Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

Reverse-engineering REST APIs: Enriching OpenAPI Specifications through Fuzzing

vorgelegt von

Jan Hartkopf

Matrikelnummer 7195323

Studiengang Informatik

MIN-Fakultät

Fachbereich Informatik

eingereicht am 25. November 2024

Erstgutachter: Prof. Dr.-Ing. Mathias Fischer

Zweitgutachter: August See, M.Sc.

Acknowledgements

Before the start of my thesis, I would like to express my sincere gratitude to inovex GmbH, who have not only magnificently supported me during writing my thesis, but also over the past 5+ years, providing me with opportunities to work on exciting projects during my studies, and granting me the freedom to contribute to real-world problems. I would especially like to thank Simon Dreher from inovex for continuously taking the time to review my progress and giving me valuable feedback, as well as Daniel Bäurer from inovex for proofreading the thesis.

Thank you to my family and friends, who have always been a vigorous source of support, without whom it would not have been that easy.

1 Abstract

APIs are ubiquitious in all industries, powering client-server applications, such as mobile apps, web platforms or IoT devices. Since these services expose sensitive data like financial, health, business and other personal information, precautions must be taken to ensure a secure and private implementation and operation of those complex systems. History shows the dramatic consequences that can happen because of insecure REST APIs. It is absolutely crucial for developers and penetration testers to find and fix such vulnerabilities before threat actors have the chance to exploit them for malicious activities.

Fuzzing is an effective method for finding security vulnerabilities in software. REST-focused fuzzers specifically target REST APIs and generally work with a formal specification of the API protocol, with OpenAPI being the de-facto standard for these API specifications. However, many REST APIs remain undocumented, e.g. proprietary SaaS APIs, or are only partially or informally documented. In such cases, effective fuzzing requires reverse-engineering of the API protocol and proper documentation using OpenAPI's standardized format.

This thesis proposes new methods for fuzzing-based REST API reverse engineering, leveraging reasoning of API dependencies, discovery of authentication requirements, detailed parameter type inference, and exploration of other interesting API characteristics. For REST-based protocol reverse engineering, this is the first time that fuzzing techniques are applied. It is evaluated how effective these methods work using a novel, specification-based approach, and how effectively exploitative REST fuzzers can use those reverse-engineered specifications to find new API issues. The results demonstrate favorable outcome and deeper understanding of undocumented REST APIs: Compared to current methods, the proposed techniques achieve a notable improvement of 9.3% for reconstructed information, and a significant boost of 71.2% for information that was completely undocumented previously. The evaluation results show the potential for exposing further security vulnerabilities, but also depict current limitations of exploitative REST fuzzers.

Contents

1 Abstract **3**

2 Introduction **6**

3 Background **8**
3.1 REST . . . 8
3.2 OpenAPI specification . . . 10
3.3 REST API security . . . 11
3.3.1 Access control . . . 12
3.3.2 Input validation . . . 13
3.3.3 Lack of resources and rate limiting . . . 13
3.3.4 Excessive data exposure . . . 14

4 Requirements and related work **15**
4.1 Requirements . . . 15
4.2 Related work . . . 15
4.2.1 Protocol reverse engineering . . . 16
4.2.1.1 General protocol reverse enngineering . . . 16
4.2.1.2 REST-based protocol reverse engineering . . . 16
4.2.2 Fuzzing . . . 17
4.2.2.1 Black-box fuzzing . . . 18
4.2.2.2 White-box fuzzing . . . 18
4.2.2.3 Gray-box fuzzing . . . 19
4.2.2.4 Web application fuzzing . . . 19
4.2.2.5 Intelligent generation of input values and language inference 20
4.2.3 Black-box REST API fuzzers . . . 21
4.2.3.1 Microsoft RESTler . . . 22
4.2.3.2 Various fuzzers . . . 22
4.2.4 Open problems . . . 23
4.2.4.1 REST-based protocol reverse engineering . . . 23
4.2.4.2 REST API fuzzing . . . 23

5 Optimization of reverse-engineering REST APIs **25**
5.1 Base OpenAPI spec creation . . . 26
5.1.1 Traffic collection . . . 26
5.1.2 Converting traffic flows to OpenAPI specs . . . 26
5.2 Static traffic analysis . . . 27
5.2.1 Producer-consumer dependencies . . . 27
5.2.1.1 Finding eligible producers . . . 28
5.2.1.2 Analyzing producer-consumer connection . . . 29
5.2.2 Request generation . . . 31
5.2.3 Dependencies in OpenAPI specifications . . . 32

5.2.4 Modeling request body parameter dependencies . . . 34
5.3 Fuzzing-based reverse engineering . . . 35
5.3.1 Grammar inference . . . 36
5.3.2 Authentication requirements . . . 37
5.3.2.1 Header-based authentication . . . 38
5.3.2.2 Cookie authentication . . . 40
5.3.3 Path variable detection . . . 41
5.3.4 Required request body properties . . . 44
5.3.5 Response objects . . . 46

6 Evaluation **47**
6.1 Implementation . . . 47
6.2 Metrics . . . 48
6.2.1 General approach . . . 49
6.2.2 Path mapping . . . 52
6.2.3 Evaluating path variables . . . 53
6.2.4 Evaluating global components . . . 54
6.2.5 Path coverage . . . 55
6.3 Target applications, setup and traffic collection . . . 55
6.3.1 Environmental setup . . . 56
6.3.2 Traffic collection . . . 57
6.4 RQ1: Effectivity of reverse engineering . . . 58
6.4.1 Mattermost . . . 58
6.4.2 Open WebUI . . . 60
6.4.3 Bunnybook . . . 61
6.4.4 Summary . . . 62
6.5 RQ2: Effectivity of exploitative REST fuzzing . . . 63
6.5.1 Mattermost . . . 63
6.5.2 Open WebUI . . . 64
6.5.3 Bunnybook . . . 65
6.5.4 Summary . . . 66

7 Limitations **67**

8 Conclusion **69**

References **71**

2 Introduction

APIs are widely used in client-server applications to facilitate data exchange between clients and servers. Their secure and robust implementation is crucial to guard data and resources against attackers. Automated software testing helps with finding potential bugs early on, minimizing risks like information exposure or Denial-of-Service attacks. An effective technique for testing REST APIs is fuzzing, which uses the REST API's specification. However, poorly documented or completely undocumented REST APIs lack proper API specifications, increasing the barrier for efficient, spec-based fuzz testing. Reverse-engineering such REST APIs is required to generate specifications in order to enable successful fuzzing.

Due to the widespread use of HTTP-based REST APIs in web applications, mobile apps, and other client-server applications (e.g. in automotive industry, IoT devices, cloud platforms etc.), combined with their exposure of sensitive and business-critical data to diverse client applications, including third-party ones, REST APIs require thorough testing. Fuzzing can help identify security vulnerabilities and bugs by generating unusual inputs and monitoring the software's behavior. Focusing on REST allows to make certain assumptions about the API, e.g. using specific HTTP verbs for specific API operations, or interpreting HTTP status codes in particular ways. As a consequence, this enables REST-optimized, high-level fuzzing.

REST APIs are often documented in standardized format according to the OpenAPI specification, which is the de-facto standard, but may also be documented in non-standardized form. Proper documentation enables researchers and developers not only to implement the API and use it in a foreseeable way, but also to run fuzzing tests against it. REST API fuzzers generally rely on these API specifications [ZA23], and the completeness and correctness of those is crucial for meaningful fuzzing results.

Undocumented or partially documented REST APIs might, for example, only be used through proprietary clients provided by the developer. Nonetheless, those APIs could also suffer from the issues mentioned and remain interesting targets for attackers. Reverse-engineering these poorly documented or undocumented REST APIs is necessary to create OpenAPI specifications that enable successful fuzzing.

The main contributions of this thesis are as follows:

- A novel static traffic analysis algorithm that can infer producer-consumer dependencies between API endpoints. This enables deeper understanding of API dependencies, and is used for supporting the fuzzing process.
- New fuzzing methods for REST API reverse engineering, targeting security requirements, parameter type inference and other interesting API characteristics to gain insights into the target REST API. For the first time, fuzzing techniques are used for REST-based protocol reverse engineering. This includes the *request generation* feature, a proposed method for verification of path variables through resolving dependency graphs inferred via static traffic analysis. The resulting API specification is stored in OpenAPI format for further use by exploitative REST fuzzers.

- Establishment of metrics that quantify OpenAPI specs regarding information gain.

It is evaluated how effective the proposed reverse engineering methods work, and how effectively exploitative REST fuzzers find new issues and vulnerabilities in APIs using the optimized, reverse-engineered OpenAPI specs. Evaluation outcomes indicate a better understanding of the targeted APIs thanks to the reverse engineering methods. They also show a promising outlook for finding more vulnerabilities, but are held back by current limitations of REST fuzzers.

The thesis is structured as follows: Chapter 3 gives background information about required topics. Chapter 4 discusses requirements and related work. In Chapter 5, the new methods are presented for gaining information of REST APIs through reverse engineering. Chapter 6 discusses the implementation, the evaluation approach and its results, while Chapter 7 covers the method's limitations. Chapter 8 summarizes the acquired insights and concludes the thesis.

3 Background

As this thesis covers the fuzzing of REST APIs, the concepts of REST and fuzzing are crucial to understand. Additionally, REST fuzzers rely on so-called *OpenAPI specifications*, also taking an important role. The security of REST APIs is also discussed in this chapter. Those required concepts are explained in the following sections. Fuzzing is discussed in Chapter 4.

3.1 REST

REST [Fie00] stands for *Representational State Transfer* and was introduced by Roy Fielding in 2000. It is an "architectural style for distributed hypermedia systems" which defines multiple software engineering principles for building modern applications in the Web. REST's constraints are the separation of client and server, statelessness on the server, indication of cacheability, usage of a uniform interface as well as a layered system. From those constraints follow certain desired non-functional properties such as scaleability, performance and portability.

The constraints are quickly explained in the following:

- **Client-server**: The client-server constraint enables separation of concerns. It allows to detach the user interface from data storage, improving portability of the user interface (implementations for multiple platforms) and decreasing the complexity of server components.
- **Stateless**: Every request from client to server must contain all required information for the server to understand it. Session state must solely be stored client-side and the server cannot make use of any stored context. One important induced property is scaleability as the server can quickly free used resources after fulfilling a request. It also does not have to manage session data across multiple requests. However, this constraint can reduce network performance (because required data must be sent with every request).
- **Cache**: Server responses are defined as either cacheable or non-cacheable. Cacheable responses can be cached by clients and therefore can make use of locally stored data instead of sending the same request again. This improves efficiency, scaleability and performance with a trade-off towards reliability in case stale data remains cached.
- **Uniform Interface**: All components use a general, uniform interface, decoupling services from their implementations, which in turn allows independent evolvability. The REST uniform interface is again defined by four further interface constraints, including resource identification and manipulation of resources through representation.
- **Layered System**: Systems can consist of hierarchical layers where each layer can only see the next layer (i.e. the one they are communicating with), decreasing overall system complexity. Layers enable encapsulation of legacy services as well as introduction of intermediary systems like load balancers or caching servers.

- **Code-On-Demand (optional)**: Clients can download code from the server and execute it. Therefore, clients can easily be extended after initial deployment and not all functionality must be pre-implemented, simplifying client complexity. As it also reduces visibility, it is considered an optional constraint within REST.

Today, REST APIs are commonly used in modern web services and mobile applications across all industries. Implementations typically make use of different standards and protocols, e.g. HTTP (as the transport protocol and for hypermedia controls) and JSON or XML (for formatting messages). To give a more concrete example, consider the following HTTP request to a simple REST API listening at *api.example.com:80*:

```
GET /cars/1234 HTTP/2
```

Listing 3.1: Example for an REST API request

The client sends an HTTP request (HTTP version 2) with the URI */cars/1234*, where *1234* is the ID of the resource the client is interested in. It uses the HTTP verb *GET*, meaning that the client requests a read-only copy of the resource from the server. The server's response looks as follows:

```
HTTP/2 200 OK

{
    "car": {
        "id": 1234,
        "manufacturer": "XYZ",
        "price": 50000,
        "sold": false,
    }
}
```

Listing 3.2: Example REST API response containing a representation of the requested resource

The response contains the HTTP status code 200 which indicates a successful response. The response body includes the server's representation of the requested resource (identifiable by its ID) and is formatted in JSON, allowing easy processing through the client application.

Now let us assume the client needs to update the resource in the server's database. The client simply edits its internal representation as needed and then updates the resource in the database via the following HTTP request:

```
PUT /cars/1234 HTTP/2

{
    "car": {
        "id": 1234,
        "manufacturer": "XYZ",
        "price": 50000,
        "sold": true,
    }
}
```

Listing 3.3: Example REST API request for resource modification

The HTTP verb is *PUT*, indicating that an existing resource should be updated, and the resource is still identified via the same resource ID in the HTTP URI. The client's updated resource representation is contained in the HTTP body. The server responds with the following:

```
1 HTTP/2 200 OK
```

Listing 3.4: Example of a successful REST API response

The response code implies the server has successfully updated the resource according to the client's modifications. Following GET requests for this resource will now return the updated resource. This simple example especially demonstrates the Uniform Interface constraint, where resources are identified in requests (in this case via URIs) and that resources can be modified using representations.

Concluding, an API is considered a REST API if it follows the REST architectural style. The term *RESTful* (i.e. an adjective) API is also common, generally meaning a web service implementing REST; however, both terms can be used interchangeably [AWS24]. In this thesis, the term REST API always refers to an HTTP-based API implementing the REST architecture.

3.2 OpenAPI specification

The OpenAPI specification [OAS24] (formerly *Swagger Specification*) is a standard for defining HTTP-based APIs in machine- and human-readable format, easing development and consuming of those services. The OpenAPI standard is maintained by the OpenAPI Initiative, under governance by The Linux Foundation [OAI]. It enables discovering and understanding API functionality and therefore also acts as a form of documentation. OpenAPI specs are the de-facto standard and are used in many software projects, both Closed- and Open-source.

There are programs[1] to automatically generate code, including test cases, based on OpenAPI specs for a multitude of programming languages. This makes OpenAPI interesting for, e.g., centrally designing a service specification first, and then generating basic client and server applications for the required languages and platforms, or to allow easy integration with existing APIs for third-party developers. Additionally, generating OpenAPI specs from implemented code is also possible.

OpenAPI specs usually define and document REST APIs including their implemented paths, HTTP verbs, accepted parameters (either in path, query or headers), data types, possible response schemas, authentication methods and more. Listing 3.5 shows a simplified example of one endpoint defined in GitLab's OpenAPI spec [GL24]:

1. https://github.com/OpenAPITools/openapi-generator

```yaml
paths:
  /api/v4/projects/{id}/access_requests:
    get:
      summary: Gets a list of access requests for a project.
        parameters:
        - name: id
          in: path
          description: The ID or URL-encoded path of the project owned
             by the authenticated user
          required: true
          schema:
            type: string
        - name: page
          in: query
          description: Current page number
          schema:
            type: integer
            format: int32
            default: 1
        - name: per_page
          in: query
          description: Number of items per page
          schema:
            type: integer
            format: int32
            default: 20
```

Listing 3.5: Simplified excerpt from an OpenAPI spec

The above example defines one path with the *GET* HTTP verb, also called *endpoint* in combination, and documents that the first (and only) path expects one parameter called "id" together with its precise location in the path. In this excerpt, the only accepted HTTP verb is GET, and the API details for using that verb contain a short summary about the endpoint as well as multiple parameters. The required parameter (*required: true*) "id" must be placed into the path (*in: path*) (where the parameter's location is already defined). Two other parameters "page" and "per_page" are optional (no *required: true*) and should be sent as a query parameter (*in: query*). For all parameters, a schema for the value is defined (e.g. *string* or *int32*) and the two optional parameters also contain a default value (applied by the server) in case they are not set client-side.

As seen in the example, the OpenAPI spec succeeds in providing a machine-readable REST API interface definition which remains interpretable by humans (even though the whole OpenAPI specification is more complex). Concluding, it is a widely known standard for detailed interface definitions and can be used by numerous tools for many different tasks.

3.3 REST API security

Security vulnerabilities in REST APIs can lead to serious implications as those services often have access to critical data, like personally identifiable information (e.g. name, phone number, address), financial data (e.g. credit card and bank details) or health data (e.g. patient records) and frequently represent a central building block for an organization's infrastructure. Issues in REST APIs can therefore result in data breaches, Denial of Service, privilege escalation,

financial losses or reputational damages. For prevention of such types of issues, OWASP, a well-known non-profit organization for information security, publishes best practices [OWA24] as well as various risks [OASP] for securing APIs. To get a better understanding of what these include, some important points about common vulnerabilities and risks are explained.

3.3.1 Access control

Non-public REST API endpoints should be protected with some kind of access control that handles user or machine authentication and authorization. As the technology used for authentication tokens, JWTs (JSON Web Tokens) have emerged as a popular solution [OWA24]. They are simple JSON data structures and usually contain identifiable information like username, user ID etc. as well as granted access rights or roles (so-called claims). JWTs can additionally include arbitrary data injectable by the token issuer. Tokens are cryptographically signed, either using asymmetric signatures or HMAC, to detect tampering. As JWTs are generally stateless (all required information is contained in the token itself), they seem to perfectly fit for usage with REST APIs, which also are stateless.

Another method for access control is the usage of API keys [OWA24]. These are usually represented by randomly generated strings, which must be sent with every API request, and are associated with a certain project in whose name a request is made against the API. API keys are useful for basic access control as they enable rate limiting capabilities and blocking of anonymous usage. [GCA]

The bottom line is both JWTs and API keys can be used for access control in REST APIs, but have different use cases. Those access control methods allow the implementation of rate limiting in order to restrict resource usage and to mitigate DoS attacks. It is important to correctly implement access controls as, otherwise, threat actors could still make use of various security vulnerabilities. As shown in the OWASP API Security Top 10 [OASP], which lists the ten most crucial risks in API security according to OWASP, 4 of the 10 risks are related to broken authentication or broken authorization. For example, OWASP's *Broken Authentication* risk [OWABA] is number 2 on the list and mentions multiple vulnerabilities to keep in mind when implementing authentication, e.g.:

- Credential stuffing attacks based on a list of known username/password combinations (e.g. collected from a data breach)
- Brute forcing of user accounts without presenting CAPTCHAs to the user or implementing account locking
- Sending sensitive data in URLs (those could be leaked e.g. via server logs)
- Inproper validation of access tokens, e.g. acceptance of expired or unsigned JWTs
- Allowing weak user passwords

This underlines that, even though REST APIs might be designed with access control in mind, this requires careful examination as there are many issues that need to be considered and properly addressed, or could otherwise open the door to threat actors.

