FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Efficient fuzz testing of web services
Student:	Bc. Matúš Ferech
Supervisor:	prof. Ing. Pavel Tvrdík, CSc.
Study program:	Informatics
Branch / specialization:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

The goal of the thesis is to provide to developers and penetration testers a fuzzer tool that enables to do fuzz testing of web services via their API efficiently. By efficiency of a fuzzer we understand its ability to provide comprehensive easy-to-understand outputs of the fuzz testing. Said otherwise, using outputs produced by a fuzzer a user can quickly and efficiently find the source of the web service misbehavior. An efficient fuzzer generates and minimizes the payload triggering bugs. This feature is essential when reproducing and troubleshooting bugs of web services. Additionally, an efficient fuzzer should provide a basic profiling of the API that can be subsequently used to discover endpoints susceptible to denial-of-service attacks.

Perform a state-of-the-art of current approaches to fuzz testing of web services via their APIs and analyze shortcomings of existing open source projects. Study methods for reduction or minimization of data outputs of fuzz testing in general and perform a survey of open source tools or libraries supporting it. Starting from your own version of fuzzer developed within your bachelor thesis [1], design the architecture of a new version of this fuzzer that satisfies the requirements for efficient fuzzing defined above. Implement the designed system as an open-source tool. Perform extensive testing of your new fuzzer on a representative collection of established web services. Using results of these tests, evaluate efficiency of your tool and its ability to discover endpoints susceptible to denial-of-service attacks.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 21 February 2023 in Prague.

[1] M.Ferech: Application-level fuzzing, Bachelor Thesis, Comenius University Bratislava, 2019. https://github.com/matusf/bachelor-thesis/releases/download/1.0.0/thesis.pdf

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 21 February 2023 in Prague.

Master's thesis

EFFICIENT FUZZ TESTING OF WEB SERVICES

Bc. Matúš Ferech

Faculty of Information Technology
Department of Computer Systems
Supervisor: prof. Ing. Pavel Tvrdík, CSc.
May 2, 2023

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Ferech Matúš. *Efficient fuzz testing of web services*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of Code listings

First of all, I would like to thank my supervisor, prof. Ing. Pavel Tvrdík, CSc., for his academic insights and thorough reviews.

Then I would like to thank my family and friends for their support, patience, and encouragement.

Last but not least, I wanna thank me
I wanna thank me for believing in me
I wanna thank me for doing all this hard work
I wanna thank me for having no days off
I wanna thank me for... for never quitting
I wanna thank me for always being a giver
And tryna give more than I receive
I wanna thank me for tryna do more right than wrong
I wanna thank me for just being me at all times

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 2, 2023 .....

Abstract

This thesis proposes a novel approach to web service fuzzing that utilizes the OpenAPI Specification. The proposed smart black-box generation-based fuzzer, named openapi-fuzzer, **generates** and **minimizes** random payloads to detect vulnerabilities in web services. It is able to minimize the bug-triggering payload to its canonical form. Thanks to this minimization, it is trivial to detect the root cause of an underlying bug. To evaluate its performance, openapi-fuzzer was tested on multiple relevant web services, including Kubernetes, Hashicorp Vault, and Gitea, where it identified several bugs. The results demonstrate that openapi-fuzzer outperforms other state-of-the-art web service fuzzers in terms of the number of bugs found and running time.

Furthermore, openapi-fuzzer conducts a performance analysis to identify endpoints that are susceptible to Denial of Service attacks. By providing developers with detailed statistics, openapi-fuzzer helps them identify and fix performance issues in their web services.

Keywords fuzzing, OpenAPI Specification, testing, web services, property-based testing

Abstrakt

V tejto práci predstavíme inteligentný generatívny black-box fuzzer, ktorý využíva OpenAPI špecifikáciu na odhalenie zraniteľností webových služieb. Jedným z hlavných prínosov fuzzera je podpora minimalizácie vstupov, ktoré spôsobujú chyby v programoch. Vďaka minimalizácií je jednoduché nájsť chybu, ktorá zapríčinila problémové správanie programu. Fuzzer bol testovaný na niekoľkých relevantných webových službách, ako napríklad Kubernetes, Hashicorp Vault či Gitea, v ktorých našiel mnoho chýb. Z porovnania s inými najmodernejšími fuzzermi sme zistili, že openapi-fuzzer nájde viac chýb za kratší čas akoiné fuzzery.

Dalšou funkcionalitou openapi-fuzzera je analýza výkonu na detekciu endpointov náchylných na útoky DOS.

Kľúčové slová fuzzovanie, OpenAPI špecifikácia, testovanie, webové služby

Abbreviations

API	Application Programming Interface
BFS	Breadth-First Search
CRUD	Create, Read, Update and Delete
DFS	Depth-First Search
DOS	Denial Of Service
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
LOC	Lines Of Code
OAS	OpenAPI Specification
REST	REpresentation State Transfer
SOA	Service-Oriented Architecture
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
JSON	JavaScript Object Notation
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

Introduction

The increasing complexity of modern software systems and the ever-increasing number of connected devices and systems make it difficult to identify and mitigate software vulnerabilities. Fuzz testing is based on providing malformed or unexpected inputs to a program to detect software defects and vulnerabilities.

It is essential to create test cases that not only trigger bugs but also enable developers or security analysts to find the root cause of the bugs efficiently. This is particularly relevant for fuzz testing, which is the process of generating and sending semi-random inputs to the API of a given app. Thus, it may be challenging to understand the generated payload. By reducing to a minimal payload that causes the system to fail, the debugging and exploitation of any discovered vulnerabilities can be greatly improved and sped up. Therefore, generating minimal and efficient payloads is a critical feature of any effective fuzzer.

The primary aim of this thesis is to create a *smart* fuzzer that takes advantage of the OpenAPI Specification to generate a well-structured input and effectively detect bugs in web services. The performance of the fuzzer will be evaluated by testing it on multiple relevant web services. In addition to sending multiple requests to each endpoint, the fuzzer will also utilize this feature to measure the round-trip times of the requests. Through this evaluation, our goal is to identify endpoints that may be vulnerable to DOS attacks.

We provide an overview of fuzzing, particularly in the context of web services. We present an overview of the fuzzer taxonomy. We thoroughly analyze existing solutions, identify their limitations, and design and implement a fuzzer that overcomes these limitations.

Chapter 1

Fuzz testing

Fuzzing is a software testing methodology that consists of subjecting a system to a large number of inputs generated by a separate program. The goal is to identify potential security vulnerabilities, functional bugs, or other flaws in the tested software.

Miller characterized this form of testing as a random traversal of a program's state space, represented as a state machine, in search of undefined states [1].

Fuzzing is often used to test complex and distributed systems that may have numerous input sources and interactions, making traditional testing techniques less effective. By automating the input generation process, fuzzing allows for efficient and thorough testing of a wide range of input scenarios. Despite its simplicity, fuzzing has proven to be a powerful technique for detecting software vulnerabilities and improving software quality; when Miller and his research team were successful in destabilizing a significant part of Unix battle-tested utilities, ranging from 24 to 33% of all utilities, by implementing a random input generation technique [1].

1.1 Why fuzzing

Nonetheless, fuzz testing is just one of many methods that can be used to improve software security and reliability. Quality assurance can be achieved via multiple other methods, including unit testing, integration testing, code review, or performance testing. Fuzzing offers some advantages to all of these methods.

One of the main advantages of fuzz testing is its **thoroughness**. By generating a wide range of inputs, including edge cases and other inputs that may be missed by other testing approaches, fuzz testing can help uncover defects that might otherwise go undetected. It can identify potential security issues, such as buffer overflows or other input-related issues that attackers could exploit, and thus, it can improve **security**.

Another advantage of fuzz testing is its **automation**. Once the fuzzer has been set up, it can generate a large number of inputs without human intervention. This can save time and resources, allowing faster and more efficient testing. This also means that the testing can be repeated as often as needed. Automation of the testing process reduces the need for specialized knowledge or equipment, which can be **cost-efficient** [2].

Flexibility is an additional aspect that makes fuzz testing stand out. One fuzzer can test multiple software with minimal or even no modification. Some types of fuzzers push it to the limit and do not even require a source code.

1.2 Need for efficiency

Though, finding the security vulnerability, bug, or other faulty behavior is important, it is only half the way to fixing or exploiting it. Both hackers and developers need to understand why the bug occurred in order to make use of it. Nevertheless, data created by fuzzers are inherently noisy due to the randomness of their generation process. It is much easier to find the root cause of the faulty behavior if the input that triggers it is minimal. By minimal input, I mean that the size of the input (in bytes) is the smallest possible and follows the lexicographical order.

The example in Code listing 1 is an input generated by the old version of openapi-fuzzer [3] that causes a bug in Hashicorp Vault. The generated payloads vary in size from 500B to 10KB. JSON in Code listing 1 has numerous fields; however, without manual intervention, it is unclear which causes the bug. It may be a combination of fields or just a single one. Moreover, fields may have different values. There is no way to detect which subset of values causes the faulty behavior. The desired input would have all fields that do not contribute to causing the bug empty, and the ones that do would shrink to the shortest possible size. A sample of this input is given in Code listing 2.

```
{
  "payload": {
    "query_params": [],
    "path_params": [],
    "headers": [],
    "body": {
      "increment": -6585199501202910990,
      "lease_id": "%F1%AD%8B%9D%F0%96%BA%AD%E0%BA%8D%F3%9D%B8%B5%F1%99%96%AC
      %AC%83%99%F3%83%81%B9%F1%9E%9F%80%F1%B1%9B%AD%F1%AB%8F%86%F2%9A%.....",
      "url_lease_id": "%F1%88%93%9D%F1%AC%A2%AA%F1%9C%93%84%F3%88%AF%82%F3%
      8A%88%81%F0%B3%B8%A3%F2%A1%B8%96%F4%8C%B3%AF%F1%BF%BC%97%F1%89%....."
    }
  },
  "path": "sys/leases/renew",
  "method": "POST"
}
```

Code listing 1 Input generated by the old version of openapi-fuzzer(7.6KB in size)

1.3 Need for fuzzing web services

In recent years, web services have become increasingly complex and widely used, making them a prime target for cyber attackers. Many services that were offered as standalone applications are now available on the Web. The crucial difference is that everyone can interact with the

```
{
  "payload": {
    "query_params": [],
    "path_params": [],
    "headers": [],
    "body": {
      "increment": 0,
      "lease_id": ". ",
      "url_lease_id": ""
    }
  },
  "path": "sys/leases/renew",
  "method": "POST"
}
```

Code listing 2 Desired minimal input

web service, which was not the case with standalone desktop GUI applications. Additionally, by exploiting a web service, attackers can potentially access data of many users and put their credentials and data in jeopardy. As a result, the need for effective security testing techniques, such as fuzzing, has recently become more necessary.

