transactions on Software Engineering and Methodology, 32(6), pp.1-45.
[32] zod.dev (No. date) TypeScript-first schema validation with static type inference. https://zod.dev/ (Accessed: 26 July 2023).

Authors Profile

Danar Gumilang Putera is a postgraduate student at the University of Indonesia. He is pursuing a master's degree from the Department of Electrical Engineering at the University of Indonesia. His current research interests include web application development and web application security.

Ruki Harwahyu received the B.E. degree in computer engineering from Universitas Indonesia (UI), Jakarta, Indonesia, in 2011, the M.E. degree in computer and electronic engineering from UI and the National Taiwan University of Science and Technology (NTUST), Taipei, Taiwan, in 2013, and the Ph.D. degree in electronic and computer engineering from NTUST in 2018. He is currently an Assistant Professor with the Department of Electrical Engineering, Faculty of Engineering, UI. He serves as an IT Adviser with the Faculty of Engineering, UI; a Lead System Developer with UI Greenmetric World University Ranking; and a member of the Learning Technology Enhancement Team, Faculty of Engineering, UI. His current research interests include computer and communication networks and Internet of Things.