3.3.2 Input validation

Validation of inputs for REST APIs is important. Threat actors could try to exploit possible vulnerabilities based on missing input validation, e.g. via file upload, direct input or parameters. For example, injection attacks are common and could lead to information disclosure, data loss or arbitrary code execution [OWAIN].

Therefore, APIs must not rely on user inputs being valid and must instead properly sanitize any inputs, be it from users, machines or other services. This could be done by validating input length, format and type, using strong types for input parameters, checking string input with regular expressions (without introducing *Regular expression DoS* [Cro03] vulnerabilities), defining request size limits or using robust parsing libraries for reading input messages [OWA24].

Security issues based on insufficient input validation are an interesting target for fuzzers because it is their speciality to generate a large number of inputs while trying to find ones that trigger unwanted behavior.

3.3.3 Lack of resources and rate limiting

All API requests consume some type of resources on the server, i.e. CPU cycles, memory, storage or network bandwidth. The exact amount of required resources is dependent on multiple factors like input parameters or the implementation of business logic [OWARL] and their runtime complexity. Of course, REST APIs should be able to fulfil all (valid) client requests in an acceptable amount of time without getting overwhelmed by the sheer amount of requests.

Threat actors could try to exhaust those resources by sending malicious requests (so-called *resource exhaustion attacks*). For example, attackers could exploit expensive requests that generate data on the server [OWARL], consuming many server resources. Another way is to misuse parameters that control the amount of items returned by setting those to a very high value, causing the server to process and respond with a huge number of items [OWARL]. The impact on the server could be amplified by sending those types of requests in high frequency. This makes it difficult for the server to respond to valid client requests in time as no or too few resources may be available.

To guard against resource exhaustion attacks, resource limits can be instantiated which can, e.g., be applied to connected microservices. Also, numeric parameters should only allow values within a limited boundary. [OWARL]

Rate limiting is another effective countermeasure: The number of allowed requests per client (identified by auth token, API key, IP address etc.) within a defined time frame is limited, and further requests are blocked directly without allocating any more server resources.

Furthermore, Web Application Firewalls (WAFs) and Content Delivery Networks (CDNs) can act as a first line of defense, preventing malicious requests from hitting the API backend servers in the first place and reducing server load in general by caching responses.

3.3.4 Excessive data exposure

APIs expose data by nature. However, only data that is absolutely required for the client should be exposed. Even though clients might filter the received data from the server and only expose the required data to the user, filtered out data can be recovered through, e.g., traffic traces of the API responses. Therefore, the server should only respond with data that is necessary for the client in the first place. For example, instead of serializing an object, which might contain sensitive data, directly to JSON, the server should pick the required properties and put them into a new object which is then exposed via its response. [OWADE]

It might not be trivial to detect such excessive data exposures through the sole means of automatic tools like fuzzers because differentiation between intentional and unintentional information exposure is hard. Software that can reliably detect this type of issue requires a deep understanding of the application and its API. [OWADE]

4 Requirements and related work

This chapter focuses on the requirements as well as on related work for this thesis' topic.

4.1 Requirements

For optimization of the reverse engineering process for REST APIs, current reverse engineering methods and REST fuzzers are examined. The following requirements are set to achieve a more effective reverse engineering process:

- Methods should understand and leverage REST properties, such as API endpoints, and other HTTP mechanisms, like verbs, request bodies and response codes. This enables working on a higher level specifically targeted at REST-based network protocols.
- On the REST level, they should understand API dependencies between multiple endpoints, i.e. producer-consumer dependencies. The ability to create dependency graphs enables deeper comprehension of the target REST API.
- Data types of common parameters, e.g. HTTP query parameters, should be defined more specifically than with general data types, like string or integer. Detailed parameter constraints should be reverse-engineered, which allows later REST fuzzing towards security issues, more efficient generation of fuzzing values as well as application of type-specific exploitation logic.
- In addition to the specific requirements already mentioned, methods should make use of other techniques for explorative fuzzing to find as much information about the target API as possible. This could include gathering information about API authentication or exploring unknown API endpoints.
- Methods should work based on independently captured traffic between an official API client and the REST API. This facilitates protocol analysis and gives valid starting values for fuzzing.
- All results gained via the reverse engineering process should be saved in standardized format as defined in the OpenAPI specification [OAS24]. This makes the methods' end results comparable with other approaches and enables further usage through other REST tools, e.g. exploitative REST fuzzers.

4.2 Related work

This section explores the current techniques for reverse-engineering network protocols and fuzzing, as well as open problems in this area.

4.2.1 Protocol reverse engineering

Protocol reverse engineering (PRE) has the goal of inferring an unknown protocol by observing communication between participants using that protocol. PRE is useful for multiple reasons, e.g. security audits, network protocol conformance testing, or malware protocol analysis (e.g. for analyzing botnet traffic). There are different approaches which can be split into two categories: *Network trace* (NetT) and *Execution trace* (ExeT) [Hua+22]. Network trace approaches use captured network communication (e.g. in libpcap format) from previously monitored messages between applications. On the other hand, in inference based on Execution trace, the application binary, its source code, or a sequence of binary instructions are analyzed. [Duc+18]

4.2.1.1 General protocol reverse enngineering

There are multiple research papers and tools available that try to infer a protocol based on network traces. For example, *FieldHunter* can reverse-engineer text-based and binary protocols by splitting network flows into network messages and performing message tokenization on those. Using statistical characteristics, it then narrows down the meaning of message fields and places them into one of multiple categories, e.g. message type, message length, host identifier or session identifier. [Duc+18]

4.2.1.2 REST-based protocol reverse engineering

While most of these tools established in research papers generally aim to support a wider range of network protocols, there are also more specialized tools that only support one protocol type. Reverse-engineering of REST APIs specifically can be seen as such a class and a subcategory of PRE. For this area, there is not much research available. However, there are tools that try to infer the REST API protocol from traffic traces (i.e. a NetT approach), for example *mitmproxy2swagger* [MP2S] or *har-to-openapi*[1], by converting mitmproxy or HAR (HTTP Archive) flow files to OpenAPI specs.

mitmproxy2swagger uses HTTP traffic flows either in mitmproxy or HAR format as input. Using a combination of HTTP headers, query parameters, path segments, HTTP verbs and request and response bodies, it constructs an internal representation of available REST API endpoints with all properties it has learned from the traffic flows. That representation is then output in OpenAPI format. Because the tool works on a higher level and makes protocol-specific assumptions (i.e. HTTP-based REST), it can work more efficiently than tools that generalize on lower level protocol types, e.g. IP-based protocols.

Another tool leveraging REST API reverse engineering is *Optic* [OPTI]. It helps developers optimizing the accuracy of their API documentation by finding differences between the developer's OpenAPI spec and the actual API implementation. For that, it captures API traffic from running test cases (which must be configured to send their traffic through Optic's proxy server), compiles a new OpenAPI spec, and compares it to the developer's spec. The scope of this project is slightly different than the other mentioned REST-based PRE tools as it does not directly aim to reverse-engineer APIs (rather, this is used as a method in order to support maintenance of OpenAPI specs). Optic requires proper test cases to begin with, so it does not support traffic

1. https://github.com/jonluca/har-to-openapi

captured from official API clients, and tests must cover all possible request combinations in order to get properly documented, e.g. a test case for an enum parameter must make use of all enum values for Optic to document that parameter thoroughly.

Table 4.1 shows a high-level overview of how the thesis' approach compares to related work.

Requirements	mitmproxy2swagger	har-to-openapi	Optic	RESTler	Thesis' fuzzer
Leverages REST properties	●	●	●	●	●
API dependencies				◐[a]	●
Detailed parameter types					●
Dynamic exploration				◐[b]	●
Can utilize traffic captured from official client	●	●			●
Results using OpenAPI	●	●	●		●

◐Partially fulfilled, ●Fulfilled

Table 4.1: Related tools for REST-based PRE and REST fuzzing

a. Supported, but only when the input spec uses a strict parameter naming scheme, or dependencies are manually added.

b. The focus lays on exploitative fuzzing rather than explorative, but this can still partially be considered dynamic exploration (of security issues), and potentially some other API characteristics.

4.2.2 Fuzzing

Fuzzing [Lia+18] (abbreviation for *fuzz testing*) is a method for automated software testing which uses a large number of invalid inputs, fed to the system under test, in order to find exploitable vulnerabilities. It is an effective, practial and fast approach for exposing bugs in software and often uncovers previously unknown problems. The idea is to use malformed data as inputs (e.g. random, mutated or special characters, or any mix of those) which are probable to trigger a bug. There also exist multiple strategies for sophisticated input generation (also called test case generation), e.g. coverage-based strategies or genetic algorithms. Modern fuzzers are very intelligent to expose unknown bugs, and fuzzing therefore became an important step in today's software testing processes.

However, the problem in generating test cases is that fuzzers are not necessarily expected to know the inner workings of the system under test and therefore have to deal with some sort of "blindness". This is deemed a main disadvantage of fuzzing which can lead to low code coverage, but there are different techniques trying to overcome this issue, e.g. combining static and dynamic information of the target software.

While the basic concept is the same for all fuzzers, they can be put into three different categories: *Black-box*, *Gray-box* and *White-box* fuzzers. All types work with different amounts of information from the target at runtime and have their own advantages and disadvantages. They

are discussed in the following. Two more fuzzing topics, namely fuzzing of web applications, and generation of acceptable fuzzing inputs, are additionally covered later in this section.

4.2.2.1 Black-box fuzzing

Black-box fuzzers [Lia+18] do not have access to any information about the system under test. The target program is literally seen as a black box. Based on a seed, inputs are generated and mutated randomly using predefined rules. Examples for mutations include bit flips, byte copies and byte removals, but can also make use of grammars or input-specific knowledge.

Black-box fuzzers have to deal with the lack of knowledge about the target program due to this technique's nature. This makes it harder to efficiently generate test cases and has parallels with brute-force searching. As a result, this can lead to low code coverage, i.e. only a relatively low percentage of possible code paths is covered by the fuzz tests. However, Black-box fuzzers are popular in software testing because of its simplicity to use, while still being able to effectively find bugs, making it universally applicable.

Remote Black-box fuzzing A REST API can be seen as a *Remote Black-box* [ZA23]. It is a *Black-box* because its inner workings are not visible from the outside. For example, there is no source code available, meaning methods like profiling or static code analysis cannot be applied. It is *remote* because the API runs on a server somewhere in a datacenter, where no access to the underlying (virtual) server is possible. That means, the Black-box's behavior visible from the outside, like CPU utilization or memory consumption, is also not available.

Of course, REST APIs must not always be Remote Black-boxes. There are many Open-source REST API implementations available. Those and even available Closed-source products can be run on a local computer with full physical and logical access. However, for this thesis, it is assumed that the analyzed REST APIs are indeed Remote Black-boxes.

As a result, fuzzing these types of REST APIs cannot make use of White-box or Gray-box approaches as only the API's designated communication interface on a high abstraction level can be used for analysis. Sending and receiving messages using the RESTful, API-specific protocol in a Black-box manner is therefore the only method to probe and analyze the API.

4.2.2.2 White-box fuzzing

White-box fuzzing [Lia+18] solves the biggest disadvantage of Black-box fuzzing by analyzing the target's source code. The inner workings of the target are fully disclosed and accessible to the fuzzer. Using different search techniques, e.g. coverage-maximizing heuristic search algorithms, they can analyze possible code paths, solve path constraints and generate appropriate test cases for them. Theoretically, White-box fuzzers can therefore achieve an optimal code coverage of 100%. In practice, however, they are not able to provide such a high code coverage due to the complexity in real software. Reasons for this are the numerous possible code paths and the imprecision of solving path constraints using symbolic execution (i.e. finding and solving input constraints for executing specific code paths).

While White-box fuzzing can theoretically provide the best results regarding code coverage, it is held back by being specific to the programming language [ZA23] and especially by requiring

access to the target's source code. It might therefore not always fulfil the requirements for practical appliance.

4.2.2.3 Gray-box fuzzing

As the name suggests, Gray-box fuzzing [Lia+18] is in between White-box and Black-box fuzzing and uses partial knowledge of the system under test. That knowledge is often obtained using code instrumentation (i.e. dynamic analysis like profiling, measuring function execution times or tracing code paths). Based on the gained insight, Gray-box fuzzers adjust their mutation strategies to maximize code coverage or to find bugs faster. Another approach is to use code instrumentation in combination with taint analysis (i.e. following a user-controlled variable through the execution paths) and to adjust that variable's values to analyze paths.

In contrast to White-box fuzzing, Gray-box fuzzing only uses the gained information to memorize already explored code paths. That way, they try to guide test case generation towards unexplored paths, but cannot guarantee that further tests actually find new paths (instead of known ones). White-box fuzzing is more efficient in this case, since they do not need to discover new paths as they are already revealed through the target's source code, allowing more systematic test generation and higher code coverage.

In conclusion, Gray-box fuzzing helps decreasing the lack of knowledge to a certain degree compared to Black-box fuzzing, but cannot achieve the level of knowledge that White-box fuzzing can. Like Black-box fuzzing, the target's source code is not needed with Gray-box fuzzing and hence can also help in such cases.

4.2.2.4 Web application fuzzing

Much research and tools are existent for Black-box fuzzing of web applications [AAB21]. Generally, those applications consist of a user-facing frontend (HTML and JavaScript code) and backend components. In [AAB22], there were two main problems identified for achieving good test coverage when Black-box fuzzing web applications: (i) Dynamic code such as JavaScript, which increases the security analysis' complexity and (ii) generating valid inputs that are accepted by backend applications.

While the first issue is specific to frontend applications using dynamic code, the second one is also relevant for REST API fuzzing (various web applications communicate via REST APIs anyway). Many different approaches exist for generating valid input data for web applications: Required information can be filled in manually by a human tester; however, this approach is costly, time-consuming and vulnerable to human error. Therefore, this approach is impractical. Another approach is to generate inputs randomly. The problem with this is to generate values actually accepted by the application. Hence, the success rate of this approach is obviously random as well, and some web pages might be missed (and therefore cannot be tested) due to unsuitable inputs. External data sources can also be used for input generation of meaningful values. The required type of information can be inferred from keywords derived from input field names. A drawback of this technique is that it is less precise due to producing domain-specific values, based on the exact data source used.

Ultimately, the paper's main contribution is the proposal of an automated approach using analysis of static and dynamic HTML resources as well as the construction of input constraints based on

client-side validation functions. The derived input constraints can then be solved. Acceptable inputs generated based on the constraints are then usable for web application fuzzing.

Although the paper's approach is called "Black-box", it can be argued that this is not a Black-box approach since source code available to the client by nature (HTML and JavaScript, even though it may be obfuscated or generated/minimized code) is analyzed to optimize construction of suitable input values.

In another paper [Gau+21] by Gauthier et al. about Gray-box fuzzing web applications, the problem of REST API inference is also mentioned. Their approach for API modeling is to use a client-side crawler which analyzes the application components and executes its functionality. The HTTP requests to the backend servers are captured via a MITM proxy. The REST inference is then done based on these traffic flows, while concrete values are aggregated for type inference.

Similarly, in [DAT24], the authors Dharmaadi, Athanasopoulos, and Turkmen also mention type inference of API parameters as one problem, which is relevant for generating values that are not instantly rejected by the API. A possible solution for this is *Format-encoded Type (FET) inference*, defining data types that are more specific than generic types such as string or integer. In an evaluation, over 1200 REST APIs were analyzed with concrete parameter values, resulting in 21 different FETs. Furthermore, API dependencies are identified as another problem since certain API endpoints may only be properly fuzzed when their dependencies are fulfilled. For example, endpoints may require specific resources to work with which need to be created first via another endpoint. These problems are also relevant for this thesis.

4.2.2.5 Intelligent generation of input values and language inference

Another problem in the area of fuzzing is to generate input values for the target application. In general, the difficulty is to find input values which are acceptable enough for the software's basic input validation, but problematic enough to trigger bugs in later code paths, e.g. the business logic. Inputs generated by fuzzers must therefore adhere to the basic input constraints in order to get through the programs's first "line of defense". Among others, compilers, interpreters and network protocols usually have strict requirements on how exactly valid inputs should look like [Lia+18].

For input generation, also called test case generation, there are two main methods: *Mutation-based* and *grammar-based* generation. Mutation-based methods use known, valid input values and perform various mutation strategies, resulting in new inputs, called *mutations*. On the other hand, grammar-based approaches utilize a formal specification of acceptable inputs, i.e. a formal grammar, and can generate valid inputs without any seeds. [Lia+18]

For example, the popular fuzzer *American Fuzzy Lop* (*AFL*) [AFL] is mutatation-based and does not make use of a formal input model. Based on user-supplied samples, or seeds, AFL applies different mutation strategies, like bit flips, bit replacements, or multi-byte block deletions and uses the results as the next program input. While effective, this can lead to a higher number of invalid inputs as these are generated more or less randomly, instead of being constrained by a formal model. In addition, AFL also makes use of light instrumentation in order to maximize code coverage.

Grammar-based fuzzers can generate input values more effectively, potentially yielding a higher acceptance ratio of generated inputs. This becomes important especially for programs where

specifically-formed values are required, e.g. in network protocols or standardized languages such as JSON or XML. However, a formal grammar definition is needed for the fuzzer to be able to craft valid, grammar-conform inputs. Either the grammar has to be defined manually by the user, or it must be inferred through other processes. For this problem, sophisticated approaches exist that try to automatically learn the input grammar of a program.

One solution presented in [GPS17] uses statistical learning techniques based on recurrent neural networks. It uses sample inputs to train a generative machine learning model in an unsupervised learning fashion. Using the resulting ML model, new program inputs can be generated. For evaluation, the PDF specification is chosen and a model for generating valid PDF objects (as contained in PDF files) is trained. The authors Godefroid, Peleg, and Singh use a diverse set of 534 PDF files as training data. The results are promising as the model is able to generate between 70% and 97% of valid PDF objects (depending on the chosen hyperparameters). However, training requires a large amount of data, as usual in machine learning, and a large amount of resources. For the evaluation, training needed between 2 and 10 hours.

GLADE [Bas+17] is an implementation of an algorithm that synthesizes a context-free grammar which encodes the language of valid input values. It uses a set of samples that are accepted by the target program and an oracle for membership queries, deciding whether a guessed value, based on the current state of the learnt grammar, belongs to the program's grammar. GLADE only relys on the supplied samples and Black-box access to the oracle, completely without dynamic instrumentation of the target program, unlike other language learning tools. It outperforms similar algorithms, like *L-Star* [Ang87], both in performance and quality of results. The algorithm has an overall computational complexity of $\mathcal{O}(n^4)$ (where n is the length of the seed input). However, due to using a greedy search strategy to efficiently search the language space, suboptimal grammars can be produced, i.e. only resulting in a subset of the target program's actual language. Additionally, GLADE also has built-in fuzzing functionality and can output valid values for a successfully learnt grammar.