Moreover, fuzzing can be used to test the interoperability of web services with other systems or software components. This is particularly important in service-oriented architectures (SOAs), where multiple web services may need to interact with each other. Fuzzing can help identify compatibility issues, protocol violations, or other communication problems that could affect the reliability and performance of web services.

1.3.1 HTTP essentials

HTTP, or Hypertext Transfer Protocol, is a protocol that governs most web communication. It works in a client-server model, where the client sends a request to the server, which then processes the request and sends back a response. HTTP requests typically consist of a request method, a URI indicating the resource to be accessed, and an optional message body containing additional data.

HTTP responses contain a status line with a version of the protocol, a status text, and a status code. The status text and status code express the situation of the requests along with the condition of the server. The status line is then followed by optional headers and a body [4].

HTTP status codes are particularly important for fuzzing. Fuzzers can employ them to determine the state of the web service. HTTP status codes fall into following five categories.

- Informational (1XX): These status codes indicate that the server has received the request and is continuing to process it. For example, 101 (Switching Protocols) indicates that the server received a request from a client to switch to the protocol specified in an Upgrade header and will start using it. For example, switching from HTTP to WebSockets.

- Successful (2XX): These status codes indicate that the request was received, understood, and

CRUD operation	HTTP method
CREATE	POST
READ	GET
UPDATE	PUT
DELETE	DELETE

Table 1.1 Mapping between CRUD operations and HTTP methods

accepted successfully. For example, 200 (OK) means that the request was successful and that the server is returning the requested data.

- Redirection (3XX): These status codes indicate that the client needs to take further action to complete the request. For example, 301 (Moved Permanently) means that the requested resource has been moved to a different location and the client should update its URL accordingly.
- Client error (4XX): These status codes indicate that there was an error with the client's request. For example, 404 (Not Found) means that the requested resource could not be found on the server.
- Server error (5XX): These status codes indicate that there was an error in the server processing the request. For example, 500 (Internal Server Error) means that there was an unexpected condition that prevented the server from fulfilling the request.

1.3.2 Representational state transfer

REST is a predominantly used set of architectural principles and constraints to design web services [5]. Web services that meet these conditions are said to be RESTful. The following are essential elements for fuzzing and understanding.

RESTful services should follow the **client-server model**. There should be a uniform interface that separates clients from servers. Servers are not concerned with the user interface or user state, so servers can be simpler and more scalable [6].

Another feature that helps with scaling RESTful applications is **statelessness**. In stateless applications, the server should not store any client context between requests. This principle makes **caching** and thus scaling easier [6].

The last one is **universal interface**, which Alex Rodriguez described and explicitly defined in the article RESTful Web services: The basics [5].

- Use HTTP methods to describe CRUD operations on server resources. There exists a one-to-one mapping from HTTP methods to CRUD operations, also shown in Table 1.1. A non-RESTful application may, for example, use the HTTP GET method and send data as query parameters to update a resource. A RESTful application would properly use an HTTP body with a PUT method.
- Expose directory-structure-like URIs. Directory-structure-like URIs do not expose the underlying technology through file extensions of scripting languages such as `.php` or `.asp`. This

fact makes migration smooth. Moreover, directory-structure-like URIs have a natural self-documenting feature and describe the resources that servers provide. For instance, to list repositories of some organization via the GitHub API, one would send an HTTP GET request to the URL ending with `/orgs/{org}/repos`.

- Use JSON or XML as a serialization format. JavaScript is a widely used language for front-end web development. Its object notation has a simple specification, and therefore most programming languages have a library for its serialization and deserialization.

Web services that adhere to the aforementioned requirements can be straightforwardly described by the OAS.

1.3.3 OpenAPI Specification

The OpenAPI Specification (briefly OAS), formerly known as the Swagger Specification, is an open standard for describing RESTful APIs. It is specified in YAML or JSON, which are **machine-readable**. It can also be machine generated, which makes it more and more popular.

By describing the structure and behavior of an API in a machine-readable format, the OAS enables tools to automatically generate client and server libraries, interactive documentation, and other artifacts that can simplify the process of consuming and working with the API. An example of interactive documentation is shown in Figure 1.1.

The combination of a machine-readable format and a detailed description of input and output structures makes the OAS a vital resource for creating a *smart* fuzzer.

1.3.3.1 Fundamental parts of OAS for fuzzers

In the following paragraphs, I will use parts of the OAS from one of the official examples [7].

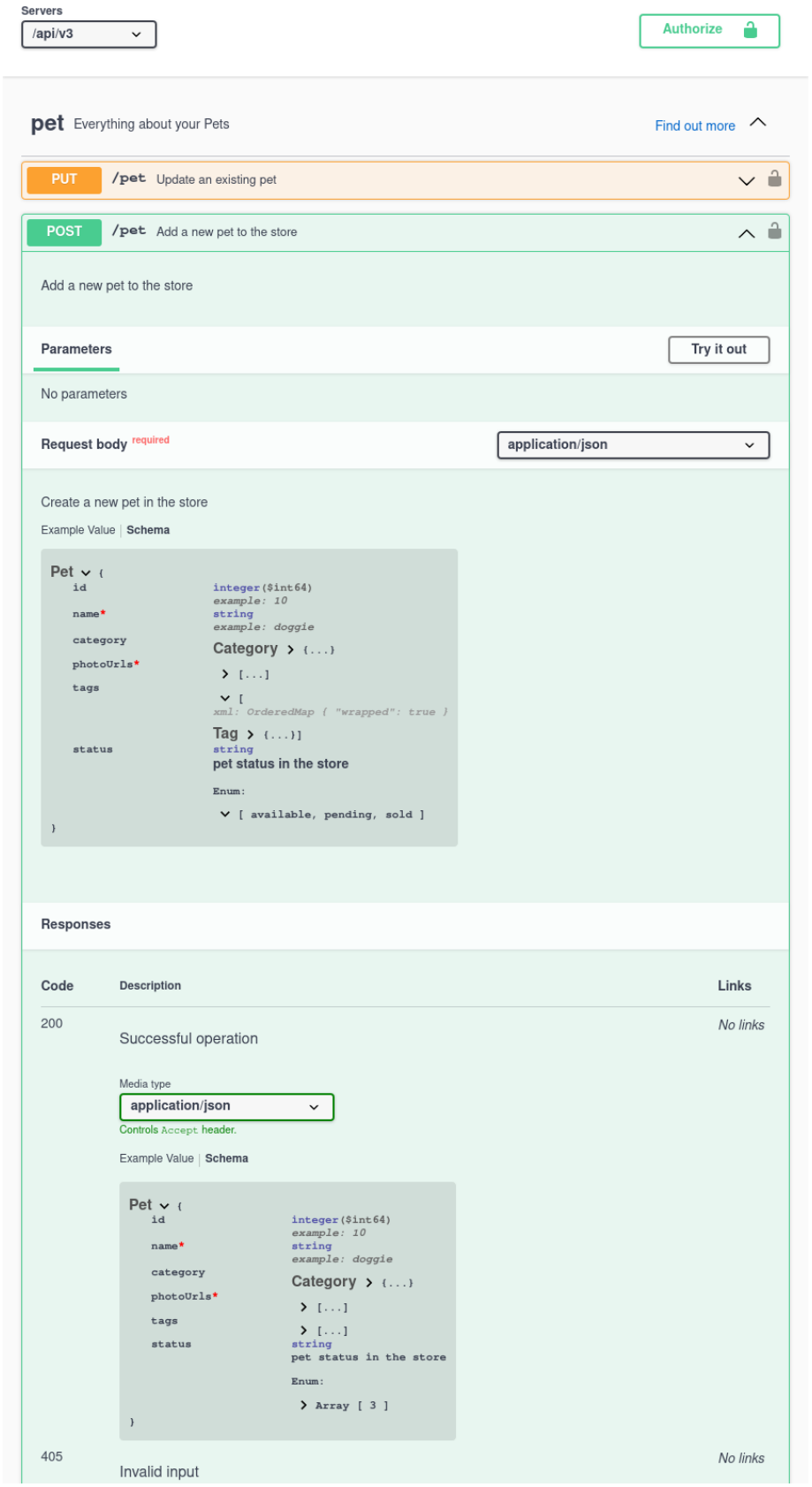
One of the top-level elements is the Servers Object, that defines connectivity information to a target server; see example in Code listing 3. It has a required `url` field, which can also be relative [7]. However, the Servers Object itself is not required. This makes it rather difficult to process programmatically.

```yaml
servers:
  - url: https://petstore3.swagger.io/api/v3
  - url: /api/v3
```

Code listing 3 An example of a Servers Object

One of the components that possess most of the useful information is the Path Object. It consists of Path Item Objects which are the relative paths to the individual path endpoints and their operations [7]. The Path Item Objects are appended to the URL from the Server Object to construct the full URL.

The primary item in a Path Item Object is the Operation Object. There are multiple possible Operation Objects, each equivalent to an HTTP method that can be used to query the endpoint.



■ **Figure 1.1** An example of an interactive documentation generated from the OAS

If a Path Item Object is templated, the Operation Object needs to contain a Parameter Object field with the same name. The OAS also specifies the type of the parameter in a Schema Object. However, a Parameter Object may contain besides path parameters also headers, queries, or cookies. Parameters not located in the path may be optional. The optionality can be toggled by the `required` field.