In more detail, the GLADE algorithm uses generalization operators which are applied to the given seed values. The used operators are *repetition* and *alternation*. Repetition means that a part of a seed, a substring, can have a defined number of occurrences. Alternation defines multiple choices which are applicable to a substring. After this is processed for the seeds, the algorithm returns multiple candidate languages. A finite amount of inputs is generated for those and, through the use of an oracle, checked whether they belong to the target language. If an input is rejected by the oracle, the according candidate language is dismissed. Measured by an increasing recall without sacrificing precision, the best candidate language is chosen and the described process is iteratively repeated with it until no further generalizations are possible.

4.2.3 Black-box REST API fuzzers

This section covers the current state of different types of Black-box fuzzers, including REST API fuzzers, and more general HTTP fuzzers. Multiple such fuzzers as well as types of detectable API issues are discussed in the following.

4.2.3.1 Microsoft RESTler

RESTler [AGP19] is an Open-source Black-box REST API fuzzer by Microsoft published in 2019, together with an accompanying paper. Compared to other fuzzers, RESTler is a stateful fuzzer, meaning it not only tests random request sequences, but intelligently saves explored properties of the API. It stores dependencies between specific API endpoints, for example, if endpoint *B* requires a resource ID returned by endpoint *A*. Additionally, dynamic feedback from REST APIs is recognized from prior test executions, e.g. learning that a request *C* is refused after a request sequence *A*, *B*. RESTler first performs static analysis of an OpenAPI spec and generates fuzzing tests based on that specification, before applying learned properties from previous API responses. That way, the search space of possible request sequences can be largely minimized, resulting in better performance and faster results.

In the evaluation, RESTler was tested against a GitLab instance, an Open-source Git server. RESTler was able to find multiple new bugs in GitLab including ones that are security-relevant. However, it is still a prototype and there are open questions remaining. For example, can the results from fuzzing GitLab be generalized and applied to other REST APIs as well? More types of fuzzing tests could be added to find more types of bugs. What types of security vulnerabilities are hidden behind REST APIs? Its other limitations include lack of support for authentication methods other than token-based authentication (like OAuth) as well as server-side redirects (i.e. HTTP status codes 301, 302 etc.).

While RESTler was able to only detect API issues based on the API responses' status codes at first, it was later extended by more detection mechanisms to additionally find logic-based API issues: In [AGP20], further rules were specified and implemented in the RESTler fuzzer. Specifically, those are the four new rules defined:

- (i) Use-after-free rule: A resource that has been deleted must no longer be accessible.
- (ii) Resource-leak rule: A resource that was not created successfully must not be accessible and not "leak" any side-effect in the backend service state.
- (iii) Resource-hierarchy rule: A child resource of a parent resource must not be accessible from another parent resource.
- (iv) User-namespace rule: A resource created in a user namespace must not be accessible from another user namespace.

These rules extend the REST API errors observable via HTTP status codes (i.e. crashes or other unexpected responses) by more logic-based issues.

4.2.3.2 Various fuzzers

Other Black-box REST API fuzzers include *EvoMaster*, which uses evolutionary computation for test generation, and also supports White-box fuzzing, although only for JVM languages and JavaScript. Another such software is *Schemathesis*, which can derive the structure and semantics of a REST API based on its API specification, e.g. in OpenAPI format. It seems to be especially user-friendly, providing easy installation methods and extensive documentation. [ZA23]

Another REST fuzzer worth mentioning is OWASP's *OFFAT* [OFFAT] (short for *OFFensive Api Tester*). Based on OpenAPI specs, it tests the target API for prevalent vulnerabilities as defined

by OWASP. The implemented fuzz tests check for vulnerabilities such as SQL injection, data exposure, broken access control, basic command injection and more. However, the project is still a work in progress and might not be feature complete at this point.

Further interesting fuzzers include *CATS* [CATS] and *ffuf* [FFUF]. CATS is a REST API fuzzer and uses OpenAPI specs as input (but also allows custom non-OpenAPI schemas). It offers highly customizable test scenarios, e.g. using only specific types of fuzzing tests, filtering for HTTP status codes, methods, request paths etc. using a domain-specific language and has extensive documentation. Ffuf is a more general HTTP web fuzzer which allows relatively simple but powerful test definition via CLI parameters and can integrate external mutators.

4.2.4 Open problems

4.2.4.1 REST-based protocol reverse engineering

Since scientific research for REST-specific reverse engineering is barely available, current tools are examined regarding their feature state. mitmproxy2swagger [MP2S] seems to be the current state of the art solution for this purpose. The tool is strictly a static analysis solution; it is fed with traffic traces and then executes static analysis on them. At the same time, this is a great disadvantage to this approach because only API features which are already contained in these traces are available for analysis. It does not try to actively gather additional information based on the already available traffic (e.g. via fuzzing).

Furthermore, while basic detection of path variables is possible, only explicitly known variable patterns are supported, which default to integer-only values [M2SP], being rather unusual in modern REST APIs. This also means that such patterns must already be known in advance, before running the software. In general, any assumptions made during the static analysis process are not dynamically verifiable since the software does not contact the target API at all.

4.2.4.2 REST API fuzzing

In [ZA23], Zhang and Arcuri analyzed and discussed open problems in existing REST fuzzers. For that, they compared seven state-of-the-art fuzzers (including RESTler) by running them against 18 Open-source REST APIs.

Analyzing the execution logs and source code of all tested fuzzers, there were multiple common issues identified: (i) The schema of the underlying OpenAPI specs might contain errors. That does not mean that the spec does not contain all implemented API endpoints, but rather that the OpenAPI schema itself has syntactic errors. This could be caused by manually written specs, or by inaccurate tools that try to automatically extract API endpoint details from the API's source code or other libraries. As a consequence, this causes poorly implemented fuzzers to simply crash and to not generate any fuzzing tests at all. (ii) The OpenAPI spec might be underspecified, meaning that not all API endpoints are actually mentioned in the spec, are not complete (e.g. one endpoint might be missing a certain parameter), or might even be wrong (e.g. wrong data type for a parameter). Using such a spec can lead to errors that are returned early (i.e. *Early Return* pattern) in the source code due to invalid input, skipping the actual business logic in the REST API. (iii) A REST API might require authentication for specific or all API endpoints, or elevated permissions for certain actions (i.e. multiple user accounts). It is not trivial to infer the

correct authentication protocol just from an OpenAPI spec or even from an API's source code if available. (iv) There may be constraints on certain string parameters, e.g. a date might be required (in a specific format) or the parameter must be an UUID. The required format could be derived from the parameter's name or the source code (again, if available).

The paper mentions more open challenges regarding REST API fuzzing, like database interactions or mocking external service dependencies. Those mentioned here are only a few examples, however, also seem to be more fundamental and of more relevance. While incompleteness of OpenAPI specs is probably the most specific problem to REST APIs (however, could likely be transferred to, e.g., GraphQL or RPC APIs), the authentication and string constraints problems seem to be more generalized and would be applicable to other problems as well. Regarding the goal of establishing a complete pipeline for reverse engineering and fuzzing REST APIs, the OpenAPI spec issues are of special importance.

5 Optimization of reverse-engineering REST APIs

The thesis aims to improve the current state of reverse-engineering REST APIs. As seen in Chapter 4, current problems in fuzzing REST APIs include the specific definition of parameter types and constraints as well as according generation of fuzzing strings, incomplete or wrong OpenAPI specs, lack of authentication requirements, and API dependencies. These problems apply to exploitative REST fuzzing and could potentially be solved through better specification of the API protocol, which can be achieved through explorative fuzzing. In this chapter, the open problems in fuzzing REST APIs are discussed and addressed with possible solutions.

On a high level, the approach is based on captured traffic traces between a user and the target REST API. These traces are converted into a base OpenAPI spec using existing software [MP2S]. Through a novel software implemented as part of this thesis, it then statically analyzes the traffic traces, e.g. for producer-consumer dependencies. Using that collected information and the base OpenAPI spec, the fuzzing process is started. Information from both static analysis and fuzzing are used to compile an enriched OpenAPI spec.

In general, the enriched spec should list API characteristics the way they are actually implemented and not how they are meant to be. This implies the enriched spec designates not what requests *should* be sent to the API, but what requests *can* be sent to it while still being accepted. As a consequence, this can lead to conflicts between the official and enriched spec.

Figure 5.1 shows an overview of the general process. In the first step of *traffic collection*, a user generates and captures traffic by using a frontend application connected to the target REST API. These traffic flows are converted into a base OpenAPI spec (see Section 5.1), and also used in the later steps. After creation of the base spec, *static traffic analysis* inspects the traffic traces with help of the base spec (see Section 5.2). In the third step, *fuzzing*, the API is dynamically probed with a larger number of HTTP requests, based on the base OpenAPI spec (see Section 5.3). For initialization of those requests, the captured requests from the traffic traces are consulted.

For this thesis, three different types of OpenAPI specs are defined:

- **Official OpenAPI spec**: This refers to the officially published OpenAPI spec provided by the developer of the REST API.
- **Base OpenAPI spec**: The base spec means a reverse-engineered OpenAPI spec for the target API, which is used as a starting point by the thesis' fuzzer. It does not contain any information gained through fuzzing.
- **Enriched OpenAPI spec**: The enriched spec refers to the reverse-engineered spec produced by the new fuzzing software from this thesis. It includes all information discovered through the thesis' approach, such as API dependencies (see Section 5.2), detailed parameter patterns (see Section 5.3.1), and authentication requirements (see Section 5.3.2).

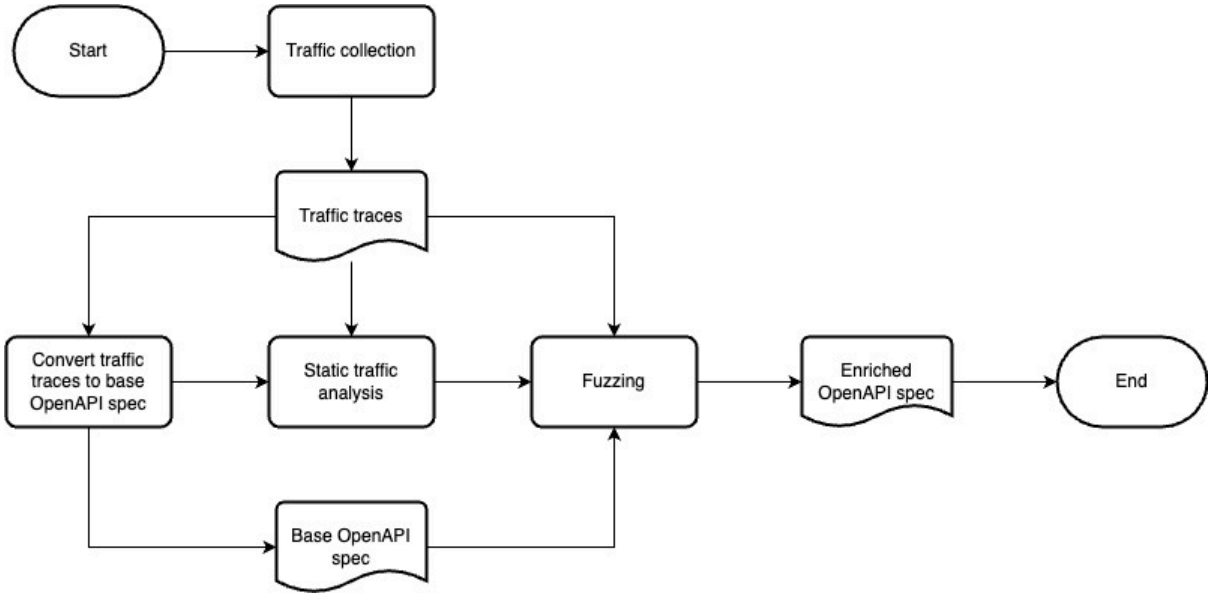

Figure 5.1: Methodology overview showing the main steps: Base OpenAPI spec creation, static traffic analysis, and fuzzing

5.1 Base OpenAPI spec creation

The first step for the approach is the creation of the base OpenAPI spec. As its foundation, traffic traces of captured API communication are used. This is further explained in the following.

5.1.1 Traffic collection

Traffic traces are the foundation for the thesis' approach. There are multiple methods for collecting API traffic, e.g. routing traffic via a proxy, or traffic sniffing on network equipment like routers, switches or wireless access points. For this thesis, traffic is stored using web browser DevTools. With this method, traffic flows are readily available to store since they are directly sent and received by the web browser itself.

Traffic is captured from communication between an official API client and the REST API server. Maximizing coverage of the API's functionality is important in order to maximize the traffic flows' information content and usefulness. For apps and web applications, this requires the user to browse the application as much as possible in order to trigger all available functionality, which directly translates to the according REST API requests.

5.1.2 Converting traffic flows to OpenAPI specs

To create the base OpenAPI spec based on the collected traffic flows, existing software [MP2S] is used. However, it only uses information available in the captured traffic. If some details were not captured, they will be missing in the resulting OpenAPI spec since these tools only perform static analysis. Fuzzing can help with dynamically acquiring new information, which is discussed later in this chapter.

5.2 Static traffic analysis

This section covers the process of static analysis of the collected traffic flows and the base OpenAPI spec. These methods do not send or generate any new API requests, hence do not require a network connection to the target API and can be performed offline. Currently, this step is used exclusively for analyzing producer-consumer dependencies implemented by the REST API.

5.2.1 Producer-consumer dependencies

REST APIs work with resource objects, and its operations include create, read, update and delete, so-called *CRUD*. These typical CRUD operations lead to inevitable dependencies, since a resource has to be created first before it can be read, updated or deleted. CRUD is a simple and clear example for such dependencies, but due to the complexity of REST APIs, there are often nested dependencies between API endpoints as well, which exist in deeper business logic.

Consider Figure 5.2 for an example: The endpoint *GET /api/{team_id}/{user_id}* needs to be called, but requires two resources, a team and a user, with their corresponding IDs. The user resource can be created via another endpoint, *POST /api/user*, while the team resource can be created via *POST /api/{org_id}/team*. It can be seen that the team's producer endpoint requires another dependency, an organization, which is produced via yet another endpoint, *POST /api/organization*. So it is clear that calling the original GET endpoint requires calling multiple other endpoints first, in the correct order, leading to a specific dependency graph of API endpoints.

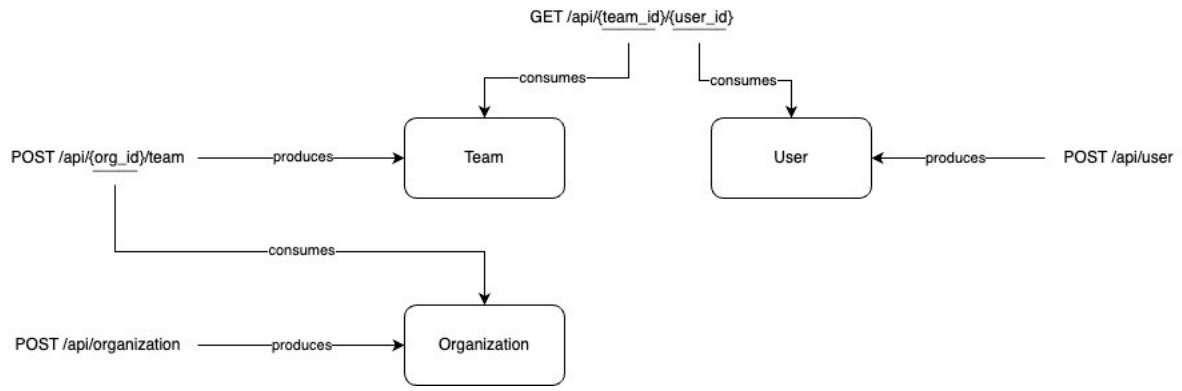

Figure 5.2: Example of a dependency graph

For fuzzers to successfully probe as much API functionality as possible and to reach deeper service states, those dependency graphs must be resolved. There are REST fuzzing tools that infer dependencies between API endpoints based on the given OpenAPI spec [AGP19]. For that, they look for a combination of endpoint and variable names and search for endpoints where those variables are used again [REAN]. For example, let us consider an endpoint *POST /user* returning a parameter *id* in its response and another endpoint *GET /user/{userId}*. Based on the first endpoint's name containing *user* and returning *id* and the second endpoint having a variable consisting of the concatenation of the two other variable names, i.e. *userId*, they infer the dependency between those two endpoints. Similar constellations are also supported and are based on the same approach.

While that approach certainly works for complete, well-documented OpenAPI specs, it is not feasible for reverse-engineered specs, which may not represent all endpoint and variable names correctly. Additionally, that approach generally only works for REST APIs implementing a consistent naming of endpoints and variables, which adheres to that approach specifically.

Since it is important to understand dependencies between endpoints in order to reach deeper service states through fuzzing, the approach established during this thesis tries to make dependency inference more feasible for a broader range of REST APIs. Based on the seeded traffic traces, the API communication is analyzed for possible variables. These variables are then searched for in all other endpoints, and if multiple endpoints use the same variable, a dependency between them is noted. The algorithm for dependency inference for a specific consumer is available as pseudocode in Algorithm 1.

5.2.1.1 Finding eligible producers

To go into more detail, dependencies between one consumer and multiple producer endpoints can be found. In context of REST APIs, a producer usually creates resource objects which are identified by a unique resource ID. Those resource IDs are used to find connections between consumers and producers.

Three assumptions are made for this inference process:

Assumption 5.2.1

Resource IDs have a certain amount of entropy, i.e. are unique, or random enough to stand out from other variable values.

Assumption 5.2.2

Producer endpoints respond with HTTP status 201 Created, indicating successful creation of a resource [RFC9110, chapter 15.3.2].

Assumption 5.2.3

An endpoint P can only be a producer of consumer C if the producer's response at time T_P has been received before the first request to C was sent at time T_C, i.e. $T_P < T_C$ must hold true. In other words, for a consumer C, only endpoints that responses were received from prior to contacting C can possibly be producers for C.

Because REST APIs can have arbitrary complexity, these assumptions allow to aim for a specific set of API characteristics which many of today's REST APIs should implement. Of course, out of the abundance of REST APIs available, there are certainly APIs for which those assumptions will not hold, however, they should work for many, if not most of them.

The first step of the algorithm is to find candidate variables CV for a potential consumer endpoint C, i.e. variables that look like they could possibly be resource IDs, returned earlier by a producer endpoint. This problem is called *parameter classification*. The goal is to find parameter values (or variables) which can be classified as resource IDs. Variables classified as potential resource IDs then become candidate variables. Currently, only request and response body parameters are analyzed. For that, the captured HTTP requests for this specific endpoint are consulted. All values in the JSON request body are analyzed by calculating their entropy. The values above a configurable entropy boundary are considered candidate variables. The same logic is applied

to all path segments of the captured endpoint data, i.e. all strings in the HTTP path that are separated by a slash.