Another field in Operation Object is Request Body Object that maps media types of request body to Media Type Objects. The Media Type Object then contains a Schema Object that describes **structure** and **types** of the request body. An example of a Schema Object is shown in Code listing 4. The request body may not be supplied depending on the `required` field in a Request Body Object.

```yaml
components:
  schemas:
    Pet:
      required:
        - name
      type: object
      properties:
        id:
          type: integer
          format: int64
          example: 10
        name:
          type: string
          example: doggie
        category:
          $ref: '#/components/schemas/Category'
        tags:
          type: array            # <- supports complex types
          items:                 # ↓ possible to reference other objects
            $ref: '#/components/schemas/Tag'
        status:
          type: string
          description: pet status in the store
          enum:
            - available
            - sold
```

Code listing 4 An example of a Schema Object in a Components Object

In addition to the Request Body Object, the Responses Object is another essential component of the OAS. The Responses Object contains a list of HTTP status codes that the API can respond to. Each of these status codes is then mapped to a corresponding Response Object, which provides additional information about the response that the API will send. The Response Object contains a mapping from media types to Media Type Objects which were described before. Finally, it is possible to specify whether an operation on an endpoint is secured by listing the names of Security Schemas.

The following top-level object in the OAS is the Components Object. In the Path Object

```yaml
paths:                                    # <- path object
  /pet/{petId}:                           # <- path item object
    put:                                  # <- HTTP method
      tags:
        - pet
      summary: Update a pet
      description: Update a pet by Id
      operationId: updatePet
      parameters:
        - name: petId                     # <- matching id in path
          in: path
          description: ID of pet to update
          required: true
          schema:
            type: integer
            format: int64
      requestBody:
        description: Update pet in the store
        content:                          # <- can send JSON or XML request
          application/json:               # <- media type objects
            schema:
              $ref: '#/components/schemas/Pet'
          application/xml:                # <- media type objects
            schema:
              $ref: '#/components/schemas/Pet'
        required: true
      responses:
        '200':
          description: Successful operation
          content:
            application/json:             # <- media type objects
              schema:
                $ref: '#/components/schemas/Pet'
        '400':
          description: Invalid ID supplied
      security:                           # <- list of security schemas
        - api_key: []
        - petstore_auth:
            - write:pets
            - read:pets
```

Code listing 5 An example of a Paths Object

example in Code listing 5, certain objects are not explicitly specified in place, but instead, reference a definition that is stored elsewhere. All of these references are directed to the components section of the OAS. By doing this, the object definitions can be reused, and the probability of errors is reduced.

The Security Scheme Object can be included in the global Security Requirement Object section of the OAS or the security property at the operation or path level. The Security Scheme Object allows the definition of security schemes such as basic authentication, OAuth2, or API key-based authentication.

However, in practice, only a few web services correctly use the Security Scheme Objects. Many web services do not annotate all endpoints with the Security Requirement Object or use different means of authentication and do not document it in the specification.

```yaml
components:
  securitySchemes:
    petstore_auth:
      type: oauth2
      flows:
        implicit:
          authorizationUrl: https://petstore3.swagger.io/oauth/authorize
          scopes:
            write:pets: modify pets in your account
            read:pets: read your pets
    api_key:
      type: apiKey
      name: api_key
      in: header
```

Code listing 6 An example of a Security Scheme Object

Chapter 2

Fuzzer taxonomy

Before we dive into and explore the particular features of fuzzers, let us take a more holistic view and describe different categories of fuzzers. This will provide us with an apparatus for comparing them.

Firstly, we can categorize them according to the level of knowledge the fuzzer has about the internal workings of the software being tested. While white-box fuzzers possess the most knowledge, black-box fuzzers possess the least. Then we differentiate them according to the level of knowledge about the input structure and finally characterize them according to the methods used to generate testing inputs.

In the latter parts of this chapter, we introduce new categories specific to web service fuzzers.

2.1 Awareness of internal program structure

2.1.1 White-box

White-box fuzzing is a technique that is aware of the program's internal structure and thus requires access to the source code. It uses program analysis to increase the coverage of the code [8], which can be defined as the degree to which the state space is traversed during a random walk. When program specifications are available, the fuzzer can leverage model-based testing techniques to generate test inputs and validate them using the specifications. This type of testing involves creating test cases that target specific parts of the code or data structures, enabling testers to identify and address vulnerabilities in those particular areas. White-box fuzzing is often used for software that requires a high level of security. However, integrating white-box fuzzers with the targeted program may require considerable time and knowledge of the code base.

2.1.2 Black-box

Black-box fuzzing, also known as functional fuzzing or input-based fuzzing, is a technique that requires no knowledge of the program's internal structure being tested [9]. As a result, it can

achieve higher performance and reusability compared to white-box fuzzing. Testers use black-box fuzzing to generate random or semi-random input data to see how the software responds. This approach can uncover unexpected and undocumented behavior that may be useful in offensive security. The disadvantage of black-box fuzzers is their lower efficiency, which results from the lack of information about the program.

2.1.3 Gray-box

Gray-box fuzzing is an approach that lies somewhere between white-box and black-box fuzzing. Gray-box testing uses instrumentation techniques, such as profiling, to analyze the program without requiring access to its source code. Gray-box fuzzing is more comprehensive than black-box fuzzing and faster than white-box fuzzing.

2.2 Awareness of input data structure

As described above, the essence of fuzzing is to provide a system with semi-valid input. The input must be sufficiently well-formed for the system to interpret it as valid but also contain errors that cause the system to fail. Knowledge about the input structure splits the fuzzers into two categories, smart and dumb.

2.2.1 Dumb

Dumb fuzzers do not assume any or little information about input structure. As we shall see in further chapters, a dumb web service fuzzer can, for instance, generate random JSON structures disregarding whether the web service accepts it. This will result in a simple implementation but produce numerous false positive results.

2.2.2 Smart

On the other hand, smart fuzzers are aware of the input structure and thus follow some RFC, specification, or protocol [10]. Therefore, they explore more state space and produce fewer false positives. We can split smart fuzzers into two subcategories, *smart-correct* and *smart-incorrect*.

The smart-correct fuzzers fully respect the specification; on the other hand, smart-incorrect fuzzers introduce a bit more randomness to the generation process. Sometimes, they ignore parts of the specification that may cause undefined behavior in the system under test. Fuzzers respecting the specification may achieve greater exploration of defined states.

2.3 Input generation techniques

2.3.1 Generation-based

Generation-based fuzzers are closely related to smart fuzzers. Not only are they aware of the structure and specification, but they create input that mostly follows it. For example, if we

were to fuzz a PNG decoder and decided to utilize a generation-based fuzzer, the fuzzer would need to create new PNG images from scratch based on the specification. Although implementing such a fuzzer requires significant work, Miller showed that it is worth it. According to Miller's tests, generation-based fuzzing performs up to 76% better in code coverage than mutation-based fuzzing techniques [11].

2.3.2 Mutation-based

Mutation-based fuzzers, on the other hand, do not require similar work in advance. In mutation-based fuzzing, the fuzzer takes an existing input, mutates it, and then tests the resulting input against the program [11]. For example, the fuzzer might flip a single bit in the input, change the order of bytes, or add or remove data. The goal is to generate many new inputs similar to the original input but different enough to exercise new parts of the program's logic. The mutation-based approach is reusable and the fuzzer can be employed in different situations.

Following properties are specific mostly for fuzzers of web services.

2.4 Statefulness

2.4.1 Stateful

Stateful fuzzers rely on capturing the state of the web service to generate more effective test cases. By using data received from previous responses, they can create request chains that build upon each other, leading to more accurate and extensive testing. The idea behind this approach is that the more valid a payload is, the more code coverage it will achieve. Valid payloads are more likely to bypass error checks and trigger the business logic, which can lead to more exploration of code coverage.

However, reaching more coverage does not imply finding more bugs. The contrary is often true. The carefully crafted request chains might be too correct and fail to trigger bugs. This may be because developers tend to focus on the so-called success path of the program, where everything behaves as expected. Consequently, success paths are often thoroughly tested and optimized. On the contrary, web APIs exhibit interesting behavior when presented with unexpected input. My prior work demonstrated this when the openapi-fuzzer triggered bugs by requesting objects not created before [12].

Furthermore, the construction of request chains inherently introduces a performance cost. Concrete examples of this will be discussed in the next chapter. However, implementation issues are even more significant. **Inconsistency** is one that arises. The specification does not require objects to be referred to by the same name in different endpoints; it is just a developer convention. For example, the GitLab API uses `id` and `path` to refer to the same object at different endpoints. These inconsistencies can hinder the performance of the fuzzers, and the only way to remedy them is through manual intervention or heuristics.

Another problem with using request chains is the **fake producer** problem, as described by Lin et al. [13]. The results of the GET request are frequently seen as the root of the dependency

tree, serving as natural producers for POST, PUT, and DELETE requests. As a result, calling them first to obtain those dependencies is sensible. However, if no object was created previously, they return an empty response, sabotaging the creation of the request chains. The problem is that the information about who is a producer and who is a consumer is not encoded in the OAS.

A further problem encountered when constructing request chains is **weak dependency**. To create a new project in the GitLab example, we must supply the project name. The stateful fuzzers detect that the project name can be retrieved by listing all projects using GET requests, resolving this dependency. However, creating a new project will fail since the project name must be unique. The issue arises from the inability to encode in the specification which values need to be provided by the user and which can be obtained from the API [13].

2.4.2 Stateless

Stateless fuzzers do not construct such request chains and thus do not have to deal with the challenges mentioned above. As a result, they are likely to produce less code coverage. On the other hand, stateless fuzzers can create requests that will not be entirely valid. This may lead to exploring more corner cases and undefined behavior and thus causing more bugs.

2.5 Minimization support

Minimization or shrinking support is the ability of the fuzzer to produce the smallest possible input for the system under test, which still triggers the bug. Naturally, fuzzers with support for minimization allow developers to be more efficient and find the root cause of triggered bugs easier. We have shown a motivation and example in Section 1.2.

2.6 Reproducibility support

Fuzzers that support reproducibility can send the same payload even after the fuzzing phase has ended. This feature can be used in two ways, as *replayable requests* or as *repeatable runs*. Although similar in their goal, their implementation and use cases differ. However, some fuzzers may support only one of these reproducibility methods, not both.

2.6.1 Replayable requests

The replayability feature of some fuzzers allows them to save the exact payload generated during the fuzzing phase to a file. This enables the same request to be sent later without requiring the fuzzer to run again. This capability is instrumental in scenarios where the payload causes a bug and needs to be shared with other developers for further analysis. Additionally, supporting replayable requests saves developers time because they can send the request after the fuzzing is complete and avoid having to sift through all of the logs generated during the fuzzing process, instead focusing only on the relevant logs generated by the specific request.