Regarding Assumption 5.2.3, let us consider a short example to see why this assumption is needed. For some consumer, there could be responses of eligible producers EP which may also contain CV (in addition to other created resource IDs), but were sent after T_C. However, since CV was already known to C at T_C, CV could not be produced by EP. Endpoints that do not fulfill this constraint are filtered out and not deemed eligible producers for C from now on.

For each candidate variable CV, the value is looked up in all eligible producers' (i.e. except the ones filtered out) JSON response bodies contained in the traffic trace. If that value was returned in a producer's response, a match is found and that endpoint is added as producer P for endpoint C. A consumer C can only have exactly one producer P for a specific variable CV. It is the first (by response timestamp) eligible producer found whose response contains CV. That means, if a producer endpoint was already found for C with CV, all other eligible producers are skipped for that endpoint. This is because the first producer that returned CV must be the actual producer, while other producers after that may reference that resource in their response (and therefore may also contain CV). After an according producer is found, that also means it is validated that the candidate variable CV is an actual variable V.

5.2.1.2 Analyzing producer-consumer connection

Now that the basic connection between C and P is discovered, further processing finds the JSON property from producer P's response body that contains variable V. For this purpose, the arbitrarily complex JSON object is parsed and searched for V recursively. If V is contained in consumer C's request body, the algorithm also gets that JSON property accordingly. Otherwise, V is contained in C's HTTP path. A mapping between the two endpoints C and P and their respective locations of V (either the found JSON property or the HTTP path) is stored.

All API endpoints can be a consumer, while a producer is constrained by its HTTP status code 201. An endpoint can be a consumer and a producer at the same time, allowing to capture deeper service dependencies. The producer-consumer dependencies are modeled in a many-to-many relationship: A consumer can have n producers, and a producer can have m consumers. Each producer can have an arbitrary number of variables which a consumer depends on.

Algorithm 1: Static traffic analysis - Dependency inference for consumer C

Require: $C \neq \emptyset \wedge E > 0 \wedge R = \{(r_i, t_i) \mid i = 1, \ldots, n \wedge t_1 < t_2 < \ldots < t_n\}$

$P, CV, V, EP \leftarrow \emptyset$

$values \leftarrow$ GetJsonValues(C.request.body) $\cup$ C.request.path.segments
$CV \leftarrow$ values.Where(v $\rightarrow$ Entropy(v) $\geq$ E)
// R holds all captured HTTP requests, ordered by timestamp
$EP \leftarrow$ R.Where(p $\rightarrow$ p.response.status = 201 & p.response.timestamp < C.request.timestamp)

for ep $\in$ EP **do**
    $values \leftarrow$ GetJsonValues(ep.response.body)
    for cv $\in$ CV **do**
        $V \leftarrow$ values.FindMatch(cv)
        if V $\neq \emptyset$ & **not** P.Any(p $\rightarrow$ p.response.body.FindMatch(cv)) **then**
            $L \leftarrow$ FindLocation(ep, V) ▷ Get exact locations of variable in *ep* and *C*
            P.add([ep, V, L]) ▷ Store found producer together with variable and locations
        end if
    end for
end for
return P

procedure FINDMATCH(this object, value)
    for v $\in$ GetJsonValues(object) **do**
        if v = value **then return** v
        end if
    end for
return $\emptyset$
end procedure

procedure FINDLOCATION(producer, variable)
    $L \leftarrow \emptyset$
    $L['p'] \leftarrow$ ['body', variable.jsonKey]
    $cVar \leftarrow$ C.request.body.FindMatch(variable)
    if cVar $\neq \emptyset$ **then**
        $L['c'] \leftarrow$ ['body', cVar.jsonKey]
    else
        $L['c'] \leftarrow$ ['path', PathPosition(C.request.path, variable)]
    end if
return L
end procedure

Example Let us consider a simple example of a producer-consumer dependency that can be found by this algorithm.

```
// Endpoint A, message 1 (A1)
POST /user
{
    "name": "John",
    "age": 42,
}

// Endpoint A, message 2 (A2)
HTTP/2 201 Created
{
    "user": {
        "id": "477ee3b7-66e8-487e-aea8-90abbf3b703b",
        "name": "John",
        "age": 42,
    }
}

// Endpoint B, message 1 (B1)
PATCH /user/477ee3b7-66e8-487e-aea8-90abbf3b703b/status
{
    "status": "online",
}

// Endpoint B, message 2 (B2)
HTTP/2 200 OK
[...]
```

Listing 5.1: Example sequence of requests with dependencies

Listing 5.1 shows multiple requests and responses to and from a REST API. Request A1 creates a user object. Response A2 returns the newly created user with its ID, starting with *477e*. Request B1 requires a valid user ID in its path, together with a status to set for the user object in the request body.

The algorithm starts with analyzing endpoint *A*, specifically the request (A1). No candidate variables are identified, hence the algorithm will stop here and continue with the next endpoint. Once the algorithm analyzes endpoint *B*'s request message (B1), it first identifies *477ee3b7-66e8-487e-aea8-90abbf3b703b* as a candidate variable *CV*. Then, it looks for *CV* in producer endpoints. Since endpoint *A* is the only endpoint in this example that returns HTTP 201 (A2), only that is deemed a producer. The producer's captured response (A2) is scanned for *CV*, and it is found successfully. Because a basic dependency can now be considered true, *CV* becomes an actual variable dependency *V*. Next, the producer's JSON response is parsed and the property containing *V* is returned. In this example, that property can be identified by its path *user.id* in the JSON object. As a result, the algorithm found out that endpoint *A* is a producer for endpoint *B*, and that the JSON property *user.id* in *A*'s response is used in *B*'s HTTP path.

5.2.2 Request generation

One goal of inferring producer-consumer dependencies is to be able to generate new consumer requests through understanding a consumer's dependencies. While some API requests contained in the seeded traffic traces are simply replayable as-is, others may only be sent once per resource, and additional invocations are forbidden by the API. Those requests require a fresh resource and ID in order to fuzz the endpoint in question. Creating new resources often needs some other

resource dependencies, which is why it is important to understand endpoint dependencies in order to solve the dependency graph.

This ability is interesting for both explorative and exploitative fuzzing. Creating new requests based on learnt dependencies is practical for exploitative fuzzing because it enables to fuzz more API endpoints, which might not be fuzzable successfully without providing valid dependencies. This is useful for explorative fuzzing as well, since it allows to potentially discover more API characteristics through a greater API coverage. Specifically, for this thesis' methodology, dependency-based request creation is used for path variable detection (see Section 5.3.3).

Now let us take a look at the algorithm. New requests can be generated for a specific consumer endpoint *C*. *C*'s producer dependencies are resolved and the algorithm is run recursively for each producer until a producer has no other producers in its dependencies. When that is the case, i.e. the first producer *P* in the dependency graph is found, its originally captured request is replayed.

If the replayed request returns a successful response (HTTP 201), it indicates that a new resource has been created which can be used for *P*'s depending consumer. If the request cannot be replayed, i.e. a negative response is returned by the REST API, simple fuzzing is applied to the request by mutating the JSON request body values. Each string value is concatenated with a short, random ASCII string, consisting of characters *A-Z*, *a-z* and *0-9*. For example, if *P* requires a resource name in its request body, the request may be rejected if a resource with that name already exists. Therefore, providing a slightly changed name can already be enough to get a successful response. Other value types, such as booleans or integers, are exempt from fuzzing since those values are likely required to stay the same, e.g. mutating the value *true* could yield an invalid value which could cause the request to fail.

The resulting producer responses from *C*'s directly depending producers are then set in *C*'s new request body, according to the inferred dependencies. Finally, the newly crafted request for *C* is sent and the response is observed.

5.2.3 Dependencies in OpenAPI specifications

The inferred producer-consumer dependencies are stored in the enriched OpenAPI spec. Other RESTful fuzzers can then make use of the knowledge about endpoint dependencies, enabling more efficient planning and execution of fuzzing.

For modeling dependencies and relationships between endpoints, OpenAPI offers support via *links* [OAL24]. Using links, an endpoint's response values can be connected to another endpoint's request parameters. This allows to represent a common pattern of REST API's endpoint relationships, which is also found by this thesis' dependency inference algorithm.

Figure 5.3 shows an overview of the connections between different OpenAPI objects using links. First, an endpoint (or operation) that is referenced needs a unique *operation ID*. That ID can be used to uniquely identify another endpoint in links' references. Alternatively, references can also use *operation refs*, which consist of the target endpoint's details, like its path and HTTP verb.

A link object is then added to an endpoint's specific response. The object can contain multiple links, and each one requires a name which is unique inside the response object. In Figure 5.3, the link name is *GetUserByUserId*.

The link contains parameters which are returned in the response and are used by the referenced endpoint. In this example, the parameter's name is *userId*. This directly links to the target endpoint's parameter name, i.e. *userId*, which is a path variable in this case. The link's response value is represented as a JSON Pointer [RFC6901]. Here, the value is contained in the response body, in property *id*. Summarized, the response value in property *id* is linked to the request parameter *userId* in the endpoint identified by its operation ID *getUser*, that is *GET /user/userId*.

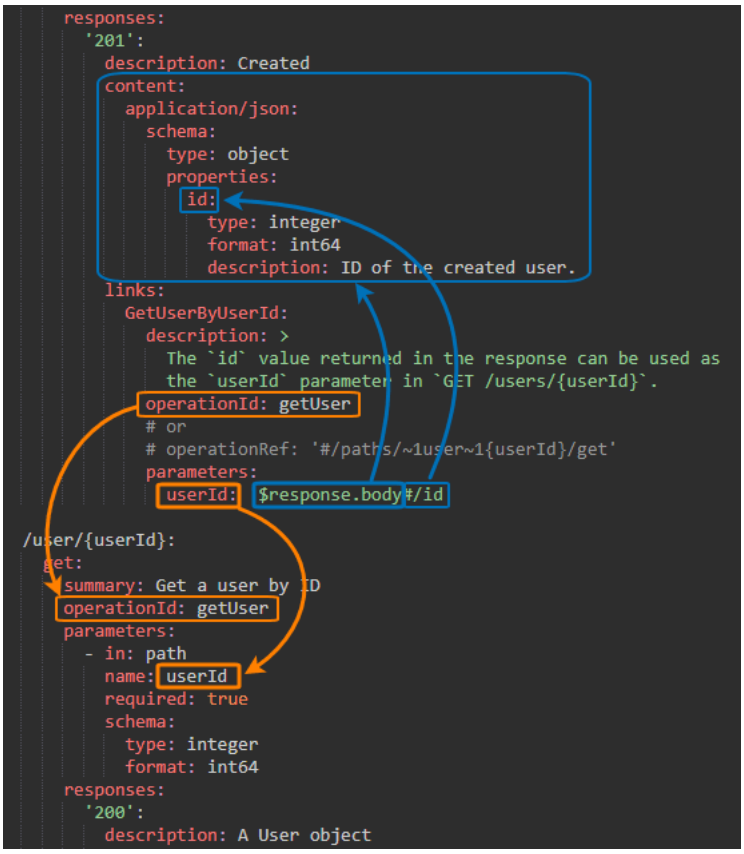

Figure 5.3: OpenAPI link example with relationship pointers, from [OAL24]

A concrete example of how the enriched OpenAPI spec section could look like as produced by the algorithm is included in Figure 5.2. Here, the *POST /users* endpoint returns an *id* property in its response body. The operation IDs are named after the endpoint's HTTP verb and their path, making them globally unique. The link names are essentially the same as the operation IDs, but additionally have the prefix *link-* to clearly separate them from operation IDs. Note that the consuming endpoint contains a path variable (*variable1*); the approach for finding those types of variables is discussed in Section 5.3.3.

```yaml
/users:
  post:
    responses:
      '201':
        content:
          application/json:
            schema:
              type: object
              properties:
                id:
                  type: string
        links:
          link-GET-users-variable1:
            operationId: GET-users-variable1
            parameters:
              id: $response.body#/id
/users/{variable1}:
  get:
    operationId: GET-users-variable1
    parameters:
      - name: variable1
        in: path
        schema:
          type: string
```

Listing 5.2: OpenAPI example of link between producer and consumer as produced by the thesis' algorithm

5.2.4 Modeling request body parameter dependencies

Previously mentioned producer-consumer dependencies use OpenAPI links, which again reference to OpenAPI parameter objects. Since those only support parameters contained in headers, queries, paths or cookies [OAS24], request body parameters cannot be represented using standard OpenAPI spec. However, for cases that are not covered by the OpenAPI specification, OpenAPI allows definition of custom objects using OpenAPI specification extensions. Spec extensions can be added to any OpenAPI spec using field names starting with *x-* which can map to arbitrarily-typed objects.

Conveniently, Microsoft's RESTler fuzzer project already defines a custom OpenAPI spec extension for request body parameter dependencies, called *annotations* [REAN]. Using RESTler annotations enables direct support for producer-consumer dependencies inferred using the thesis' algorithm in RESTler, and fulfills the required functionality. Therefore, RESTler annotations are adopted for this thesis.

RESTler annotations are only used for modeling request body parameter dependencies, and all other endpoint dependencies are represented using standard OpenAPI links. While links are defined in producer endpoints and point to their consuming endpoints, annotations are defined the other way around. That means, they are defined in consumers and reference dependent producer variables.

For better clarity, consider the example in Listing 5.3. RESTler annotations use the field name *x-restler-annotations* and consist of an array of dependency objects. Those objects define the dependent producer parameter using the producer's HTTP verb/method (*producer_method*), the

producer name (*producer_endpoint*), the producer's dependent variable (*producer_resource_name*) and the consumer's request parameter name (*consumer_param*). Annotations support more fields, but those four are sufficient for this use case. In this example, it can be seen that the endpoint *POST /users/me* references the *user_id* response variable from producer *POST /users*, which needs to be put into the consumer's request body parameter named *id*.

```
/users:
  post:
    responses:
      '201':
        content:
          application/json:
            schema:
              type: object
              properties:
                user_id:
                  type: string
/users/me:
    post:
      requestBody:
        content:
          application/json:
            schema:
              type: object
              properties:
                id:
                type: string
    x-restler-annotations:
      - producer_method: POST
        producer_endpoint: /users
        producer_resource_name: user_id
        consumer_param: id
```

Listing 5.3: OpenAPI example using RESTler annotations for request body parameter dependency modeling

5.3 Fuzzing-based reverse engineering

This section deals with fuzzing REST APIs in order to discover new API characteristics. In contrast to static traffic analysis, the methods in this section require an active network connection to the target API since new API requests are sent to it, with their responses being analyzed.

As a starting point, the fuzzing process uses the created base OpenAPI spec as well as the collected traffic flows. Since REST APIs can be considered Remote Black-boxes (see Section 4.2.2.1), the proposed methods exclusively make use of Remote Black-box fuzzing methods. This assumption broadens the spectrum of target applications for which the proposed methods can be applied, and those types of REST services may be the most interesting ones to reverse-engineer. All methods presented in this section are included in the *fuzzing* step as shown in Figure 5.1.

5.3.1 Grammar inference

Grammar inference is used to generalize the known, accepted values for a specific API parameter into a regular expression. This method tries to address the type inference problem [Gau+21] [DAT24]. Since OpenAPI specs often tend to only include a general parameter type definition, e.g. string or integer, exploitative fuzzers cannot efficiently apply type-specific exploitation logic. For example, if it would be known that a certain parameter accepts dates like *2024-12-01*, date-specific logic could be applied, e.g. exploiting year overflows [DAT24]. Moreover, knowledge of accepted values may drastically decrease the time and resources required for later fuzzing since the search space for string-typed parameters is arbitrarily complex. As the proposed solution, a language learning algorithm is run against the API to infer a more detailed type definition.

GLADE [Bas+17] is chosen as the language learning algorithm. The algorithm is more thoroughly explained in Section 4.2.2.5, but the most important points are quickly recapped. For the algorithm to work, two prerequisites are needed: (i) A set of values that are known to be part of the grammar (seed), and (ii) an oracle that can be used for membership queries of arbitrary values. GLADE has a reasonable computational complexity of $\mathcal{O}(n^4)$ (where n is the length of the seed input). While this might look like a huge complexity at first sight, it is actually the most performant language learning algorithm currently available. Additionally, as it is not based on machine learning, unlike other approaches, it does not require upfront training of ML models and can quickly be set up and applied to real-world problems.

An *oracle machine* is a machine that uses an oracle that can magically solve the decision problem for some language $O \subseteq \{0,1\}^*$ [AB09, p. 70], i.e. a problem that can be answered with either *yes* or *no*. In this case, the oracle must solve the problem of whether a certain value is part of the language of a specific parameter as accepted by the REST API. The targeted API can be seen as an oracle, since it either responds with a *positive* (HTTP status codes 200-299) or *negative* (all other status codes) answer, depending on the chosen input. Additionally, the oracle, i.e. the REST API, can be arbitrarily queried (by the oracle machine). Thus, the API serves as the oracle as required by GLADE.

Since the computational complexity is dependent upon the seed values' lengths, the worst-case runtime can be controlled through careful selection of appropriate seed values. This is important because the algorithm is run for every fuzzable parameter, and the fuzzer's runtime would otherwise increase exponentially with every parameter. Moreover, while the seed should not grow too large, it should also contain relevant values that can be used for meaningful mutations. For example, let us consider the seed value *true*, while the oracle accepts the values *true* and *false* (i.e. those are the only two values in the grammar). If this is the only seed value, GLADE generates a number of mutations such as *truetrue*, *tru* and *tr* and sends those to the oracle. The resulting grammar is *((true))*. While this represents the accepted value *true*, it does not make any statements about the other accepted value *false*. This is because the seed captures too few semantics for GLADE to gather more information about other values in the grammar, which is used for generating mutations. Therefore, it is important to use a diverse set of values as the seed, representing interesting semantics and a broader range of values in the grammar.