2.6.2 Repeatable runs

In repeatable runs, the fuzzer saves the seed used to generate the payload during the fuzzing phase. With this seed, the fuzzer can generate the same payload again in a subsequent run. This approach achieves repeatability and increases efficiency.

2.7 Buildin support for performance analysis

Web service fuzzers send a lot of requests to the APIs. During fuzzing, they obtain additional information for free. For example, fuzzers can collect the round-trip times of each request and calculate basic statistics from these data. These statistics provide an overview of the web service's performance and help find DOS-susceptible endpoints.

2.8 Authentication support

To prevent unwanted interaction from adversaries, most of the web services available on the Internet use authentication and authorization mechanisms. Authenticated users are granted varying privileges based on their authorization, allowing them to interact with the web service with the given capabilities. However, unauthenticated fuzzing often fails to trigger any business logic because requests are rejected at the application's entry, rendering it ineffective.

However, unauthorized fuzzing is vital in security testing because it exposes bugs that unauthorized users could exploit.

Chapter 3

Analysis of existing fuzzers

In this chapter, we will look at existing work in the field of fuzzing web services. We will explore and categorize existing fuzzers based on the taxonomy described in the previous chapter.

In my bachelor thesis [12], I analyzed several state-of-the-art fuzzers of that time.

Among them apiFuzz [14], a simple dumb fuzzer that does not utilize the OAS to create a new payload. Its weakness is that it performs testing on only one endpoint and has limited error reporting. Another is TnT-Fuzzer [15], that used the OAS but only for simple fields (i.e., not complex structures such as objects and arrays). Furthermore, it used a mutation technique for payload generation, which violated the data schema and was a source of inefficiency.

Due to the progress in fuzz testing research and development, more sophisticated and effective solutions have emerged since then. There is a plethora of fuzzers for testing web services, especially those that utilize the OAS. The selection of the following fuzzers was based on their impact, documentation, research, and availability of source code.

3.1 Common characteristics

Due to its vast reusability, the technique of **black-box** fuzzing is often favored when fuzzing web services through their APIs. It is especially useful in offensive security, as the attacker may not have access to the source code, leaving black-box fuzzing as the only option. Additionally, if the source code was available, it would be easier to hook up internal functions of the web service and bypass the API entirely.

One common feature of all of the fuzzers discussed in this thesis is their awareness of the input structure. They utilize the OAS to generate structured input for the system under test, making them **smart** and significantly more effective than their predecessors because the payload passes through the input parser of the web service. Otherwise, the parser would prevent the payload from triggering any business logic.

3.2 RESTler

RESTler is a fuzzer developed by a Microsoft Research team [16]. It was described in my bachelor thesis [12]; however, its influence on other fuzzers is too significant not to mention it.

The following section contains parts from my bachelor thesis [12].

The main asset of RESTler is statefulness. It claims to be the first stateful REST API fuzzer. Before we explore its numerous features, let us look into what statefulness means in the context of API fuzzers.

Almost every service has some internal state, mainly in the form of a database. Making a request to the API may change the state. For example, a POST request can be used to create a new item, while a PUT request can be used to update an existing one. The query of an item that has not been created yet through a request is useless, as it results in a response error code of 404 - Not Found, indicating a user error. Such requests lead to the consumption of valuable resources.

To minimize meaningless requests, RESTler infers producer-consumer dependencies among request types. For example, sending request `A` makes sense only if request `B` was sent before. RESTler is able to infer those dependencies by examining the OAS v2 of the fuzzed service. Then it performs a breadth-first search (BFS) to create a valid sequence of requests with satisfied dependencies.

At the start of the fuzzing, RESTler processes the OAS of the web service, it parses all requests and finds which fields are fuzzable. Fuzzable fields are those that require a user input value. It may be in headers, body, query, or path parameters. Subsequently, the RESTler fuzzer generates request chains by iterating the following two steps. First, it tries to **extend** every request by another request. This subsequent request is added to a chain only if all its dependencies are satisfied by some preceding request in the request chain. Then it **renders** the request chains. In the rendering phase, all fuzzable fields take concrete values from a small *fuzzing dictionary*. The *fuzzing dictionary* consists of several values for each type. For example, for a string type, it may consist of an empty string, some large string, and a string of small size. For an integer type, it can hold values like 0, -1, 1, and some large number. Then, the request chain is executed sequentially. RESTler collects responses from previous requests and uses them as input for fuzzable values for subsequent requests. The request chain is valid only if all responses were successful (denoted by the received status code 2XX). The 5XX status code represents a server error, and RESTler logs the request chain. Other status codes mean that the request chain was not successful and, therefore, the requests are not committed to the chain. This strategy is referred to as a BFS strategy, as it traverses the dependency graph in breadth-first order.

RESTler uses a BFS strategy to test all possible values of *fuzzing dictionary* for each fuzzable type. However, when the dictionary size is large, the number of combinations becomes astronomical, making it impractical. To mitigate this, the size of the dictionary must be kept small, which may lead to a lack of randomness.

To mitigate this, the Microsoft Research team also incorporated the random-walk strategy into RESTler. In the **extending** phase it will not try to add every matching request to every

sequence. It will rather create only one sequence and randomly add a new request to it which satisfies all the dependencies. Therefore, it will explore the search space in greater depth.

The Microsoft Research team tested RESTler on several APIs. One of those was purposefully made *Blog Post Service* with a subtle bug in checksum verification. The bug was triggered when the user used the GET method to obtain a blog (and its checksum) and then supplied the checksum (along with other parameters) via the PUT method to update the blog. Thanks to the inspection of producer-consumer dependencies between those endpoints, RESTler was able to trigger it. Moreover, thanks to the BFS strategy, RESTler was able to achieve larger code coverage. If the correct checksum was not supplied, the code path would never have been run when the blog was updated.

In addition to *Blog Post Service*, the Microsoft Research team also fuzzed GitLab. GitLab is a git hosting platform for collaboration. The Microsoft Research team fuzzed several REST resources, including commits, branches, issues, repos, groups, and projects, and compared the BFS to the random walk strategy. The results of the comparison were quite interesting. The BFS fuzzing strategy produced significantly fewer requests that signal user error, mainly error 404 - Not Found. However, when comparing the code coverage achieved over time, both strategies yielded similar results. Surprisingly, the random walk strategy obtained better results in certain REST resources.

3.3 foREST

Lin et al. identified that the main challenge in fuzzing RESTful APIs is an efficient resolution of dependencies between requests [13]. Previous fuzzers created a graph from the dependencies and then used BFS or topological sorting to traverse it. However, those graphs tend to become dense, and thus the BFS and topological sort may have a quadratic time complexity.

The novelty of foREST consists in using a different data structure to store the dependency graph. It captures the essential dependencies in a tree structure and thus decreases the time complexity of traversing from quadratic to linear.

APIs usually have multiple endpoints consisting of pairs: an URL path and an HTTP method. If we split the URL paths by the slash symbol into several tokens, they can be viewed as a tree. Each token is a node, and they can be connected in such a way as to reconstruct the URL. The empty token (`/` URL) is considered the root. Nodes have attributes in the form of HTTP methods. The URL of the endpoint is recreated by traversing the tree from the root and sending an HTTP request with the appropriate method found in the node.

Lin et al. assume that the necessary parameters or resources for visiting a child node in the tree are often already fulfilled when its parent node has been visited. Thus, they propose to traverse the tree preorder to generate valid test cases, i.e., it is more probable that the node depends on its parent rather than on its sibling.

The fuzzing algorithm uses DFS to traverse the tree, and if it reaches a node that contains some HTTP method, it sends requests to the API. HTTP methods are executed in the following order; first, it sends a GET request, followed by several POST requests, and finally, it sends PUT and DELETE requests. This order is used because POST requests create resources necessary for a subsequent update or deletion by the PUT and DELETE methods. The GET request is sent

first, as it may resolve some subsequent dependencies.

To create valid requests, foREST records the responses from requests in a resource pool. When creating a new request, it uses values from the resource pool. If it is unable to find a match, it generates a random value of a primitive type. However, it is not clear from the article [13] whether foREST can generate structured random types such as objects and arrays.

Error reporting is similar to RESTler. 2XX and 3XX status codes indicate success, while the 4XX are used for the modification of requests. The 5XX status codes denote errors in the web service that are then logged. Nonetheless, does not mention payload minimization [13].

To prove the superiority of the tree-based approach, Lin et al. implemented a BFS and topological sort strategy as well. They fuzzed a widely-used web service for building websites - WordPress with foREST for six hours. Measurements show that the tree-based approach covered most lines of code (18217), which was better than 16998 achieved by topological sort and 14649 achieved by BFS.

In a subsequent experiment, Lin et al. compared foREST with two other state-of-the-art fuzzers, namely RESTler and EvoMaster, using GitLab as the system under test. The results showed that foREST was able to detect three bugs, while RESTler and EvoMaster could not identify any. In addition, foREST achieved higher code coverage than the other two fuzzers.

3.4 Schemathesis

One of the most advanced fuzzer for web services at present is Schemathesis [17]. It leverages Hypothesis [18], a property-based testing library, to generate valid inputs that conform to the OAS. Hypothesis offers generators for primitive types, which can be combined to form more complex types. Schemathesis uses a library to create generators for the types specified in the OAS.

Schemathesis also employs Hypothesis to generate request sequences. Previously, it used a custom-made resolver to create request sequences, but the authors have now implemented a faster and more efficient solution using Hypothesis's `RuleBasedStateMachine` type that also produces better error messages. In testing, the new approach proved to be faster and generated fewer errors compared to the previous method [19].

Apart from these features, Schemathesis offers additional support for GraphQL web services and extensive customization options. Users can implement custom serializers, generators for custom types, and run commands via hooks.

In order to evaluate the effectiveness of Schemathesis, Hatfield-Dodds and Dygalo conducted an experiment in which they compared it to seven other fuzzers. They tested three different configurations of Schemathesis on sixteen open-source web services. The results showed that Schemathesis was the most effective fuzzer in terms of the number of bugs found. Additionally, it generated only one report per bug, which is uncommon among other fuzzing tools. Schemathesis also minimizes the payload generated after triggering a bug for more *efficient* debugging. It checks not only straightforward bugs denoted by the 5XX HTTP status codes but also semantic bugs, such as incomplete or overly-permissive schemas, unexpected status codes, schema non-conformance, or information disclosure. However, the web services tested were not well-known or widely-used projects, so it remains to be seen how Schemathesis would perform on more popular

services.