The algorithm is executed for every API parameter contained in the base OpenAPI spec. The parameter's type from the base spec is ignored since it may only reflect the type that has been sent to the API at the time of capturing traffic from an official API client, however, it is possible that more types are actually accepted. For example, if the base spec's parameter's type is

number, it is not treated as an integer-only parameter. Instead, it is assumed that other types, like non-numeric strings, could be valid inputs as well. Therefore, each parameter is fuzzed through the thesis' fuzzer using a simple approach. A static list of values is contained in the software which is used for fuzzing all parameters. Those values try to represent a variety of different data types and edge cases, while keeping the list's size as small as possible (to optimize runtime complexity). If the API returns a positive message using the fuzzed parameter value, it is saved as a positive mutation. Parameter values from the traffic traces are always treated as positive mutations as those stem directly from the originally captured requests sent via an official API client. The GLADE language learning algorithm is then started using all positive mutations as seeds. When GLADE finishes execution for the particular parameter, the learnt regular expression is written to the parameter's *pattern* object [OAD24] in the enriched spec, which is a specified OpenAPI field for defining valid input strings in form of regular expressions. Consider Listing 5.4 for an example of how the pattern looks like in an OpenAPI spec. In this example, the query parameter *role* only accepts strings containing lowercase characters (a-z).

```
paths:
  /users/:
    get:
      parameters:
        - name: role
          in: query
          schema:
            pattern: "[a-z]"
```

Listing 5.4: OpenAPI example of parameter with pattern

5.3.2 Authentication requirements

Many REST APIs require some form of authentication for a set of their endpoints. For example, an API consumer, like a user, could authenticate themselves through the use of an authentication token, so the API knows who it is communicating with. Based on the authenticated user, further authorization constraints can be enforced by the server, e.g. permitted actions, role assignment, visibility of certain information etc.

When fuzzing REST APIs, it is advantageous to have knowledge about the API's authentication requirements. With that, fuzz tests against the authentication logic itself can be executed. This may be the most interesting use case for exploitative fuzzing using the documented authentication requirements. For example, authentication vulnerabilities such as improper token validation or acceptance of expired tokens could be uncovered. This unlocks a new class of bugs which would not be possible without awareness for authentication requirements.

There are also APIs that do not explicitly require authentication, but instead apply different limits based on the authentication state. Fuzzers can consciously test both authenticated and unauthenticated states and analyze differences between them. For example, some APIs implement rate limiting or other resource limitations (i.e. quotas) for unauthenticated users. These limits can also exist for authenticated users but may vary. Using the authentication requirements, it could be tested whether these limits are properly enforced or if vulnerabilities exist in connection to the authentication state (e.g. by analyzing the family of *RateLimit* HTTP headers [RLH]).

Additionally, knowing which endpoints require authentication can make fuzzing more efficient, e.g. by omitting endpoints for which no authentication credentials are provided. This can help prevent exhausting rate limits or triggering lockout rules during fuzzing. Due to these advantages, the authentication requirements are fuzzed by the newly implemented fuzzer and reflected in the enriched OpenAPI spec.

For HTTP and REST APIs, there are multiple authentication methods possible. Bearer authentication is commonly used, with Basic and cookie authentication being other choices. For enriching reverse-engineered OpenAPI specs, the goal is to detect those authentication methods as well as custom variations of them. While there are more authentication methods, like API keys, OAuth2 or OpenID Connect, these seem to be most prevalent in REST APIs and are therefore focused. Bearer, Basic and cookie authentication can be broken down into two generalized classes: *Header-based* and *cookie(-based)* authentication. Their workings as well as the approach for finding authentication information are explained in the following.

5.3.2.1 Header-based authentication

Header-based authentication uses HTTP headers for communicating authentication information to the server. Most prominently, the standardized *Authorization* header [RFC9110, chapter 11.6.2] is used for providing authentication credentials.

Multiple authentication schemes can be used via the Authorization header, e.g. the mentioned Basic and Bearer schemes. For example, a client utilizing the Bearer scheme [RFC6750] uses an access token generated by an authorization server in order to access protected resources. Sole possession of this token grants authorization. A client transmits its request with the Authorization header in the form of *Authorization: Bearer $ACCESS-TOKEN*. The Bearer scheme also supports transmission of access tokens as a form-encoded body parameter or via an URI query parameter, but for REST APIs, the Authorization header is used most often.

The access token can have arbitrary form, e.g. a randomly-generated string. Modern APIs often use the cryptographically-signed JWTs (JSON Web Tokens), providing information in a secure and stateless manner, such as the user's identity (e.g. user ID) and their claims (roles, permissions etc.).

Algorithm The algorithm now tries to find out whether a certain endpoint requires authentication. For that, one precondition must be met: The current response code must not indicate a failure due to unauthenticated access. This is checked by replaying the unmodified, captured request and looking at the response status code, which must be unequal to *401 Unauthorized* [RFC9110, chapter 15.5.2] and *403 Forbidden* [RFC9110, chapter 15.5.4]. Otherwise, the algorithm could not differentiate between failures due to intentional request modification, or failures which were present from the beginning (e.g. because the captured access token is expired). The precondition must be fulfilled for all authentication-related fuzzing methods presented in this section.

Next, the algorithm checks whether a captured HTTP request from the seeded requests contains the Authorization header. If that is the case, it is likely that authentication for that endpoint is needed. To verify, a new request *R* is sent. *R* is an exact copy of the original request, but with the Authorization header removed. If the API responds to *R* with either HTTP status code 401 or 403 the endpoint indeed requires authentication.

The status code 401 is checked as it indicates that the client lacks valid authentication credentials. This should be the expected status generally used for this case. However, it is also possible for the API to utilize the status code 403. That is defined as "the server understood the request but refuses to fulfill it". The according RFC also mentions authentication credentials in connection with this status code (if provided, they are deemed invalid), but this code is of course not only limited for authentication-specific issues. In summary, both status codes are possible when authentication is required, hence both are checked.

Authentication scheme Furthermore, it is analyzed whether the API uses the Basic or Bearer scheme. This can simply be checked by looking at the first few characters contained in the header's value. If the value starts with *Basic*, it uses Basic authentication. If it starts with *Bearer*, it uses the Bearer authentication scheme. If none of those match, a custom authentication scheme (or other more uncommon ones) might be used. All information gathered by the algorithm is stored as an internal representation and used later on for OpenAPI spec generation.

In addition to the standardized Authorization header, REST APIs might sometimes also use custom authentication schemes using other headers. To accommodate for that, if the Authorization header is not present in an endpoint's captured request, all headers are checked for authentication information. The captured request is again copied to a new request R, and the algorithm removes each header one by one. R is then sent to the API with just one header being absent. That means if the original request contains n headers, there are at most n fuzzed requests for that endpoint. Analog to the above approach, the response status is checked, and if it is either 401 or 403, the header that has just been removed is required for authentication. Figure 5.4 shows a visual overview of the fuzzing algorithm.

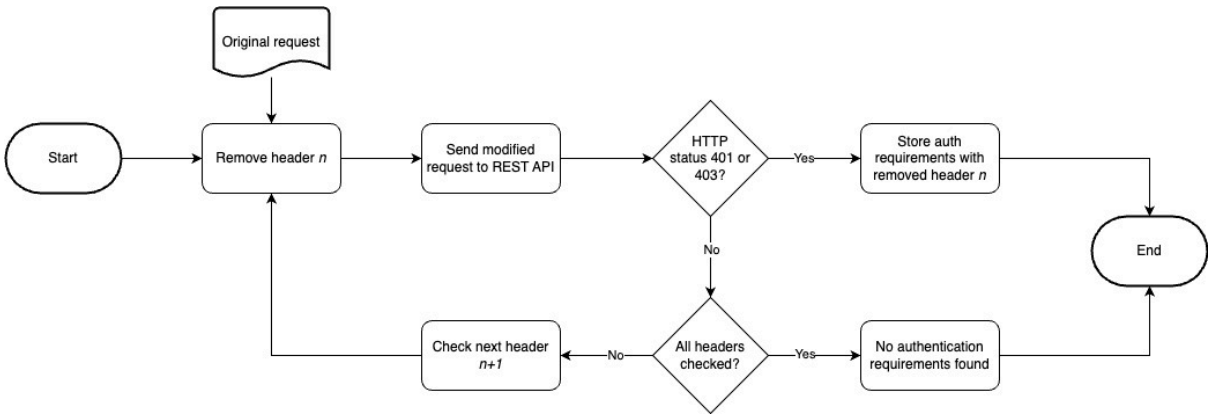

Figure 5.4: Authentication fuzzing logic

Formal specification Based on the data gathered through this fuzzing process, authentication information is translated to OpenAPI objects. For this purpose, the OpenAPI specification [OAS24] defines two relevant objects: The *security scheme object* and the *security requirement object*. The security scheme object allows definition of the REST API's general authentication methods. This definition is globally available to other objects in the spec. Each endpoint can then reference a top-level security scheme object in its own security requirement object.

Let us consider a concrete example: Listing 5.5 shows a minimal example of a global security scheme object defined under the *components* key, and an endpoint-specific security requirement

object. The security scheme object's *type* is *http*; other options include *apiKey*, *openIdConnect* or *mutualTLS*. The *scheme* key defines the *Bearer* scheme (this key is only applicable to the *http* type).

Under the *paths* key, the *GET /users* endpoint defines its security requirements via the *security* key, which holds a security requirement object. That references the global security scheme object by its name *BearerAuth*, concluding that this endpoint requires Bearer authentication. In this case, the value for that key is an empty array, and may optionally contain roles or, for OAuth-based schemes, scopes that are needed for this endpoint.

```
components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: Bearer
paths:
  /users:
    get:
      security:
        - BearerAuth: [ ]
```

Listing 5.5: Example of security scheme and security requirement objects

In case that a custom or unknown authentication scheme is used (i.e. neither Basic nor Bearer), the *scheme* will be represented as *Custom*. Since this value is not defined in the OpenAPI specification and the authentication scheme is not standardized, custom fuzzing logic is required for exploitative fuzzing.

For custom authentication schemes that do not use the Authorization header, another form of serialization to OpenAPI spec is required. In OpenAPI, custom headers are treated as API keys and can be represented as type *apiKey* with location *header* and *name* being the name of the custom header. As an example for how security schemes for such cases are serialized, refer to Listing 5.6.

```
components:
  securitySchemes:
    TokenHeader:
      type: apiKey
      in: header
      name: X-Token
```

Listing 5.6: Example of OpenAPI security scheme for custom header

5.3.2.2 Cookie authentication

Similar to header-based authentication, HTTP cookies can also be used for transferring credentials to the server. Since cookies are also key-value pairs, just like headers, which are sent via the *Cookie* header, the approach for finding cookie-based authentication data is very similar to the header-based approach for custom authentication schemes (not using the Authorization header).

Indeed, the algorithm removes each cookie one by one and generates new requests with exactly one cookie missing, compared to the originally captured request. When the API responds

with either HTTP status code 401 or 403, the cookie that was last removed is required for authentication.

The OpenAPI specification also supports cookie authentication. In contrast to the security scheme object used for HTTP-based schemes, cookie authentication in OpenAPI specs is defined as an *apiKey* scheme with its location being *cookie*. The *name* key then defines the cookie's name that is used. Analog to header-based authentication, endpoints then reference the security scheme object. For better understanding, consider Listing 5.7.

```
components:
  securitySchemes:
    CookieAuth:
      type: apiKey
      in: cookie
      name: AuthCookie
paths:
  /users:
    get:
      security:
        - CookieAuth: [ ]
```

Listing 5.7: Example of cookie authentication in OpenAPI

5.3.3 Path variable detection

One problem that is apparent in current reverse-engineered API schemas is that HTTP paths, also called API endpoints in combination with their HTTP verb, contain context-specific values. For example, let us consider an endpoint *GET /api/users/abc123*, i.e. HTTP GET requests to the path */api/users/abc123*. While capturing real API traffic, the user may have requested this endpoint multiple times and also received valid responses. However, it is apparent that the last part of the endpoint URL, *abc123*, is likely a variable and not a fixed part of the endpoint like the preceding parts */api/users/*. Since the user only requested that endpoint with this specific value filled in, current software for traffic-to-OpenAPI conversion only sees that data and converts it to an API endpoint with the fixed value instead of marking that part as a variable.

This becomes an issue for fuzzing. The fuzzer is not aware that specific parts of an endpoint are actually variables that could be fuzzed, and rather treats those parts as hardcoded path segments, leaving potential variables completely unexplored. Therefore, those variable path segments should be detected and properly marked as variables in the resulting API schema. The approach for that is explained in the following.

Idea The first step is to identify the kind of variables that are included in API endpoints. As seen in Section 3.1, REST defines multiple constraints, one of those being the Uniform Interface. That constraint is again sub-divided into multiple properties.

The relevant one for now is the sub-property *Resources and Resource Identifiers* [Fie00, pp. 88 sqq.]. It says a *resource*, which is a key abstraction in REST, can be any information that can be named, e.g. a document, a person etc. Resources are identified by *resource identifiers*. An example representation from the modern Web for such an identifier is a URL.

Authors or developers can freely choose a resource identifier, based on their vision what is best fitting.

In other words, resources used in REST APIs are generally referred to by their resource identifier, and can be included in URLs. Modern APIs usually use uniquely-identifiable information as identifiers, e.g. UUIDs or otherwise randomly-generated strings (in contrast to, e.g., monotonically-increasing integers). These are indeed most often included in URLs, i.e. API endpoints. As a result, the following assumption is constructed:

Assumption 5.3.1

Endpoint variables contain resource identifiers.

Based on this new assumption and Assumption 5.2.1 (resource IDs have a certain amount of entropy), a method is needed to distinguish fixed endpoint segments from variable ones. This is essentially the same parameter classification problem as mentioned in Section 5.2.1.1. In this case, the parameters are potentially included in the HTTP path. The goal is to find path segments that likely contain resource IDs.

Option 1: Word dictionaries One method is to use word dictionaries for comparing the words found in API endpoints with words used in human languages. That allows for filtering out, e.g., non-English words, effectively returning unique resource identifiers. For example, considering the endpoint from above (*GET /api/users/abc123*), the URL can be split into its segments, resulting in *api*, *users* and *abc123*. Each segment is then classified as either *English word* or *Non-English word*. The optimal result is that the words *api* and *users* are classified as English words, and *abc123* as Non-English. Therefore, it can be assumed that the segment *abc123* is likely a variable since it is not contained in the English dictionary.

However, this approach can become problematic when APIs implement endpoint names with non-standard words. For example, while "API" (or *api*) is a familiar term for computer scientists, it is actually not contained in the English dictionary (it is also not a proper word but rather an acronym). That means that *api* would wrongly be classified as Non-English word, which would not help in separating fixed and variable endpoint segments, since, in this example, *api* is a fixed part of the API endpoint.

While cases like this could be handled manually, there probably exist a larger number of such issues with other non-standard or technical terms which are not contained in any dictionary, but used as REST API endpoint names. Additionally, differentiating between singular and plural is at least another difficulty to consider.

Option 2: Entropy measurements Since it is assumed that variables contained in API endpoints are identifiers based on randomly-generated data, they should have relatively high entropy. Entropy measurements of endpoint segments allow for classification as resource IDs based on their randomness, or uncertainty. Like in parameter classification in Section 5.2.1.1, variables that are classified as potential resource IDs are called candidate variables. While most English words have relatively low entropy, the discussed identifiers should yield much higher entropy values. That difference in entropy enables to set a boundary for proper classification.

As part of this thesis, it is measured where that boundary should optimally be. Based on a large word list consisting of about 370,000 English words [EWL], the average entropy of a word is

2.77 bits. The English word with the highest entropy is *pneumoventriculography*[1] with 3.91 bits.

In contrast, the entropy of a resource identifier used in URLs captured from real traffic between a Mattermost client and server is 4.18 bits. Another resource identifier as used in Matrix contains 4.63 bits of entropy. Table 5.1 shows more tested values and their calculated entropy.

String	Description	Entropy in bits
api	Technical term	1.59
academic	English word	2.5
justarandomstring	Manually crafted string	3.5
123e4567-e89b-12d3-a456-426614174000	Example UUID	3.69
gEqFRSfGXJ83PXtT	Password with 16 alphanumeric characters	3.88
pneumoventriculography	English word with the highest entropy	3.91
tonbj5o4riromc8x95sxp1qzfr	Resource ID from Mattermost	4.18
!YlHySmLwqbHbgSBuOz%3Amatrix.org	Resource ID from Matrix	4.63

Table 5.1: Entropy of different strings

One approach would be to set the boundary at 4 bits, since no English word has 4 bits of entropy. However, while there are certainly many resource identifiers having more than 4 bits of entropy, there are also identifiers with less entropy, which would in turn be excluded from further processing. Instead, the boundary could be lowered in compromise for a potentially higher number of misclassifications.

Wrongly classified variables (i.e. not actual resource IDs, but words from the English dictionary) would then be treated as candidate variables at first, but later processing would find that it is no real variable. Therefore, the entropy boundary is chosen to be 3.5 bits. This should cover most variables included in HTTP paths using randomly-generated strings, while minimizing misclassifications of English words since only 1.4% of them have at least 3.5 bits of entropy.

After candidate variables are extracted from an endpoint's HTTP path, it is validated whether they are actual variables, or just static strings with high entropy. For that, it would be optimal to fuzz the API endpoint with another valid string in place for the candidate variable's value and observe the response. For example, if the endpoint is *GET /api/users/abc123*, and segment *abc123* is classified as a candidate variable, try to call *GET /api/users/xyz789*, where *xyz789* is another variable known to be valid. If the latter API call succeeds, it can be said that the candidate variable and its according path segment are indeed a variable that accepts different values, and this can be written to the enriched OpenAPI spec.

Path variable verification Based on the assumption that variables in HTTP paths are generally resource identifiers, another valid resource identifier is required for validation, which means another resource is needed. Because it is not assumed that traffic traces contain multiple calls to the same API endpoint with multiple different resources, a new resource is created. For this, the inferred producer-consumer dependencies for the checked candidate variable are utilized and the request generation algorithm from Section 5.2.2 is used for resource creation. Optimally, the algorithm produces a fresh resource with a new resource ID. If no new resource can be created (e.g. because of highly-custom API characteristics), the next candidate variable is checked.

1. https://en.wiktionary.org/wiki/pneumoventriculography

This resource ID is then used in place of the identified candidate variable and sent to the API. The response is checked for a successful HTTP status code. If the response is successful, it is now validated that the candidate variable is indeed a variable. Otherwise, validation did not succeed because the candidate variable apparently is not an actual variable, and the next candidate variable is checked. In those cases, HTTP status *404 Not Found* is expected to be returned. This status code could mean that the endpoint is found but the resource ID does not correspond to an existing resource. However, since the resource ID is newly created, it is known that the resource must exist (assuming the REST API implements consistency), hence the 404 status implies an unknown endpoint, underlining that the tested path segment is a static part.

For successfully validated variables, the path segment is replaced with an OpenAPI path parameter, indicating a variable in the endpoint's HTTP path. Consider Listing 5.8: Previously *GET /api/users/abc123*, the last path segment is validated as variable and therefore replaced with *{variable1}* in the HTTP path. Note that curly brackets imply a variable in OpenAPI. The endpoint also contains a new parameter *variable1* with its location being *path*. Endpoint variables found through the thesis' fuzzer follow the naming scheme *variableN*, where *N* is the number of the path variable inside the endpoint. Endpoints can contain an arbitrary number of path variables.

```
1 paths:
2   /api/users/{variable1}:
3     get:
4       parameters:
5         - name: variable1
6           in: path
```

Listing 5.8: OpenAPI example of discovered path parameter

5.3.4 Required request body properties

Another enhancement for reverse-engineered OpenAPI specs is to find out which JSON properties in a request body are required. With that knowledge, exploitative fuzzers could first concentrate on the required properties until a successful combination of property values is found, and then continue fuzzing optional parameters. Furthermore, some official specs by developers include this information and reconstruction of this data would bring the reverse-engineered spec closer to the official one.

To gain information of which JSON properties are required in the request body, the following algorithm is used: For an endpoint X, it is first checked whether the originally captured request R from the seeded traffic flows is replayable, i.e. if it can be resent without any modifications and the REST API responds with a successful response. If R is not replayable, the request body property values are fuzzed by concatenating a small amount of ASCII characters to them (this is the same approach as in Section 5.2.2; only string values are fuzzed). The goal here is to modify the request just enough so it will eventually get accepted by the API. One reason for non-replayable requests are duplicate resource names, i.e. a resource with the requested name already exists, but slight modifications (fuzzing) of the name cause successful creation of a new resource.

After a successful API response is achieved, either with or without fuzzing, the algorithm continues with the next step. Now, every JSON property is removed one by one, starting with the

first one. If *R* previously was found to be non-replayable, the body is again fuzzed with the same approach from the first step. A new request with the modified JSON body (with the respective property removed) is sent to the API. Since the earlier request was successful, if the response should be negative now, it must be because of the removed JSON property. So, if the request is not accepted, the removed JSON property must be required and is marked as such. Otherwise, if the API returns a successful response, the removed property apparently is optional since its lacking still yields a positive API response. This procedure is repeated until each JSON property has been removed once from the request body. The assumption here is that if the algorithm is able to produce a successful API response through applying simple string mutations, then it will work again the second time when sending the request with one property removed. This means that the second request should not fail because of failed string mutations, but only because of the missing request property. Figure 5.5 shows a visual overview of the algorithm.

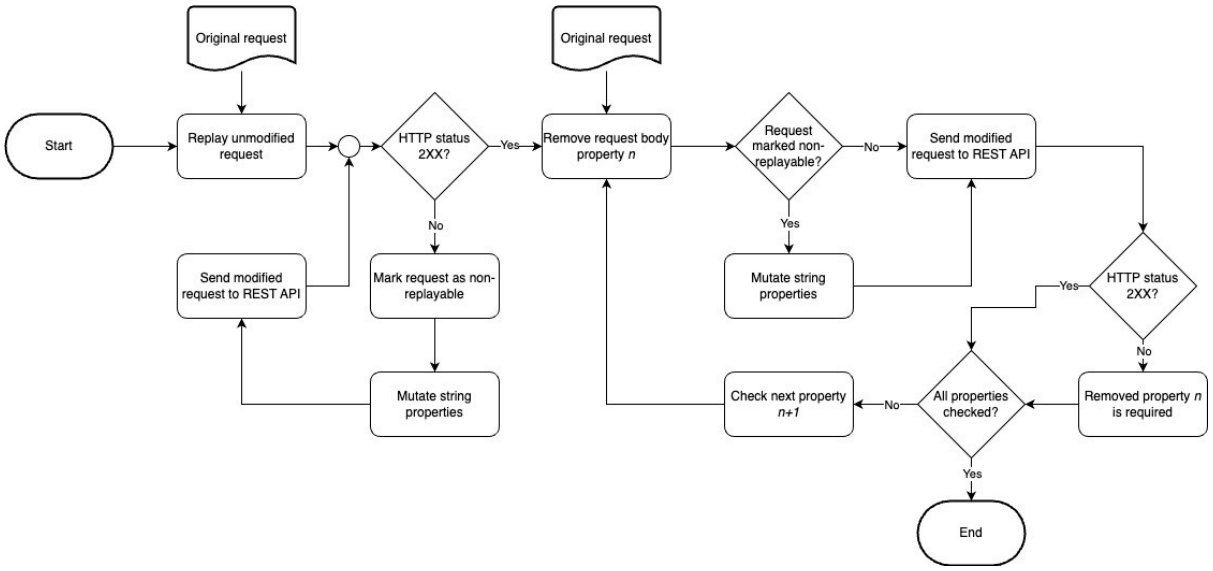

Figure 5.5: Required request body properties fuzzing logic

All properties marked as required are then written to the enriched OpenAPI spec. As part of OpenAPI's schema object, used for request body schemas, the OpenAPI specification [OAS24] supports the *required* field. It can hold an array of request body properties which must be set. Therefore, all JSON properties which are marked as required are put into the *required* array in the according schema object. Consider Listing 5.9 for an example where the properties *name* and *email* are marked as required. The last three lines define those properties of the endpoint's request body as required, while the third property *alt-email* remains optional by default.

```yaml
paths:
  /users/:
    post:
     requestBody:
       content:
         application/json:
           schema:
             type: object
             properties:
               name:
                 type: string
               email:
                 type: string
               alt-email:
                 type: string
             required:
               - name
               - email
```

Listing 5.9: OpenAPI example of required request body properties

5.3.5 Response objects

The OpenAPI specification allows the definition of possible HTTP response status codes and response objects. Those can be used by developers to better understand possible API responses, both for successful and failed requests.

Captured REST API responses are already defined in the base OpenAPI spec. Those requests will mostly be successful ones as they were sent by an official API client and were most likely accepted by the server. Conversely, this means that API responses for failed requests (i.e. negative responses) are hardly included in the base spec.

Because of the nature of fuzzing, it often generates invalid API requests resulting in negative API responses. Instead of thoroughly treating this characteristic as a disadvantage, it is used in favor of OpenAPI spec enrichment: All negative responses encountered through fuzzing are added to the enriched OpenAPI spec. This encompasses all fuzzing approaches and their corresponding requests explained in this section. Any negative response not already contained in the relevant OpenAPI's *responses* object is added to it. At the end of processing, the enriched spec contains the successful responses from traffic capturing as well as any failed responses from fuzzing.

Now that all methods are presented, the following chapter focuses on their evaluation.

6 Evaluation

In this chapter, the evaluation approach for measuring the effectivity of this thesis' methodology is explained and the results are analyzed; the implementation is also quickly discussed. Figure 6.1 shows an overview of the evaluation process. The following research questions are defined:

- **RQ1**: How effective is the extension of REST API specifications through fuzzing, given a base spec?
- **RQ2**: How effectively can the extended REST API specification be used for finding new API issues?

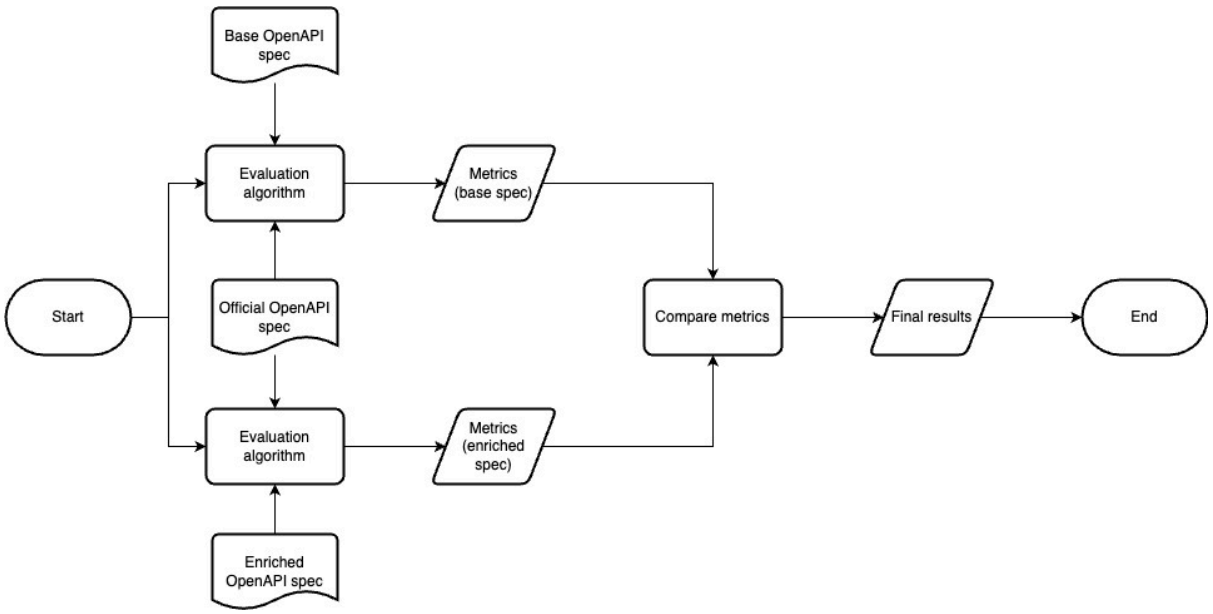

Figure 6.1: Evaluation process overview

6.1 Implementation

This section explains implementation details of the softwares implemented as part of this thesis. Both for the fuzzer and the evaluation algorithm, one software project is implemented, respectively. The fuzzing software roughly consists of five main parts and functionalities: Initialization (parsing collected traffic flows and conversion to internal objects), static traffic analysis, fuzzing logic, OpenAPI spec handling, and interoperability with the GLADE language learning algorithm. All fuzzer's parts are written in C# and use the latest .NET Core 8.

In the initialization part, a base OpenAPI spec is read and converted into .NET objects for further use. Additionally, the traffic traces in HAR format are deserialized using the HarSharp[1] library.

1. https://github.com/giacomelli/HarSharp

The algorithms for static analysis as well as fuzzing logic are mostly implemented in pure C#. However, for more comfortable handling of HTTP requests and responses, the Flurl[2] library is used. All logic for (de)serialization of OpenAPI specs (e.g. in the initialization and for writing enriched specs) makes use of Microsoft's OpenAPI.NET [3].

The interoperability with the GLADE language learning algorithm is a bit more complex. Bastani et al. released an implementation of their algorithm in [Bas+17] as a Java library[4]. Using Java libraries in .NET is not straightforward. However, there is an officially endorsed fork[5] of the author's original work which extends the library and makes it usable as a CLI tool. Usage as such allows calling the software from withing the thesis' fuzzer. The forked tool takes an executable program as oracle and runs it with the current mutation as a program parameter. It also needs a location of files containing the seed values. If the oracle program returns exit code 0, GLADE assumes the mutation is part of the target language. Otherwise, the checked mutation is not. These oracle programs are dynamically written by the fuzzer as needed. They contain a simple HTTP query to an internal REST API controller exposed by the fuzzer, allowing to externally trigger the fuzzer to fuzz the target REST API with the parameter mutation chosen by GLADE. The oracle program transmits both the fuzzed parameter (identified by an internal ID) and the chosen mutation to the fuzzer via REST. If the fuzzed parameter is accepted by the target API, the fuzzer returns a successful status code (200 OK) to the calling oracle program, which forwards this successful response to GLADE as exit code 0. If the target API does not accept the mutation, the fuzzer responds with a negative HTTP response (400 Bad Request), causing the oracle program to terminate with exit code 1, signalling a negative response to GLADE. In summary, when the fuzzer needs to fuzz a specific parameter via GLADE, it dynamically creates the oracle program, writes the gathered seed values, and calls the GLADE CLI tool with according parameters (for oracle program and seed locations). The CLI tool then executes the GLADE algorithm, using the oracle program and seeds, until GLADE's termination. As a result, the found regular expression is written to stdout and registered by the fuzzer for further processing. The internal REST API controller functions as a communication channel between oracle program and fuzzer, while the oracle program directly forwards fuzzing results (i.e. membership queries) to GLADE.

Similarly, the evaluation algorithm, explained in the following, is also implemented in C#. For some functionality, the same libraries are used as for the fuzzer, e.g. for serialization of JSON and OpenAPI specs.

6.2 Metrics

This section covers the approach and metrics used for evaluation to measure the effectivity of the proposed reverse engineering process. It also deals with special cases which are not optimally covered by the general approach and therefore require particular attention and careful treatment.

2. https://github.com/tmenier/Flurl
3. https://github.com/microsoft/OpenAPI.NET
4. https://github.com/obastani/glade
5. https://github.com/kuhy/glade

6.2.1 General approach

The basic idea is to analyze the changes between two OpenAPI specs and measure their similarity and differences. Generally, an official OpenAPI spec is compared to a reverse-engineered one, i.e. either a base or enriched spec. The difference in metrics for *official vs base* and *official vs enriched spec* can then be interpreted as a measurement of reverse-engineered information obtained through the methods presented in this thesis. A basic comparison of line changes or the length of the whole OpenAPI spec is not sufficient since these characteristics do not capture the logical, or semantical, changes in those specs, but only the syntactical ones. For example, the addition of a new HTTP method for a specific API endpoint could be a change of just one line, but may actually be an impactful change which would almost be neglected if only measured by line changes.

Instead, one metric used for measurements of spec comparison is chosen to be the *Jaccard index* [Mur96] (originally developed by Gilbert in 1884). It is a statistic that can be used for measuring the similarity between two sets. The Jaccard index of two sets A and B is defined as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

In other words, it returns the percentage of similarity of the two sets. Closely related, the *Jaccard distance* can be used for measuring dissimilarity and is defined as:

$$J_D(A,B) = 1 - J(A,B)$$

Tokenization For the evaluation of the thesis' approach, the sets are built using lexical tokenization. Before comparison of the two specs, any existing OpenAPI references are resolved (*$ref* keys) to ease direct comparison of spec elements. Each of the two to be compared OpenAPI specs have a unique set of token sets. The words contained in each OpenAPI spec are used as tokens and assigned to each according set. Token sets are not yet evaluated using the Jaccard index. This tokenization is done per spec level, i.e. only the keys of the key-value pairs on the current level are considered tokens. Keys' values and keys in deeper levels are ignored for the time being. For example, when tokenizing a certain OpenAPI path like */api/users/*, only the keys on this level, i.e. OpenAPI operations, are tokenized. Since operations use HTTP verbs as keys, the tokens may be *GET* and *POST*, which are then added to the spec's token set. After a level's tokens are processed, the algorithm continues with the value of the next mutual key in a DFS-based fashion and recursively runs the algorithm for that level until all tokens are analyzed. Tokens that are exclusive to one spec are already processed by adding them to the respective token set.

Ignored tokens Specific keys tend to only contain human-readable, arbitrary metadata or exist for reference only, but do not express any actual API functionality. This makes them nearly impossible to reverse-engineer, while only containing non-technical metadata. As a consequence, these keys are always ignored and not added to any token sets to prevent falsified measurements. The full list of ignored tokens includes *description*, *summary*, *title*, *operationId* and *tags*.

Information types There are two types of information which are analyzed through the algorithm: *Reconstructed* and *additional* information. Reconstructed information means data that was already present in the official spec and is also included in the reverse-engineered spec. Additional information means data that is not present in the official spec, but could be discovered anyway and is therefore present in the reverse-engineered spec.

Metrics calculation Both types of information require different types of evaluation. Simply using the Jaccard index on the built token sets would result in inaccurate results that cannot consider both types of information at once. Therefore, two different metrics are created, each one evaluating solely one type of information. For each spec level, the already crafted token sets are used for building the *token intersection* set T, which is the intersection of the current level's token sets A and B. Using the intersection is effectively the same as omitting elements exclusive to one spec only, which is what is needed here. For evaluating reconstructed information, spec 1's tokens are compared against the token intersection set T using the Jaccard index. This effectively compares spec 1's tokens against spec 2's tokens without any exclusive elements, resulting in the evaluation of how many tokens from spec 1 are contained in spec 2. For additional information, the comparison is very similar but in the opposite direction: Again using the already built set T, additional information is evaluated by comparing the token intersection set T against spec 2's tokens, this time using the Jaccard distance. This effectively compares spec 1's tokens without any exclusive elements against spec 2's tokens, resulting in the evaluation of how many additional, or exclusive, tokens are contained in spec 2. Usage of the Jaccard distance instead of the Jaccard index yields higher results, i.e. closer to 1, the more exclusive elements are present. Therefore, for both reconstructed and additional information, the higher the results, the better.

This can be summarized as follows: Let A and B be the token sets for specs 1 and 2, respectively, for a specific spec level present in both specs. The token intersection set T is defined as:

$$T = A \cap B$$

Reconstructed information is measured using:

$$I_{REC} = J(A,T).$$

Additional information is measured using:

$$I_{ADD} = J_D(T,B).$$

Using the level-based approach, each spec level is evaluated on its own, resulting in more fine-grained results. For each level's analysis, the results, calculated as explained above, are stored for that unique level. When all tokens are analyzed for both specs' levels, the saved results for a specific path are averaged to get an overall path metric. This is done for all analyzed paths, resulting in multiple path metrics, each one calculated from the path's associated child indices. For example, considering only I_{REC} indices, if a path */api/users/* has the indices 0.75 for level */api/users/* and 0.25 for level */api/users/ > GET*, the overall index for that path would be 0.5, meaning it has an average similarity of 50% regarding reconstructed information. The same would be done for additional information, i.e. averaging all path's I_{ADD} indices. All resulting path metrics are then again averaged to get the final spec metrics, one for reconstructed information and one for additional information.

Example Let us consider a simple example for this algorithm. Listing 6.1 shows two OpenAPI specs. Spec 1 is an official spec and spec 2 is an enriched spec. Starting with path mapping, the algorithm searches for one path in each spec which are semantically equivalent. This is straightforward in this example since both specs only have one path without any variables, which is */users/*. For the successfully mapped path, the algorithm now starts comparing the path's definitions from both specs. For each spec and each spec level, a unique token set is allocated. The tokens on this level, without any tokens from deeper levels, are added to each spec's token set. Spec 1 has the tokens *get* and *post*, so spec 1's set A for the current level is $A = [\prime get\prime, \prime post\prime]$. Spec 2 has only one token, *get*, so spec 2's token set B is $B = [\prime get\prime]$. Now that the token sets for the current level are built, the token intersection set T is constructed from A and B, which is $T = A \cap B = [\prime get\prime]$. To measure reconstructed information I_{REC} in the enriched spec, spec 2, the Jaccard index is calculated for sets A and T, which is 0.5. This means the enriched spec lists half of the tokens from the official spec on the current spec level. In this case, spec 2 is missing the token *post* from spec 1, which is penalized via this calculation. To measure additional information I_{ADD} in the enriched spec, the Jaccard distance is now calculated for sets T and B, which is 0. This makes sense since spec 2 has no additional tokens on this level, compared to spec 1.