To conclude, many advanced fuzzers prioritize identifying dependencies among endpoints to achieve extensive code coverage, often at the cost of randomness and performance. Schemathesis stands out as the only fuzzer that utilizes payload minimization for more efficient debugging. However, none of the explored fuzzers took advantage of the significant number of requests sent to the web service to create basic statistics to identify endpoints vulnerable to DOS attacks, possibly due to the complexity of multiple request runs. It would be interesting to see how these fuzzers perform on real-world web services and compare their performance.

Table 3.1 provides a quick comparison between the described fuzzers.

	RESTler	foREST	Schemathesis
awareness of internal program structure	black-box	black-box	black-box
aware of input data structure	smart	smart	smart
input generation technique	fixed-set	generation	generation
statefulness	stateful	stateful	simplified stateful
minimization support	no	no	yes
replayable requests	yes	unknown	yes
repeatable runs	unknown	unknown	yes
buildin' performance analysis	no	no	no
authentication support	yes	yes	yes
error reporting	5XX	5XX	5XX & semantic

Table 3.1 Comparison of web service fuzzers

Chapter 4

Architectureof the new openapi-fuzzer

My previous work reported in the bachelor thesis [12] on openapi-fuzzer was a success since it successfully detected bugs in well-established web services such as Kubernetes, Hashicorp Vault, and Gitea. As a result, the architecture of the new fuzzer is based to a large extent on the previous version.

Let us now describe the architecture of the new openapi-fuzzer using the taxonomy introduced in Chapter 2. All the design choices are based on the primary goals of the fuzzer: maximize the fuzzer's effectiveness and the developer's efficiency.

4.1 Black-box

When it comes to fuzzing web services, choosing a black-box approach is a clear choice. It provides high reusability across different web services and requires minimal configuration to run on them. Sacrificing reusability for better introspection is possible; however, then it would make more sense to bypass the API and integrate the fuzzer into the internal implementation of the web service. The additional case for using a black-box design is to provide security researchers with a tool to determine vulnerabilities without access to the source code. openapi-fuzzer requires only an API URL and its OAS for efficient fuzzing.

4.2 Smart-incorrect

In general, whenever it is possible to create a *efficient smart* fuzzer, it is advisable to do so. These types of fuzzers have an understanding of the input structure and can generate more valid test cases, reducing the number of false positives. Due to its schema, openapi-fuzzer can determine the structure and types of the individual fields, as well as the media type of the payload. However, the goal of this fuzzer is to trigger as many bugs as possible, not to achieve the highest code coverage. Thus, I decided to disregard parts of the OAS and introduce randomness instead.

Although the OAS indicates which attributes and parameters are optional, openapi-fuzzer ignores this information. Instead of using heuristics and brute force, as done by other fuzzers, to determine the optimal parameter combination, openapi-fuzzer supplies all parameters and relies on the shrinker to eliminate the redundant ones. Additionally, providing more parameters tests the API more rigorously, enhancing code coverage.

openapi-fuzzer also ignores the specification regarding array limits and special formats for various fields and introduces randomness instead. openapi-fuzzer generates arrays of random sizes and allows the shrinker to determine the optimal size to trigger a bug. Additionally, not using the specified formats may cause unexpected behavior as some APIs might specify the format but not validate it.

4.3 Generation-based

The decision to use a generation-based approach follows from the fact that the grammar for the inputs is already known, thanks to the OAS. Therefore, it is logical to use this information to generate inputs. As explained in Section 2.3, research has shown that generation-based fuzzers tend to outperform mutation-based fuzzers by a considerable margin.

4.4 Stateless

Previous solutions used multiple requests or request chains to trigger bugs. RESTler and foREST use quite complex resolution algorithms, while Schemathesis uses a simpler one. The objective of complex dependency resolution is to increase code coverage. Nonetheless, my goal is not to cover the most lines of code but rather to thoroughly test the web service and uncover as many bugs as possible.

Even though stateful fuzzers offer some advantages, for example, RESTler identified certain bugs that would have gone undetected by openapi-fuzzer, I believe these bugs are relatively rare. The drawbacks of stateful fuzzing outweigh its benefits in most cases.

4.5 Minimization support

One of the most significant challenges faced by previous versions of the fuzzers was the large number of recorded results. Whenever a bug was discovered, it was logged, and the fuzzer continued to operate until terminated manually. Consequently, a considerable number of results were produced. This issue was further compounded by the fact that individual fields were subjected to a wide range of random values, making it challenging for maintainers to pinpoint the root cause of a bug. The primary goal of openapi-fuzzer is to improve the developers' efficiency. openapi-fuzzer aims to achieve this by generating a **single** *and* **minimal** report for each bug. By minimizing the input size as much as possible, openapi-fuzzer makes it easy to identify the parameter responsible for triggering the bug.

4.6 Replayable requests

Due to the benefits of replayable requests described in Subsection 2.6.1, I decided to implement this feature in the openapi-fuzzer. The openapi-fuzzer provides a subcommand to resend the stored payload after the fuzzing phase is finished.

4.7 Repeatable runs of openapi-fuzzer

The aim is to provide developers with a seamless experience. My goal is to enable the developers to test the correctness of the software during development. The proposed workflow involves the developer implementing a new feature, running the fuzzer (either locally or in a continuous integration system), finding a bug, and fixing it. However, I also want to ensure that if the same bug occurs in the future, the fuzzer can reproduce it.

To address this issue, the openapi-fuzzer saves the seeds that caused the erroneous behavior and uses them to generate future payloads before employing new random ones.

4.8 Performance analysis

The openapi-fuzzer does performance analysis during each run. It can optionally save all of the gathered information for future analysis, such as graph plotting. By examining these statistics, developers can determine which endpoints may be the bottleneck of the web service and are potentially vulnerable to a DOS attack.

4.9 Authentication support

As described in Section 2.8, authentication support helps increase the code coverage produced by the fuzzer. Therefore, I also decided to enable authenticated and unauthenticated runs of openapi-fuzzer.

	RESTler	foREST	Schemathesis	openapi-fuzzer
awareness of internal program structure	black-box	black-box	black-box	black-box
aware of input data structure	smart	smart	smart	smart
input generation technique	fixed-set	generation	generation	generation
statefulness	stateful	stateful	simplified stateful	stateless
minimization support	no	no	yes	yes
replayable requests	yes	unknown	yes	yes
repeatable runs	unknown	unknown	yes	yes
buildin' performance analysis	no	no	no	yes
authentication support	yes	yes	yes	yes
error reporting	5XX	5XX	5XX & semantic	5XX & semantic

Table 4.1 Comparison of openapi-fuzzer with other web service fuzzers

Chapter 5

Implementation

The architecture of openapi-fuzzer is mostly similar to its previous version, but the implementation details have undergone significant changes. The only things that have remained the same are the implementation language and some of the underlying libraries.

5.1 Payload minimizaton

The primary objective and challenge was to implement payload minimization, which involves iteratively decreasing the size of the generated payload that triggers a bug while minimizing the individual fields that are not responsible for the bug. Minimizing primitive types, such as strings and numbers, is straightforward, as the minimum value for a string is an empty string, and for a number, it is zero. However, minimizing complex types, such as arrays and objects, is more difficult. In the case of an array, all elements that do not trigger the bug should be removed, and those that do should be minimized. There are also cases where an element triggers a bug only when located in a particular position, in which case all preceding elements should be minimized, and the minimized element should be placed after them. An empty array is the natural minimum value for the array type. The object type follows a similar minimization process to that for arrays.

The earlier version of openapi-fuzzer utilized the Arbitrary library [20] to generate structured data from unstructured input. Although this library initially supported shrinking, it was eventually removed in later versions. I came across a discussion about the removal of this feature, where the maintainers referred to a paper by David R. MacIver and Alastair F. Donaldson, titled "Test-Case Reduction via Test-Case Generation: Insights From the Hypothesis Reducer" [21] for the minimization implementation.

The paper proposes a technique for generating smaller and smaller values using internal reduction. The key concept of internal reduction involves reducing the input sequence to the generators, rather than reducing the generated values. Generators are designed to produce smaller values when presented with a smaller input sequence.

The internal reduction has an advantage over external reduction in mitigating the test-case validity problem. For instance, when attempting to minimize an HTML document generated using external reduction, it can be challenging to reduce the document size while also maintaining

its validity. This is because it requires domain knowledge to shrink an HTML document correctly. However, the internal reduction does not require such knowledge to minimize a document, since it generates a smaller version of it based on the mutated input sequence. This feature makes internal reduction more versatile since users do not have to write a specific reducer for every test-case.

The Hypothesis library, mentioned in the previous text, is used mainly in unit testing [18]. In unit tests, developers have the most knowledge about the code, such as the number and types of function arguments and return values. Consequently, it is possible to leverage this information to implement the most efficient test case generators. However, **the API endpoint can be seen as a function** as well. An endpoint is defined as a pair of HTTP method and a URI. The OAS provides information about the number and types of input arguments. Additionally, the HTTP status code allows us to distinguish whether the request failed or not. Considering the similarity between the function and the endpoint, it is possible to reuse existing libraries instead of implementing custom reducers for the payload.

However, Hypothesis is implemented in Python, and integrating it into openapi-fuzzer would introduce a significant runtime overhead and dependencies. I sought a Rust-based alternative and discovered Proptest [22], which is a property-based testing library inspired by Hypothesis.

To integrate Proptest into openapi-fuzzer, the *Strategy* trait had to be implemented for the types used in web services, including path parameters, query parameters, headers, and JSON bodies. The Strategy trait in Proptest is analogous to the generator in Hypothesis. From now on, I will use these terms interchangeably. Implementing the Strategy trait for primitive types was a simple task since Proptest already provides implementations for them. However, the implementation of strategies for compound types such as arrays and objects posed a significant challenge.

Proptest supports *deriving* the Strategy trait for compound types. Deriving a trait is the process of generating an implementation of the trait at compile time using a macro. However, this approach would require us to compile the fuzzer with the OAS for each web service, making it impractical. Alternatively, I could manually implement the Strategy trait for the compound types.