After all calculations for the current spec level are done, the algorithm recursively continues with the next mutual token's level. In this example, the only mutual token is *get*, so the next level is */users/* > *get*. Again, new token sets are allocated for this spec level. The only token in both specs is *responses*. Therefore, $I_{REC} = 1$ (spec 2 has all tokens from spec 1) and $I_{ADD} = 0$ (spec 2 has no additional tokens). The next mutual (and only) token is *responses*, so the next level becomes */users/* > *get* > *responses*. For this level, the only token is *200*, so the same results are yielded ($I_{REC} = 1$ and $I_{ADD} = 0$).

The next level becomes */users/* > *get* > *responses* > *200*. Tokens on this level are a bit more interesting again. The sets are $A = [\prime content\prime]$ and $B = [\prime content\prime, \prime links\prime]$, so $T = [\prime content\prime]$. Spec 2 has all tokens from spec 1 (*content*), so $I_{REC} = 1$. However, spec 2 now has an additional, or exclusive, element (*links*), so using the Jaccard distance, $I_{ADD} = 0.5$. This result rewards the addition of the element *links* in the enriched spec, which in this case lists producer-consumer dependencies which are not present in the official spec. The next (and only) mutual token is *content*, so the next level becomes */users/* > *get* > *responses* > *200* > *content*. The *links* element is not further analyzed since there is no counterpart available for analysis. Of course, the algorithm would continue comparing the next level, but let us stop here. Gathering all calculated indices (for reconstructed information: $[0.5, 1, 1, 1]$; for additional information: $[0, 0, 0, 0.5]$), averaging those yields the path metrics $I_{REC} = 0.875$ and $I_{ADD} = 0.125$ for path */users/*. Since, in this example, this is the only path in both specs, that is equivalent to the final spec metrics. As a result, the enriched spec has 87.5% of reconstructed information on average and 12.5% of additional information on average.

```yaml
# Spec 1 (official)
paths:
  /users/:
    get:
      responses:
        '200':
          content:
            [...]
    post:
      [...]

# Spec 2 (enriched)
paths:
  /users/:
    get:
      responses:
        '200':
          content:
            [...]
          links:
            link1:
              parameters:
                [...]
```

Listing 6.1: Parts of two OpenAPI specs

6.2.2 Path mapping

While the general approach covers most parts of the OpenAPI specs, there are, however, some cases which require extra attention. The handling of defined endpoint variables cannot directly be handled with the general approach. Instead, to even consider general spec comparison, two semantically equivalent API paths need to be mapped first. The problem is that paths can have different definitions in the official spec and a reverse-engineered spec (i.e. base or enriched). This mostly concerns the definition of endpoint variables which are directly contained in the HTTP path. The problem, called *path mapping problem*, consists of two subproblems:

- **Different variable names**: Both specs define one or more variables in the HTTP path, but these variables have different names. An example are the definitions of the same path as (i) */users/{id}* and (ii) */users/{variable1}*. A variable at the same position has two different names, e.g. because the original variable name could not be reconstructed, and the remaining path structure is otherwise equivalent.
- **Concrete values**: Instead of defining variables, the reverse-engineered spec uses a concrete value for that variable that has been captured in the traffic traces, e.g. */users/{id}* in the official spec and */users/abc* in the reverse-engineered spec (where *abc* is a concrete value for variable *id*). This phenomenon arises from the difficulty to differentiate between variables and static path components. The problem was earlier defined as parameter classification problem in this thesis.

To solve these issues, the algorithm starts path mapping by analyzing all paths from the two specs. Let us consider the first subproblem (*different variable names*) during a comparison between an official spec and an enriched spec. For each path in the official spec, it is checked

whether there is a semantically equivalent counterpart in the enriched spec. For that, all defined path variables, both from the official spec and the enriched spec, are temporarily ignored. It is then checked whether the two paths without variables have equivalent structure, in which case they are semantically the same path. To prevent incorrect mappings between paths without any variables and paths with at least one variable, another logical check is needed. Consider comparison of path */users/* in the official spec and */users/{variable1}* in the enriched spec. Obviously, these paths are not the same. When ignoring variables, both paths would become */users/* and would therefore be wrongly considered equivalent. Therefore, the total number of variables in the official spec's path $|V|$ must be greater than or equal to the number of variables in the reverse-engineered spec's path $|V'|$, while there must also be at least one variable defined, i.e. $|V| \geq |V'| > 0$. That way, mapping of reverse-engineered paths containing variables to official specs' paths without variables cannot happen. This approach ensures accurate path mapping even when variable names differ.

For the second subproblem (*concrete values*), the approach is similar to the one for the parameter classification problem as the core problem is basically the same. Endpoint variables are detected by measuring all path segments' entropy and choosing the ones above a configurable threshold. These chosen path segments are then treated as variables. Further processing is the same as for the first subproblem above, i.e. variables are ignored and paths are checked for equivalent structure, while respecting the constraint regarding total number of variables. That means this subproblem requires an extra step for finding variables first, instead of having variables already defined directly in the spec.

Unmapped paths There may still be some cases in which the presented path mapping logic does not succeed and cannot map a reverse-engineered path to a path from an official spec. This can have different reasons, e.g. the reverse-engineered path contains variables which are not detected as such, or the official spec does not contain that path at all. In these cases, the reverse-engineered path cannot be properly analyzed and is therefore excluded from evaluation. The metric *unmapped paths* keeps track of the number of paths that are affected by this. Unmapped paths are deduplicated in a way so that two semantically equivalent unmapped paths are only counted as one unmapped path.

6.2.3 Evaluating path variables

Closely related to the path mapping problem is the problem of evaluating discovered endpoint variables. Unlike the standard evaluation approach, paths containing variables require special evaluation logic to accommodate for the paths' differences caused by variables. For that, after a path mapping is successfully found, the number of variables in both of the two paths is extracted. Since it is already known through path mapping that both paths are semantically equivalent, it is enough to simply compare the number of variables in this step. As a consequence, differently named variables are counted as semantically the same variable. For each variable and each spec, a synthetic token is added to a special token set which is used internally for computing the Jaccard index. This is effectively equivalent to calculating the percentage of discovered path variables in the reverse-engineered spec compared to the official spec. The resulting index is added to the corresponding path's list of Jaccard indices, which are averaged at the end to produce the overall path metric.

When paths contain variables, they are also defined in the endpoint's *parameters* object as a variable in *path*. The same logic explained above also applies to this evaluation step, so that the same path variable, just with a different name, is treated as semantically equivalent. On this spec level, the normal token sets are used and are shared with the other parameters. This special logic ensures that discovered path variables listed under *parameters* are also rewarded as reconstructed, even though they might have different names, which are always hard to reconstruct since they can be chosen arbitrarily.

For example, let us consider the mapped paths */users/{id}/{id2}* from an official spec and path */users/abc/{variable1}* from an enriched spec. The official spec's path contains two variables, while the enriched spec only contains one variable. The *parameters* objects also contain these path variables, i.e. *id* and *id2* in the official spec and *variable1* in the enriched spec. Therefore, it can be said that the enriched spec lists half of the available path variables, which is equivalent to a Jaccard index of 0.5. As a result, it is ensured that discovered endpoint variables can be evaluated as part of the standard evaluation approach after applying this special logic.

6.2.4 Evaluating global components

Global components specified via OpenAPI's *components* object allow declaration of security schemes which can be reused anywhere in the spec. Currently, security schemes are also the only relevant data for the evaluation which is stored in the components object. The current state of globally defined security schemes poses a difficulty for evaluation via the generic algorithm explained in this chapter. The reason for that is because the security scheme's name is directly defined via the key (or token). This means that these tokens can take arbitrary, unpredictable values, even for semantically equivalent schemes. As a consequence, comparison via tokens could become inaccurate since semantically equivalent security schemes can have completely different keys/names. Calculating Jaccard indices of disjunct token sets, which would always be the case for two differently-named security schemes, would artificially decrease the spec metric for reconstructed information and artificially increase the spec metric for additional information. Since reconstruction of these arbitrary names is very unprobable to succeed, comparison of the actual security schemes also becomes unprobable. Using the general algorithm, which recursively compares values of mutual tokens, would probably cause omitting evaluation of globally defined security schemes because their names (i.e. tokens) differ.

For example, consider Listing 6.2, where both specs define a security scheme for the Bearer authentication scheme. That scheme is used in both official and enriched specs, however, the official spec names that scheme x while the enriched spec names it y. Otherwise, the enriched spec perfectly reconstructed the scheme. Using the general algorithm, this perfect reconstruction would not be rewarded because the schemes' keys are not equivalent (i.e. no mutual tokens). The definition would even be penalized when comparing spec level *components* > *securitySchemes* since the token sets (spec 1 has $A = [\prime x\prime]$; spec 2 has $B = [\prime y\prime]$) are disjunct. Instead, this special case is handled by using the combination of a scheme's *type* and *scheme*. Those two OpenAPI-specific keys are used for mapping security schemes across specs. This effectively ignores the scheme's arbitrarily chooseable name and directly jumps to the comparison of the actual definition, which is then evaluated using the normal algorithm. However, if no scheme can be mapped, e.g. because there are no mutual security scheme types, evaluation of security schemes is handled the same way as in the general algorithm, i.e. no further comparison is done. Approaching this special case in a best-effort way results in fair evaluation of clearly

reconstructed security information, while handling it the default way when special treatment is not possible.

```yaml
# Spec 1 (official)
components:
  securitySchemes:
    x:
      type: http
      scheme: bearer


# Spec 2 (enriched)
components:
  securitySchemes:
    y:
      type: http
      scheme: bearer
```

Listing 6.2: Two OpenAPI specs defining global security schemes

6.2.5 Path coverage

Additionally, the *path coverage* metric serves as a metric for orientation. It describes the number of mapped API paths included in the reverse-engineered spec compared to the official spec and gives an idea of how many paths are included in the evaluation. Unmapped paths, which may be included in the reverse-engineered spec but could not be mapped to a semantically equivalent path from the official spec, are not considered in the path coverage metric.

Path coverage is heavily dependent on how much API functionality was executed during traffic capturing because the number of API functionality used also potentially correlates with the number of API endpoints contacted. Using only one function during traffic capturing may result in only one API endpoint being contacted, with many ones remaining uncovered. Since the methods presented in this thesis currently do not bother with uncovered or "hidden" endpoints, this metric should only be used for orientation and not for measuring the methods' effectivity. Instead, it must be ensured that the number of endpoints contacted during traffic capturing is maximized, which will then be reflected through the path coverage metric.

6.3 Target applications, setup and traffic collection

Three target applications are chosen for evaluation. All applications' REST APIs should be accessible through an official, web-based API client, i.e. a web browser frontend application directly provided by the developer (no third-party client). The applications are divided into small, medium and large, measured by their official OpenAPI spec's number of API paths. The entire software evaluated is distributed under Open-source licenses and available at no extra cost. Evaluated REST APIs are:

- **Mattermost** [MTTR]: An online chat optimized for usage in organizations, providing direct chats, group chats, organization-wide messaging channels etc. Its feature set is

similar to the proprietary Slack, which is provided as SaaS. The official OpenAPI spec[6] lists 368 API paths and is therefore considered a *large* REST API.

- **Open WebUI** [OWUI]: A ChatGPT-style web interface for usage of various backends providing Large Language Models (ollama, OpenAI-compatible APIs). Besides LLM inference (chatting), the application allows user creation, setup of different backends, management of models, Text-to-Speech configuration etc. Note that this application is composed of multiple services which all expose their own REST API (e.g. middleware APIs for the actual backend APIs). The REST API evaluated here is the "main" API (available at */api/v1/*) responsible for handling backend-unspecific interactions with the user interface. To get the official OpenAPI spec, the application must be running and is then available at */api/v1/openapi.json*. It lists 107 API paths and is considered a *medium*-sized REST API.
- **Bunnybook** [BB]: This application is a "tiny social network". It features functionality for posting content, friend requests, a page listing friends' posts, user search and private messaging. This is more of a demo or tech stack showcase application, however, its size perfectly fits the requirements for a *small* REST API. The official OpenAPI spec is hosted by the running instance at *:8000/openapi.json* and lists 21 API paths. Other potential *small* "real-world" (or "production-ready") applications offer *at least* around 50 paths.

6.3.1 Environmental setup

All target applications are setup and run using the default configuration unless explicitly stated otherwise.

Mattermost is setup using:

```
1 podman run -p 127.0.0.1:8065:8065 mattermost/mattermost-preview
```

The *mattermost/mattermost-preview* container image is a self-contained Mattermost installation, including database software and preset configuration for fast deployments of Mattermost. It should not be used in production environments, but is convenient for testing (and fuzzing). The version used is 9.8.0.

Open WebUI is setup using:

```
1 podman run -p 127.0.0.1:3000:8080 -e
    OLLAMA_BASE_URL=http://127.0.0.1:11434
    ghcr.io/open-webui/open-webui:main
```

The setup uses the official Open WebUI container image and is configured for a locally-running ollama instance for LLM inference. Version 0.3.22 is used for evaluation.

Bunnybook is setup using the provided *docker-compose.yaml* file using command *podman compose up*. Because this application enforces strict rate limiting by default, which impairs the fuzzer's effectivity and ability to function properly, the source code is patched to completely disable rate limiting. Additionally, the default expiration of JWT tokens is extended from 15 minutes to 1 year via the provided configuration file, so that tokens do not expire during fuzzing.

6. Available from a dynamically generated URL at https://api.mattermost.com, built from multiple OpenAPI files at https://github.com/mattermost/mattermost/tree/master/api/v4/source

To properly cover functionality of friend requests, a dummy user is created via the web interface. The creation of this user is not part of the captured traffic traces. The latest available version is used for evaluation, which is commit *6666f4a*.

6.3.2 Traffic collection

This section describes the steps executed during traffic capturing for each of the three chosen target applications. The traffic traces are captured using the following interactions in the frontend:

- **Mattermost**
  - Open web application
  - Sign-up for new user account
  - Complete initial application setup: Create organization, skip other questions
  - Send message to default organization channel
  - Create new channel within organization
  - Set user's online status to *away*
- **Open WebUI**
  - Open web application
  - Sign-up for new user account
  - Start new chat, select model, send one message and await response
  - Go to admin panel
  - Create new user via admin panel
  - Go to admin panel settings
  - Click on *Users*
  - Click on *Connections*, disable OpenAI API, save settings
- **Bunnybook**
  - Open web application
  - Sign-up for new user account
  - Login with newly created account
  - Create a public post
  - Create a comment attached to that post
  - Search for the dummy user via the search field, visit its profile
  - Send a friend request to the dummy user
  - Click on *Friends* icon in the top-right corner

- Click on *Conversations* icon in the top-right corner
- Click on *Notifications* icon in the top-right corner

6.4 RQ1: Effectivity of reverse engineering

This section covers the evaluation's results for the target applications regarding the first research question. Table 6.1 lists the final metrics for the base and enriched OpenAPI specs as calculated by the evaluation algorithm. It features the results for reconstructed information (I_{REC}) and additional information (I_{ADD}), both for base and enriched specs, respectively, their absolute differences, as well as the number of paths covered (i.e. without unmapped paths) and the number of unmapped paths which cannot be further analyzed. The numbers are rounded to three decimal places.

Since all base OpenAPI specs are created using existing software (see Section 5.1.2), these results can also be interpreted as a direct difference to current methods.

Application	I_{REC} Base	I_{ADD} Base	I_{REC} Enriched	I_{ADD} Enriched	Path coverage	Path cov. %	I_{REC} diff	I_{ADD} diff	Unmapped paths
Mattermost	0.782	0.089	0.849	0.117	26/368	0.071	0.067	0.028	15
Open WebUI	0.668	0.035	0.701	0.028	17/107	0.159	0.033	-0.007	0
Bunnybook	0.755	0.031	0.859	0.122	11/21	0.524	0.105	0.091	1

Table 6.1: Evaluation result metrics: Reconstructed (I_{REC}) and additional (I_{ADD}) information for base and enriched OpenAPI specs

6.4.1 Mattermost

Results for Mattermost show an average metric for the base spec's reconstructed information of 78.2% (I_{REC} Base). The base spec has exclusive/additional information of 8.9% on average (I_{ADD} Base). On the other hand, both metrics are higher for the enriched spec, with 84.9% of reconstructed information (I_{REC} Enriched) and 11.7% for additional information (I_{ADD} Enriched) compared to the official spec. This shows that the enriched OpenAPI spec contains 8.6% more reconstructed information and 31.5% more additional information on average compared to the base spec. Regarding path coverage, 26 paths are covered, which is the highest for the evaluated applications, but the least regarding percentaged path coverage because Mattermost is by far the largest application. It is hard to cover all paths; some are not even used via the web frontend, e.g. server-to-server APIs.

Out of the 16 paths that contain variables, 6 are successfully verified through request generation, yielding a success rate of 37.5%. Mattermost uses additional dependency patterns currently not implemented in the thesis' fuzzer, e.g. usage of resource IDs that can only be obtained from other endpoints' *request* bodies. Therefore, the request generation's effectivity is held back by the diversity of dependency patterns in Mattermost, but the achieved result can be considered a good start.

Security requirements For Mattermost's REST API, the security requirements are worth mentioning. The official OpenAPI spec defines two security schemes: *bearerAuth* (with a lower b) and *BearerAuth* (with a capital B). As the naming suggests, both schemes use the Bearer authentication scheme, but *bearerAuth* additionally sets the *bearerFormat* to *token*. The latter is an optional key and can be set to arbitrary values [OABA]. Interestingly, the *bearerAuth* scheme is used as a global requirement, which means it is required for all API endpoints, unless another security requirement is explicitly set directly within an endpoint. The *BearerAuth* scheme is explicitly set for all paths beginning with */plugins/*. Those paths seem to offer plugin functionality for Mattermost; some of those are described as "internal" endpoints in the official spec.

However, the thesis' fuzzer detects another authentication requirement, which is cookie authentication via a cookie called *MMAUTHTOKEN*. That cookie is contained in most of the captured API traffic, while no single API request contains the Authorization header, which is required for the Bearer scheme as defined in the official spec. Obviously, this is a discrepancy between the official spec's definition and the actual implementation as defined in the enriched spec. It is manually verified that the authentication requirements found by the fuzzer are indeed correct, i.e. replaying a working request while stripping the MMAUTHTOKEN cookie causes the API to return HTTP status *401 Unauthorized*. This case is a great example for undocumented REST API behavior which is successfully uncovered by the fuzzer. As a result, security-focused, or exploitative, REST fuzzers can potentially test the implementation of Mattermost's cookie authentication regarding security vulnerabilities using the enriched spec, even though undocumented in the official OpenAPI spec.