Unfortunately, I was unable to find documentation on how to implement the Strategy trait for object types. While there is a documented way to implement a Strategy for a hashmap that could be subsequently transformed into a JSON object, the interface was not suitable. It required that all keys and values be of a certain strategy. Nevertheless, I needed to specify a different Strategy for every value based on its type in the OAS, while keeping keys unmutated. I encountered similar issues when attempting to implement a Strategy for array types.

The Proptest library had some drawbacks that made me reconsider its use. It was not being actively maintained, had a large codebase of around 30,000 lines of code, and relied heavily on macros, making the code hard to read. Therefore, I reached out to the authors of Hypothesis for advice, and one of the suggestions was to reimplement Hypothesis using their simplified implementation of the library, called Minithesis [23], and the published article [21].

Implementing a property-based testing library is a challenging task, and understanding the different naming conventions used in article [21] and Minithesis made it even more difficult. I struggled to decide between implementing my own version of Minithesis or understanding and using Proptest. During the examination of the Proptest source code, I found a comment

describing that a vector of Strategies also implements the Strategy. This enabled me to create a strategy for a JSON object by constructing a vector of pairs implementing the Strategy, and mapping it to the JSON object type.

The implementation of Strategy for JSON objects is shown in Code listing 7. The function takes a description of the JSON object type, which is detailed in Subsection 1.3.3.1. It iterates through the properties of the object and creates a vector of pairs of Strategies. To preserve the validity of the object, the keys are left intact using the `Just` Strategy, which does not mutate the value. The values are then recursively generated by calling a function to create a Strategy for the specified type. This vector of strategies also implements the Strategy trait. To be able to compose it with other types, it must be transformed into a common JSON type, `serde_json::Value`, which is achieved by using the `prop_map` and `prop_map_into` methods.

```
fn generate_json_object(
    object: &ObjectType,
) -> BoxedStrategy<serde_json::Value> {
    object
        .properties
        .iter()
        .map(|(name, schema)| {
            let schema_kind = &schema.to_item_ref().schema_kind;
            (Just(name.clone()), schema_kind_to_json(schema_kind))
        })
        .collect::<Vec<_>>()
        .prop_map(serde_json::Map::from_iter)
        .prop_map_into()
        .boxed()
}
```

Code listing 7 Implementation of the Strategy for a JSON object

5.2 Roundtrip time measurements

The timing of request roundtrips to detect DOS-prone endpoints is a secondary goal of this work. This task is straightforward to implement: the time before sending the request and after receiving the response is measured, and the difference is stored; as shown in Code listing 8. After running a test for a single endpoint, basic statistics, such as mean, maximum, minimum, and standard deviation, are calculated from the measured times. The output can be seen in Figure 5.1. Additionally, there is an option to store the timings and a flag that indicates if the request triggered a bug in a file. This information can be used to determine whether the bug-triggering payload causes any difference in the response times and, thus, to identify endpoints that are susceptible to DOS attacks.

METHOD	PATH	STATUS	MEAN (µs)	STD. DEV.	MIN (µs)	MAX (µs)
GET	auth/token/accessors/	ok	282.78	141.57	198	1822
POST	auth/token/create	ok	800.24	309.48	449	4413
POST	auth/token/create-orphan	ok	580.98	179.31	373	1221
POST	auth/token/create/{role_name}	failed	23938093.50	12048311.12	191	30003228
GET	auth/token/lookup	ok	453.84	112.99	263	1326
POST	auth/token/lookup	failed	432.90	93.12	316	663
POST	auth/token/lookup-accessor	ok	507.57	123.14	333	1334
GET	auth/token/lookup-self	ok	425.25	249.29	272	4103
POST	auth/token/lookup-self	ok	402.38	85.39	275	828
POST	auth/token/renew	failed	409.69	258.46	299	2553
POST	auth/token/renew-accessor	ok	422.10	83.25	294	952
POST	auth/token/renew-self	ok	459.29	223.91	318	3807
POST	auth/token/revoke	failed	387.33	70.45	297	636
POST	auth/token/revoke-accessor	ok	458.44	90.03	321	1036
POST	auth/token/revoke-orphan	failed	428.89	60.55	365	575
POST	auth/token/revoke-self	ok	411.98	220.71	286	3756
GET	auth/token/roles	ok	230.95	80.43	161	772
GET	auth/token/roles/{role_name}	ok	385.44	162.35	139	1175
POST	auth/token/roles/{role_name}	failed	28070625.48	7363619.71	197	30003747
DELETE	auth/token/roles/{role_name}	ok	426.44	257.64	129	3298
POST	auth/token/tidy	ok	469.65	115.40	281	924
GET	cubbyhole/{path}	ok	247.67	128.33	152	1749
POST	cubbyhole/{path}	ok	306.37	137.05	132	892
DELETE	cubbyhole/{path}	ok	310.03	125.08	132	781
POST	identity/alias	ok	402.06	175.55	267	3087
GET	identity/alias/id	ok	186.76	33.80	134	359
GET	identity/alias/id/{id}	ok	269.29	117.69	110	778
POST	identity/alias/id/{id}	failed	28127234.78	7262202.07	189	30003085
DELETE	identity/alias/id/{id}	ok	590.70	295.39	177	2318
POST	identity/entity	ok	821.09	196.56	446	1255
POST	identity/entity-alias	ok	467.34	229.11	390	3975
GET	identity/entity-alias/id	ok	231.85	43.04	174	439
GET	identity/entity-alias/id/{id}	ok	325.96	220.04	132	2970
POST	identity/entity-alias/id/{id}	failed	28127286.75	7262200.18	180	30003093
DELETE	identity/entity-alias/id/{id}	ok	565.14	288.27	171	2508
POST	identity/entity/batch-delete	ok	789.90	372.51	433	5802
GET	identity/entity/id	ok	222.97	37.94	190	400
GET	identity/entity/id/{id}	ok	351.16	167.11	140	1227
POST	identity/entity/id/{id}	failed	27668846.48	8035027.96	196	30003069
DELETE	identity/entity/id/{id}	ok	606.97	310.70	181	2249
POST	identity/entity/merge	ok	819.85	355.43	417	5543
GET	identity/entity/name	ok	216.04	33.10	191	436
GET	identity/entity/name/{name}	ok	350.99	147.17	150	1060

■ **Figure 5.1** Output of the openapi-fuzzer

```
for path in paths:
    for method in methods:
        result = TestRunner.run(config, any::<Payload>(spec), |payload| -> {
            start_time = time::now()
            response = send_request(payload)
            elapsed = time::now() - start_time

            if response is unexpected:
                return Fail(response)
            return Success
        })

        match result:
            case Success => report_ok(),
            case Fail(minimized_payload) =>
                save(minimized_payload)
                report_fail()

        save_statistics()
```

Code listing 8 Pseudocode of the main fuzzing loop

5.3 Detailed run

Only two arguments are mandatory to run the openapi-fuzzer. A path to the OAS and a URL of the API of the web service. The complete list of available options for a run subcommand is shown in Code listing 9. After parsing the command line arguments, it parses the OAS. As described in Subsection 1.3.3.1, the OAS can have references to objects implemented in the component section. To use the objects, the fuzzer needs to resolve the references. We use the `openapi_utils` [24] library to do it.

Then we instantiate the fuzzer with a configuration retrieved from the command line. The fuzzer then iterates through all combinations of URI paths and HTTP methods specified in the OAS, as shown in Code listing 8. During each iteration, a new TestRunner is instantiated from the Proptest library. Since we implemented a Strategy trait for the types needed to fuzz web service, Proptest TestRunner is able to create and subsequently shrink the payload. The TestRunner takes a configuration and a closure in which a request is sent to the API. Before sending the request, the fuzzer can override generated headers with headers specified from the command line. After receiving the response, we determine whether the payload is *interesting*. A payload is interesting if it caused a 5XX status code or a status code that is not among the expected status codes defined in the OAS.

The TestRunner in the default configuration tries to send at most 256 requests to the API, and if all of them pass, it declares the endpoint ok and continues to fuzz the next one. The number of attempts can be specified on the command line. When an interesting payload is found, the TestRunner switches to a shrinking phase and tries to minimize it. After the minimization phase is over, the payload is saved in a file in JSON format for future use.

```
openapi-fuzzer run --help
Usage: openapi-fuzzer run -s <spec> -u <url>
    [-i <ignore-status-code...>]
    [-H <header...>]
    [--max-test-case-count <max-test-case-count>]
    [-o <results-dir>]
    [--stats-dir <stats-dir>]

run openapi-fuzzer

Options:
  -s, --spec        path to OAS file
  -u, --url         url of api to fuzz
  -i, --ignore-status-code
                    status codes that will not be considered as finding
  -H, --header      additional header to send
  --max-test-case-count
                    maximum number of test cases that will run for each
                    combination of endpoint and method (default: 256)
  -o, --results-dir directory for results with minimal generated payload used
                    for resending requests (default: results).
  --stats-dir       directory for request times statistics. if no value is
                    supplied, statistics will not be saved
  --help            display usage information
```

Code listing 9 Help message of openapi-fuzzer run subcommand

5.4 Replayability

To help with debugging, I implemented a resending functionality for openapi-fuzzer. It is run by the `resend` subcommand and requires the path to the saved payload generated by the fuzzer. During the fuzzing process, users can include additional headers, which are typically used for authentication. However, these may have a short lifespan, and timed-out headers will cause client errors and prevent the reproduction of the results. To address this, I also added support for extra headers in the `resend` subcommand.

5.5 Repeatability

The repeatability is implemented by saving the seed of an interesting payload to a regressions file. In the next run, openapi-fuzzer generates the new payload using the existing seed. If the payload leads to a failure, openapi-fuzzer reports it and proceeds to the next endpoint. Otherwise, it generates new payloads from random seeds. The Proptest library handles the saving and retrieval of the seed from the regressions file, making this functionality straightforward to implement.

Chapter 6

Testing and evaluation

To evaluate the effectiveness and efficiency of openapi-fuzzer, I conducted a series of extensive testing and measurements. openapi-fuzzer was executed on three battle-tested web services: Hashicorp Vault, Gitea, and Kubernetes; for exact versions of these web services, see Table 6.1. These services were fuzzed both with and without authentication to determine the impact of authentication on the results.

Web service	Version
Vault	v1.13.0 (a4cf0dc4437de35fce4860857b64569d092a9b5a), built 2023-03-01T14:58:13Z
Gitea	1.19.0-rc1 built with GNU Make 4.1, go1.20.1 : bindata, sqlite, sqlite_unlock_notify
Kubernetes	v1.26.1 (8f94681cd294aa8cfd3407b8191f6c70214973a4), built 2023-01-18T15:51:25Z

Table 6.1 List of versions of fuzzed web services

6.1 What is a bug

As described in previous sections, openapi-fuzzer is able to report server errors (denoted by 5XX status codes) as well as semantic errors (denoted by the remaining status codes). However, the OAS [7] does not require that all possible status codes be specified in the OAS. Although missing status codes may indicate missing documentation or potential underlying bugs, they do not directly violate the OAS. The missing 4XX client errors are the most common. Therefore, I decided to count only the 5XX server errors as a bug. When openapi-fuzzer encounters a status code other than 5XX or 4XX, it will be noted separately.

The following tables have format $N + M$ where N is the number of 5XX status codes, and M is the number of status codes other than 5XX and 4XX.

6.2 Evaluation

The main objective of the testing was to answer the following questions:

- Did the new architecture of openapi-fuzzer hurt its effectiveness?
- Is the new version of openapi-fuzzer able to minimize the payload?
- How does openapi-fuzzer compare to other state-of-the-art fuzzers?

6.2.1 Sustained effectiveness

To find out whether the new version of openapi-fuzzer can find at least as many bugs as the old one, I decided to fuzz all the aforementioned software. Both fuzzers completed two runs, unauthenticated and authenticated. However, as mentioned in Chapter 4, the old version of openapi-fuzzer does not support termination detection. Hence, I ran the new version of openapi-fuzzer and measured the time required to finish. Then I ran the old version of openapi-fuzzer and killed the process after it ran out of time.

As shown in Table 6.2, the new version of openapi-fuzzer triggered a higher number of bugs in all cases. I attribute the higher effectiveness to better allocation of resources. When the new fuzzer finds a bug, it logs it and no longer fuzzes the endpoint.

	openapi-fuzzer	old version
Vault	63 + 0	13 + 0
Vault authenticated	64 + 2	12 + 2
Gitea authenticated	4 + 2	4 + 0
Kubernetes authenticated	22 + 2	1 + 1

Table 6.2 The number of bugs found with the new and old version of the fuzzer

6.2.2 Payload minimization

The first benefit of openapi-fuzzer is evident at first sight. When it finds a bug, it minimizes it, saves it, and continues to fuzz the next endpoint. This was not the case in the old version. The old version saved the finding but continued with the fuzzing, too. As a result, when the bug was easy to find, the old version of openapi-fuzzer produced many files. Moreover, all the files were different since the payload was randomly generated. Therefore, when we compare the two versions of the fuzzer, we will compare the sum of sizes of generated payload for each endpoint by the openapi-fuzzer to the sum of sizes of one generated payload for each endpoint and to the sum of all generated payloads for each endpoint by the old version of the fuzzer.

As we can see in Table 6.3, openapi-fuzzer significantly outperforms the old version in minimization. But it outperforms it in effectiveness as well, as shown in Subsection 6.2.1. All the findings are in GitLab repository [25].

	openapi-fuzzer	old version (one payload per endpoint)	old version (all payloads)
Vault	6'046	108'967	29'515'310
Vault authenticated	6'145	99'976	27'530'977
Gitea authenticated	417	34'166	6'119'932
Kubernetes authenticated	5'242	11'462	63'068

Table 6.3 Payload sizes (in bytes) between openapi-fuzzer and the old

Let us take a look at a specific example of a minimized payload, shown in Code listing 10. There are 10 query parameters and one path parameter. All of the 10 query parameters are minimized to their canonical form. The path parameter is the only fuzzable value that is not completely minimized. After sending the request using openapi-fuzzer `resend` subcommand, we can verify that the payload indeed still triggers the bug. Moreover, after manual inspection, I found out that the path parameters are also in their minimal form. The web service does not respond with a server error if we change or delete any of the "`..?`" characters.

```
{
    "payload": {
        "query_params": [
            ["allowWatchBookmarks", ""],
            ["continue", ""],
            ["fieldSelector", ""],
            ["labelSelector", ""],
            ["limit", ""],
            ["pretty", ""],
            ["resourceVersion", ""],
            ["resourceVersionMatch", ""],
            ["timeoutSeconds", ""],
            ["watch", ""]
        ],
        "path_params": [
            ["namespace", "..?"]
        ],
        "headers": [],
        "body": null
    },
    "path": "api/v1/watch/namespaces/{namespace}/podtemplates",
    "method": "GET"
}
```

Code listing 10 A minimized payload from fuzzing Kubernetes

Unfortunately, the old version of openapi-fuzzer did not produce a payload similar to that shown in Code listing 10. Therefore, let us look at examples of payloads where both fuzzers triggered the same bug.

The payload in Code listing 11 produced by openapi-fuzzer shrunk all but one parameter to the minimum. The only nonempty parameter contains a single Unicode character. Just

from looking at the generated payload, we can guess the cause of the bug. The field is named `last_read_at`, which probably was expecting some form of date. The implementation probably tries to parse the date by slicing and indexing into it. However, slicing a Unicode string may not produce a valid UTF-8 string. This conversion then likely fails and crashes the program.

```
{
    "payload": {
        "query_params": [
            ["last_read_at", " "],
            ["all", ""],
            ["status-types", ""],
            ["to-status", ""]
        ],
        "path_params": [],
        "headers": [],
        "body": null
    },
    "path": "notifications",
    "method": "PUT"
}
```

Code listing 11 A minified payload produced by openapi-fuzzer v0.2 from fuzzing Gitea

On the other hand, the payload in Code listing 12 produced by the old version of the fuzzer contains a Unicode string in every field, making it difficult to guess the cause of the bug. We are not even able to determine what part of the Unicode string triggers the but. It may be its length, some special character, or they might not be relevant at all.

From these measurements, we can conclude that the new version has excellent minimization abilities compared to the old version, but also in general.

6.2.3 Comparison with other state-of-the-art fuzzers

I have chosen the Schemathesis fuzzer for comparison with openapi-fuzzer because foREST is not open-sourced yet, and Schemathesis outperformed RESTler by a large margin [17]. I ran Schemathesis using the following command: `st run <path/to/oas.yaml> --base-url <url-of-web-service>`, with the optional `-H` flag to add an authentication header. From Table 6.4, we see that openapi-fuzzer outperformed the Schemathesis fuzzer in all cases.

Furthermore, Schemathesis encountered internal errors when fuzzing most of the services. On the other hand, openapi-fuzzer did not encounter any. Detailed numbers of internal errors are given in Table 6.5.

I also measured the run-times of both fuzzers; results are shown in Table 6.6. The sources used for this comparison are available in [25]. The table shows that, in general, openapi-fuzzer fuzzes web services faster than Schemathesis. However, there were two cases where Schemathesis seemingly outperformed openapi-fuzzer, but in fact, these runs are incomparable. The first was during authenticated fuzzing of Gitea, where Schemathesis encountered an internal error

```
{
    "payload": {
        "query_params": [
        [
            "last_read_at",
            "%F2%88%99%AC%F1%AC%9F%B1%F4%88%98%AF%F3%9E%9D%B3...."
        ],
        [
            "all",
            "%F0%A9%AC%8F%F2%84%83%99%F0%91%A4%B0%F2%A9%A1%9C...."
        ],
        [
            "status-types",
            "%F0%A2%BC%84%F1%B6%B0%AF%F3%9B%9F%82%F0%AC%93%BC...."
        ],
        [
            "to-status",
            "%F1%92%B6%9E%F2%9E%86%B6%F3%AE%9D%82%F2%A1%B2%BA...."
        ]
    ],
        "path_params": [],
        "headers": [],
        "body": null
    },
    "path": "notifications",
    "method": "PUT"
}
```


Code listing 12 A payload produced by old version from fuzzing Gitea

	openapi-fuzzer	Schemathesis
Vault	63 + 0	4
Vault authenticated	64 + 2	4
Gitea	0	0
Gitea authenticated	4 + 2	0
Kubernetes	0	0
Kubernetes authenticated	22 + 2	20

Table 6.4 The number of found bugs between openapi-fuzzer and Schemathesis

during fuzzing every endpoint and terminated fuzzing prematurely. The second was during Vault fuzzing, where openapi-fuzzer discovered bugs that caused Vault to respond much slower, resulting in longer fuzzing times. Wheres Schemathesis was not slowed down by Vault since it did not discover these bugs, see Table 6.4

	openapi-fuzzer	Schemathesis
Vault	0	0
Vault authenticated	0	0
Gitea	0	0
Gitea authenticated	0	325/325
Kubernetes	0	202/723
Kubernetes authenticated	0	213/723

Table 6.5 The number of internal errors between openapi-fuzzer and Schemathesis

	openapi-fuzzer	Schemathesis
Vault	n/a	8.2
Vault authenticated	n/a	7.8
Gitea	0.5	25.5
Gitea authenticated	22.5	0.5
Kubernetes	5.5	34
Kubernetes authenticated	6	110

Table 6.6 Approximate running times (in minutes) of openapi-fuzzer and Schemathesis

6.3 Types of bugs

After further analysis of the bugs discovered by openapi-fuzzer, we can conclude that the majority of them were caused by several reasons. These include querying an object not created previously, parsing errors such as problems with parsing dates or numbers, and errors when openapi-fuzzer provided special characters as input to the web service.

6.4 Performance analysis

Besides bug-finding, openapi-fuzzer also conducts a performance analysis of individual endpoints. Unless there are noticeable performance issues, performance analysis is useful mainly for the developers of the web service. Considering that they know what response times are acceptable. openapi-fuzzer displays a summary of the analysis for every endpoint and can be instrumented to save further details. An example summary is shown in Figure 5.1. As we can see in the figure, `auth/token/create/role_name` and `auth/token/lookup-accessor` are among the Vault endpoints that require a significantly longer time to respond.

Figure 6.1 shows graphs from detailed statistics for the `auth/token/create/role_name` endpoint generated by openapi-fuzzer. As we can see from the first graph, the long round-trip times can be attributed to the bug-triggering payload. Thus, this endpoint might be susceptible to DOS attacks. In the second graph, we can see a relatively even distribution of round-trip times around 30 seconds.

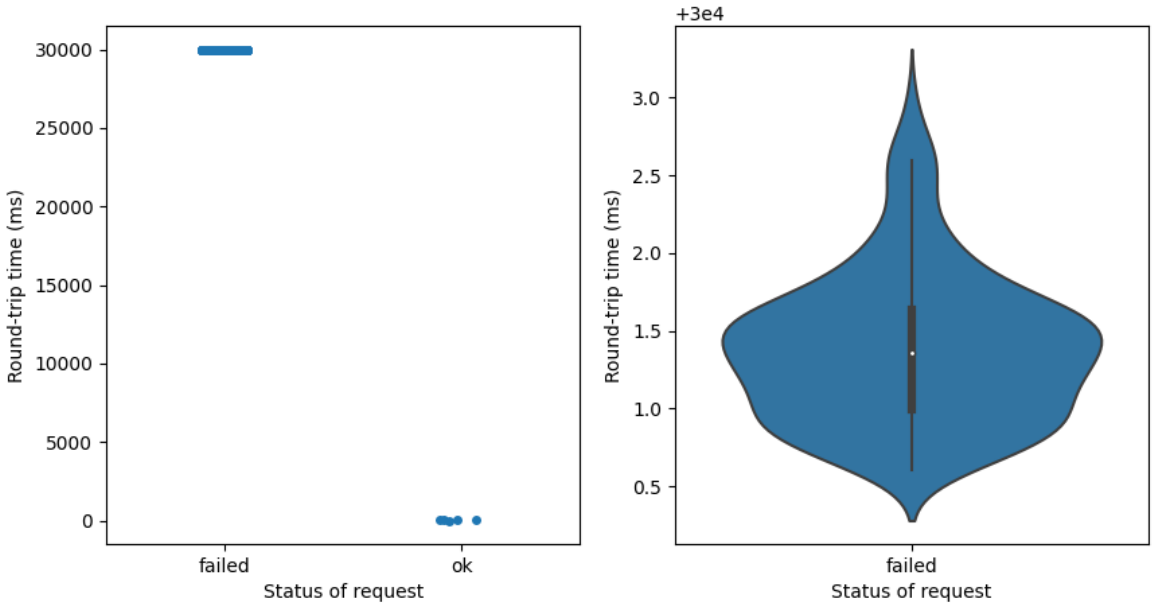


Figure 6.1 Round-trip times of requests sent by openapi-fuzzer to single endpoint

Chapter 7

Conclusion

Testing proved to be the most effective way to ensure the quality of the software. Numerous testing techniques have emerged, but most require manual test writing. Nevertheless, these techniques do not scale well with the increasing complexity of software systems. Thus, the area of automated random testing, fuzzing, was researched.

The main objective of this thesis was to redesign the previous version of openapi-fuzzer developed within my bachelor thesis and to add support for payload minimization. After comprehensively investigating available options, I implemented minimization using the Proptest library. To evaluate minimization, I compared the payload sizes between the two versions of openapi-fuzzer. Not only is the generated payload now much smaller, but most of the time, its size is also as small as possible. Furthermore, a minimized payload makes identification of the bug trivial, as was also demonstrated in Subsection 6.2.2.

Additionally, I compared the openapi-fuzzer's ability to find bugs with the state-of-the-art Schemathesis fuzzer. In the article by Schematheisis' authors [17], they conducted an experiment in which the Schemathesis fuzzer outperformed all other state-of-the-art fuzzers, including RESTler [16]. I performed an authenticated and unauthenticated fuzzing of well-known battle-tested web services, including Kubernetes, Hashicorp Vault, and Gitea. The results show that openapi-fuzzer was able to find more bugs in a shorter time in every tested software. Moreover, it did not experience any internal error, which was not the case with Schemathesis.

The secondary objective of this thesis was to add support for the simple detection of endpoints susceptible to DOS attacks. The openapi-fuzzer records the round-trip times for every request and creates simple statistics for each endpoint. It provides an option to save the round-trip times alongside a flag (denoting whether the request caused a bug) for further processing. From those measurements, I subsequently generated graphs to inspect the performance of the selected endpoint in greater detail.

7.1 Future work

Although the openapi-fuzzer has yielded satisfactory results, there is still room for improvement. The fuzzer could make better use of the OAS by extracting additional information to enhance its

fuzzing effectiveness. Moreover, there is potential to improve the performance of openapi-fuzzer.

7.1.1 Removing attributes

An option to improve the effectiveness of openapi-fuzzer is to include support for removing specific attributes. As explained in Section 4.2, openapi-fuzzer presently ignores that all required attributes must be included in a payload and always includes them in a payload. It relies on the shrinker to minimize the attributes. However, the web service may behave differently when given a payload with a minimal attribute or without the attribute. Although the current strategy is beneficial for exploring most states, as described in Subsection 2.2.2, the one suggested could reveal more bugs since web services may display undefined behavior if they do not receive a mandatory attribute. To implement this feature, Proptest's `UnionStrategy` could be used.

7.1.2 Simple coverage

Another feature that could benefit the users of openapi-fuzzer is support for simple code coverage. The OAS lists the possible responses for each endpoint. The fuzzer could record the status codes and responses it received and thus determine whether it received all possible.

7.1.3 Strict mode

Support for the strict mode could be added to fuzz the web services more thoroughly. As for now, openapi-fuzzer checks only the status code of the received request and reports an error if the status code is 5XX or is not defined in the specification. We could take it one step further and check if the received response conforms to the structure defined in the specification. This would enable openapi-fuzzer to report bugs, such as missing or additional fields or a different payload structure.

7.1.4 Parallelization

As mentioned in Chapter 4, openapi-fuzzer is a stateless fuzzer. Moreover, during the fuzz testing of the web services, no bug was triggered by a combination of multiple requests. Thus, we can deduce that the requests do not interfere with each other. This feature and observation might enable openapi-fuzzer to send the requests in parallel, resulting in increased performance.

Appendix A

Appendix

The source code of the openapi-fuzzer can be found on GitHub at the following URL: `https://github.com/matusf/openapi-fuzzer`. Version 0.2 is in branch `0.2.0` and soon will be merged to the master branch. The README file contains instructions on building the fuzzer, as well as on its usage.

Bibliography

1. MILLER, Barton P; FREDRIKSEN, Lars; SO, Bryan. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*. 1990, vol. 33, no. 12, pp. 32–44.
2. MCNALLY, Richard; YIU, Ken; GROVE, Duncan; GERHARDY, Damien. Fuzzing: the state of the art. 2012.
3. MATÚŠ, Ferech. *OpenAPI Fuzzer*. 2021. Version 0.1.3. Available also from: `https://github.com/matusf/openapi-fuzzer`.
4. RAGGETT, Dave; LE HORS, Arnaud; JACOBS, Ian, et al. HTML 4.01 Specification. *W3C recommendation*. 1999, vol. 24.
5. RODRIGUEZ, Alex. Restful web services: The basics. *IBM developerWorks*. 2008, vol. 33, p. 18.
6. FREDRICH, Todd. Restful service best practices. *Recommendations for Creating Web Services*. 2012, pp. 1–34.
7. *OpenAPI Specification* [`https://spec.openapis.org/oas/v3.1.0`]. 2021.
8. NEYSTADT, John. Automated penetration testing with white-box fuzzing. *MSDN Library*. 2008.
9. TAKANEN, Ari; DEMOTT, Jared D; MILLER, Charles; KETTUNEN, Atte. *Fuzzing for software security testing and quality assurance*. Artech House, 2018.
10. MILLER, Charles. How smart is intelligent fuzzing-or-how stupid is dumb fuzzing. *Independent Security Evaluators*. 2007.
11. MILLER, Charlie; PETERSON, Zachary NJ, et al. Analysis of mutation and generation-based fuzzing. In: *Independent Security Evaluators, Tech. Rep.* 2007, vol. 4.
12. FERECH, Matúš. *Application-level fuzzing*. 2021. Bachelor's Thesis. Comenius University Bratislava.
13. LIN, Jiaxian; LI, Tianyu; CHEN, Yang; WEI, Guangsheng; LIN, Jiadong; ZHANG, Sen; XU, Hui. foREST: A Tree-based Approach for Fuzzing RESTful APIs. *arXiv preprint arXiv:2203.02906*. 2022.
14. *apiFuzz*. 2020. Available also from: `https://github.com/viralpoetry/api-fuzz`.

15. *TnT-Fuzzer*. 2020. Available also from: https://github.com/Teebytes/TnT-Fuzzer.

16. ATLIDAKIS, Vaggelis; GODEFROID, Patrice; POLISHCHUK, Marina. Restler: Stateful rest api fuzzing. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 748–758.

17. HATFIELD-DODDS, Zac; DYGALO, Dmitry. 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.

18. MACIVER, David R.; HATFIELD-DODDS, Zac; MANY OTHER CONTRIBUTORS. Hypothesis: A new approach to property-based testing. 2019. Available from DOI: 10.21105/joss.01891.

19. HATFIELD-DODDS, Zac; DYGALO, Dmitry. Deriving Semantics-Aware Fuzzers from Web API Schemas. *arXiv e-prints*. 2021, arXiv–2112.

20. *Arbitrary*. 2020. Available also from: https://github.com/rust-fuzz/arbitrary.

21. MACIVER, David R.; DONALDSON, Alastair F. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper). In: HIRSCHFELD, Robert; PAPE, Tobias (eds.). *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, vol. 166, 13:1–13:27. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-154-2. ISSN 1868-8969. Available from DOI: 10.4230/LIPIcs.ECOOP.2020.13.

22. *Proptest*. 2023. Available also from: https://github.com/proptest-rs/proptest.

23. *Minithesis*. 2022. Available also from: https://github.com/DRMacIver/minithesis.

24. *Openapi Utils*. 2022. Available also from: https://github.com/JordiPolo/oas_proxy/tree/master/openapi_utils.

25. *Findings of openapi-fuzzer*. 2023. Available also from: https://gitlab.fit.cvut.cz/ferecmat/openapi-fuzzer-findings.