Path mapping An issue with the evaluation of the reverse-engineered specs for Mattermost is path mapping. For this target application, some API paths in the traffic traces contain the string *me*, which serves as an abbreviation for the currently authenticated user's ID. This means that *me* is actually a value for a variable path segment. However, since the string has low entropy, it is not considered a candidate variable and therefore neither checked using request generation. This causes the path mapping algorithm to not find a semantically equivalent path from the official spec. From the 15 unmapped paths, 13 paths are unmapped due to this issue. The other two unmapped paths are not contained in the official spec at all (these are */api/v4/cloud/products/selfhosted* and */api/v4/system/onboarding/complete*). Therefore, these paths are automatically excluded from evaluation.

Parameter patterns Mattermost's REST API uses many HTTP parameters. In total, the fuzzer probes 31 API parameters for which an accepted pattern is inferred using the language learning algorithm. These learnt patterns allow observation of another deviation between official OpenAPI spec and actual implementation: While the official spec defines 10 of those 31 parameters as *type: number* (*number* meaning float or double according to the OpenAPI specification [OAS24]), 9 of those 10 parameters actually accept arbitrary strings containing alphanumeric and special characters. Only one parameter is effectively constrained to exclusively accept floating point numbers.

This observation suggests that the official spec lists parameters with values that *should* be sent to the API in order to make sense in the application's business logic, but not what *can* be sent to the API and still be accepted regardless. Deviations between expectations of what should be sent and what can be sent make room for unexpected behavior, potentially exposing

vulnerabilities to attackers. Reasons for these discrepancies could be oversights by the developers in the implementation or an inaccurately written OpenAPI spec not reflecting the actual implementation.

6.4.2 Open WebUI

The results for Open WebUI show that, on average, the base spec contains 66.8% of reconstructed information (I_{REC} Base), while the enriched spec reaches 70.1% (I_{REC} Enriched). This is an additional 4.9% of information successfully reconstructed in the enriched spec over the base spec. An interesting phenomenon is happening for the metric of additional information, since the enriched spec's result (I_{ADD} Enriched) is actually minimally lower (absolute difference of 0.7%) than for the base spec (I_{ADD} Base). Of course, enriched specs must always have at least as much of additional information as base specs since no already available information is stripped, so it is clear that this is a mathematical inaccuracy. This stems from the fact that the enriched spec has more reconstructed elements which do not contain exclusive information on their own. Because every reconstructed element also adds an index for additional/exclusive information, which is 0 when there is no exclusive information contained in the reconstructed object, this increases the number of indices being 0 and therefore decreases the overall average for I_{ADD}. Analyzing the metrics, another reason that the enriched spec's I_{ADD} is not significantly higher than the base spec's one is that only one extra token from the enriched spec is considered exclusive, which is a pattern for one specific parameter.

More parameters are not analyzed using the language learning algorithm since this is the only parameter defined in the whole spec (for both base and enriched specs). This indicates that the REST API generally makes rare use of HTTP parameters such as query parameters. Checking the official OpenAPI spec, it is verified that the spec indeed only defines 13 query parameters for its 107 API paths, which results in a parameters/path ratio of 0.12. For reference, Mattermost defines 267 query parameters for its 368 paths, which is a parameters/path ratio of 0.73. Since the enriched spec neither contains any producer-consumer dependencies which could potentially increase I_{ADD} (because official specs tend to rarely include information on dependencies), all other tokens are considered reconstructed information. For example, there are multiple required request body properties found by the fuzzer and included in the enriched spec (not found in the base spec), however, this data is also included in the official spec and therefore classified reconstructed information as well.

API dependencies Interestingly, there are producer-consumer dependencies for this REST API, e.g. between starting a chat (*GET /chats/new*) and sending a message in a chat (*POST /chats/{id}*). However, these dependencies are not found by the thesis' fuzzer during the static analysis phase because the REST API does not return HTTP status code 201 for successful resource creation (as explained in Assumption 5.2.2), but instead uses the more generic status code 200. Therefore, that endpoint is not considered an eligible producer and hence not analyzed further. Apart from that, the dependency pattern found in the captured traffic is identical to the one assumed and implemented by the fuzzer (see Section 5.2.1.1), so that this dependency would be detectable. It can be argued that returning HTTP status 200 for resource creation is less appropriate than status 201, which is clearly intended for giving the client hints about creation of resources [RFC9110, chapter 15.3.2]. Since there is no inferred dependency information available, request generation also cannot work correctly. For this application, there is only one

path containing variables, in turn being the only one at all which would be verifiable using request generation.

Security requirements Another interesting point found for this REST API is its implementation of authentication requirements. While fuzzing for required authentication data, none was found. Therefore, the enriched OpenAPI spec does not include any security requirements, same as the base spec. However, from analyzing traffic traces, reading the official OpenAPI spec and manual verification, it is clear that the API actually requires authentication and uses the Bearer authentication scheme. The reason why authentication is not properly detected by the fuzzer is that the official API client sends duplicate authentication information, both via the Authorization header and a custom cookie called *token*. In both locations, an identical JWT token is used. The presence of only one of those two is sufficient for authentication to succeed.

As a consequence, since the fuzzer currently removes one parameter at a time, i.e. either a header or a cookie, the other one is still present and results in successful authentication. To the fuzzer, it looks as though no authentication is required since all requests are accepted without any 4XX status codes, even after removing each header and cookie once. This special problem can be fixed by fuzzing all possible header/cookie combinations and noting all absent headers/cookies when a fuzzed request fails due to missing authentication. This potential fix increases the required requests sent to the API exponentially.

6.4.3 Bunnybook

The results show that reconstructed information is 13.8% higher (I_{REC} Base = 75.5%, I_{REC} Enriched = 85.9%) and additional information is 294% higher on average (I_{ADD} Base = 3.1%, I_{ADD} Enriched = 12.2%) in the enriched spec compared to the base spec. The strong boost in additional information comes from a mix of multiple inferred parameter patterns, dependency information as well as security requirements, all of which are absent in the official spec and hence categorized as additional information.

Thanks to request generation, 5 of 8 endpoints' path variables are successfully verified, which yields a success rate of 62.5%. While for the remaining 3 endpoints, the algorithm successfully creates dependent resources through the producer, the API responds with status code 403 when trying to use these resources with their respective consumer endpoints. In this special case, the resources are user objects, and the consumer endpoints accept user IDs as a path variable for certain user-related actions, e.g. listing a user's conversations. Since all fuzzing requests are authenticated using the captured authentication token, the authenticated user does not match the newly created target user, and hence the API correctly declines the API request, causing request generation to fail in these cases.

Security requirements What is interesting about the resulting enriched OpenAPI spec are the discovered security requirements, or authentication schemes. While neither official nor base spec define any authentication requirements, the enriched spec contains that information; more specifically, the REST API implements the Bearer authentication scheme. However, through this additional information, it becomes clear that only a few specific endpoints actually require authentication, e.g. for creating a public post. All other endpoints can be accessed without authentication.

Looking at the endpoints with no authentication requirements, it turns out that some of those return private user data. For example, the API allows unauthenticated access to an endpoint *GET /web/profiles/{userId}/notifications* returning private notifications of a user, containing received friend requests, message data and potentially other private information. This allows an attacker to leak a user's notifications simply by knowing their user ID, which can be easily obtained, e.g. through the user search API (*GET /web/profiles*), which also allows unauthenticated access. Of course, this target application is treated as "not production-ready", and obviously for a reason, but nevertheless, it successfully demonstrates the fuzzer's ability to expose critical security vulnerabilities like in this case through enrichment of OpenAPI specs.

Path variable detection Another concern worth mentioning is one misclassification of a path variable (part of the parameter classification problem). Bunnybook's enriched OpenAPI spec defines one path variable which is not actually a variable. The official spec lists that path as */web/profiles/{profile_id}/outgoing_friend_requests/{target_profile_id}*, but the enriched spec lists it as */web/profiles/{variable1}/{variable2}/{variable3}*, meaning the static part *outgoing_friend_requests* is wrongly classified as variable. This is because the string's entropy reaches the set entropy threshold. That issue is also the reason for the one unmapped path since, naturally, there is no path defined in the official spec which matches those three variables.

The problem can be fixed by requiring successful verification of variable parts through request generation (after all, this exact problem is the reason why request generation exists). However, since request generation might not be successful for more complex producer-consumer dependency patterns in its first iteration, it is possible that not all truly variable parts end up being in the enriched spec because their verification fails. Therefore, this might not be a universal solution but requires careful consideration on a case-by-case review until request generation supports more dependency patterns encountered in the wild. Since this is the only misclassification as part of the parameter classification problem in this evaluation, it suggests that the current entropy-based approach works well enough for the vast amount of cases, while the number of false positives is kept very low (and can be completely removed as explained).

6.4.4 Summary

The evaluation results indicate successful reverse engineering of the evaluated REST APIs. Across the board, the scored numbers for the enriched specs beat the base specs' numbers, meaning the proposed methods are effective. Averaged over all evaluated applications, the presented approach yields an increase of 9.3% of reconstructed information, and a very notable jump of 71.2% for additional information. The results show interesting observations regarding security requirements and API quirks, e.g. sending duplicate authentication data, and inaccurate documentation in official OpenAPI specs, such as accepting a broader amount of parameter types, or undocumented cookie authentication. For one application, Bunnybook, security vulnerabilities are exposed purely through deeper understanding of the API enabled by the thesis' fuzzer, which result in unintentional exposure of private data. Request generation, a proposed method for path variable verification through resolving statically inferred dependency graphs, works well for the beginning (success rate of 50% when dependency data is available), but, as expected, can be further improved through support of a broader spectrum of dependency patterns used in REST APIs.

6.5 RQ2: Effectivity of exploitative REST fuzzing

For analysis of RQ2, an exploitative fuzzer is run against the three target applications using the reverse-engineered specs, i.e. with their base and enriched specs respectively. It is evaluated whether the fuzzer is able to find more problems and possible vulnerabilities using the enriched spec compared to the base spec. For this evaluation, Microsoft's RESTler fuzzer is used because of its stateful fuzzing approach which potentially learns critical request sequences and searches for logical API issues (see Section 4.2.3.1). In addition to scientific research about RESTler [AGP19] [AGP20], it offers extensive documentation which makes it possible to quickly configure the fuzzer to one's needs. RESTler offers three fuzzing modes: *Test*, *Fuzz-lean* and *Fuzz*. The Test mode is used for testing basic functionality to validate the fuzzer's configuration and only fuzzes each endpoint once (smoke test). Fuzz-lean is used for quick but more extensive fuzzing in hope for easily exposed bugs. The Fuzz mode is the full-blown fuzzing mode which tests many different combinations of API requests.

For this evaluation, RESTler is run in Fuzz mode, otherwise using the default configuration. RESTler uses an internal *RESTler grammar* which needs to be compiled based on an OpenAPI spec. Since this grammar is stored in JSON format, based on differences in these RESTler grammars for base and enriched specs, it can be analyzed which parts of an OpenAPI spec are actually used for fuzzing and which parts are ignored and hence do not contribute to the fuzzing process.

Fuzzers generally use so-called *time budgets* as an allocation of wall-clock time in which the fuzzer is allowed to do its work. The fuzzing process is stopped when reaching this time limit. By default, RESTler uses a time budget of 168 hours which equals 7 days. For this evaluation, the time budget is set to 12 hours; this roughly equals letting the fuzzer run over night. In the rest of this section, the fuzzing results are discussed for each target application.

6.5.1 Mattermost

As explained in Section 6.4.1, Mattermost uses cookie authentication. Unlike the thesis' fuzzer, RESTler does not have any access to actual traffic traces and only uses an OpenAPI spec for further analysis. To let RESTler know how to authenticate towards the REST API, the required authentication cookie is statically added as part of the Cookie header via RESTler's configuration at *Compile/defaultDict.json*, which allows declaration of custom headers, among other things.

API issues and bugs RESTler reports multiple reproducible bugs in the Mattermost API. These bugs are categorized in bug buckets, i.e. requests triggering the same type of bug, just with different values, are sorted into the same bug bucket. In total, there are two types of bugs found, triggered by (i) invalid parameter values and (ii) invalid payloads. RESTler reports a third bug bucket, however, for these bugs, the API returns HTTP status *501 Not Implemented* [RFC9110, chapter 15.6.2], which cannot really be considered a bug since this seems to be expected behavior by the developer when accessing unimplemented functionality, even though the status code lays in the 5XX range. This "bug" is triggered by omitting the query parameter *format*, whose only valid value seems to be *old*, for endpoint *GET /api/v4/config/client*. In this case, the API responds with a specific error message saying that the "new format is not implemented", indicating that *format=old* must always be present.

For the other two bugs, number (i) is found using both base and enriched specs. The bug is triggered by accessing an endpoint with a very specific combination of query parameters and values[7] and results in the REST API returning HTTP status *500 Internal Server Error*. Bug number (ii) is only found using the base spec, but not using the enriched spec. This bug can be triggered by uploading an invalid JSON payload via *PUT*[8]. Accepted by the API validation, the data gets forwarded to the database layer, where it is finally denied due to schema constraints, causing an exception in the application.

The reason why this bug is not found using the enriched spec is that this specific endpoint requires certain path variables. The enriched spec properly defines path variables for the problematic endpoint, but RESTler seems to struggle acquiring valid values for these path variables in order to access and mutate existing API resources. On the other hand, the base spec does not define any path variables and instead uses the captured values, which already point to valid API resources, therefore being readily available to RESTler. More specifically, the endpoint requires a valid user and a valid team ID. Even though the enriched spec contains producer-consumer dependencies for that endpoint, RESTler is not able to create a new user resource because it does not send a valid e-mail address as part of the JSON payload's *email* property to the endpoint *POST /api/v4/users*. Although RESTler might understand the dependencies, it is not able to properly resolve those as the first step for resolving the dependency graph, the user creation, fails.

Open problems This underlines the importance of knowing and defining format constraints for both parameters and JSON properties, e.g. using the *pattern* key as implemented for HTTP parameters by the thesis' fuzzer. Not even the official OpenAPI spec for Mattermost defines a more specific format than *type: string* for the *email* property. Even though other REST API fuzzing tools like CATS might be able to infer correct string constraints from property names [CATF], they still need to be able to resolve dependency graphs in order to generate acceptable API requests for most endpoints (which does not seem to be the case for CATS).

Unfortunately, RESTler does not seem to have support for OpenAPI's *pattern* key, which makes definition of API parameters using that key ineffective, at least in its current state. Lack of support is verified by analyzing the RESTler grammar generated using the enriched spec. The grammar does not feature any usage of the *pattern* key whatsoever. Other features used from the enriched spec, beside endpoint dependencies (through *links* and *x-restler-annotations* objects) and path variables, are required request body properties. All other features of the enriched spec, such as security requirements, expected HTTP responses, and, as mentioned, string constraints through the *pattern* key are not used by the RESTler fuzzer.

6.5.2 Open WebUI

Since Open WebUI uses Bearer authentication, RESTler is manually configured to include a valid Bearer token in its API requests. Additionally, this application has the quirk of returning HTTP status *307 Temporary Redirect* [RFC9110, chapter 15.4.8] on certain GET endpoints when no trailing slash is provided. The client is redirected to the same path, including the

7. The problematic request is: *GET /api/v4/teams?page=7308341037009178271&per_page=1.23&include_total_count=fuzzstring&exclude_policy_constrained=fuzzstring*
8. The problematic request is: *PUT /api/v4/users/$USER_ID/teams/$TEAM_ID/channels/categories*, e.g. using JSON payload *[{}]*

trailing slash. Since RESTler does not support HTTP redirects, which is also mentioned as one limitation in its paper [AGP19], as a workaround, required trailing slashes are manually injected into the RESTler grammar for affected endpoints.

API issues and bugs For both base and enriched specs, one bug is found: When requesting a specific endpoint with a query parameter set to a big number[9], the API returns HTTP status 500. There are no other bugs found for this application, so in this case, the enriched spec does not contribute to finding further API issues.

Fuzzing path variables Moreover, thanks to the declared path variables in the enriched spec, RESTler is able to fuzz these variables. This is applicable to one endpoint, *GET /api/v1/chats/{chat_id}*. However, since no dependencies are defined (as explained in Section 6.4.2), RESTler cannot obtain and use valid resource IDs for this and instead uses generated fuzz values. In this evaluation, no additional bugs are found through this; all requests to that endpoint containing non-existent resource IDs, i.e. all requests, are denied with HTTP status 401. If a dependency would have been added for its producer endpoint *POST /api/v1/chats/new*, this would probably be a case which can be handled by RESTler in its current state. Resource creation should be successful since the producing endpoint has no special string constraints for its request body properties (in contrast to Mattermost, for example). Therefore, the newly produced resource ID could be used in the consumer endpoint.

This suggests that declaring producer-consumer dependencies could potentially be valuable information in order to fuzz deeper API logic. In this case, however, lack of declaration of dependencies prevents RESTler from using valid resource IDs for the consumer endpoint's path variables, hence all requests to that endpoint are promptly denied.

6.5.3 Bunnybook

Bunnybook requires Bearer authentication. Again, a valid Bearer token is statically provided to RESTler via its configuration. Bunnybook's enriched OpenAPI spec scores the highest metrics in this evaluation, for both reconstructed and additional information. This means that the enriched spec for this target application, compared to its official spec, should be the most meaningful reverse-engineered OpenAPI spec in this evaluation.

API issues and bugs RESTler is able to find reproducible bugs that can be categorized into two bug buckets. However, these bugs are neither found through base nor enriched specs, but through the statically added Authorization header. Both bugs are authentication-related and can be triggered by sending (i) an empty JSON object *{}* as value for the Authorization header via *GET /web/posts?wall_profile_id=fuzzstring&limit=1.23* and (ii) by sending binary data as the Authorization header's value to *POST /web/register*. For bug (i), the REST API returns HTTP status 500; bug (ii) results in HTTP status *502 Bad Gateway*. Certainly, both bugs found are very interesting because they can be triggered through manipulation of authentication data. This is an indication of possible bugs in the authentication logic and could be worth further analysis. Unfortunately, those bugs would not have been found if valid authentication data would not

9. The problematic request is: *GET /api/v1/chats/?page=79365262329813061369*

have been provided manually in the first place. Although these authentication requirements are successfully reverse-engineered in the enriched spec, RESTler seemingly does not understand how to use that information to produce intelligently crafted API requests. It is not even required to actually know how to fetch valid authentication tokens from the API (however, this would be another step in the right direction) since both bugs found work without possessing valid authentication information at all. Therefore, it would be enough for a fuzzer to understand that Bearer authentication via the Authorization header is used by the API to find the bugs discovered in this evaluation.

6.5.4 Summary

While the enriched specs beat the base specs regarding information content about the target APIs, they currently do not effectively contribute to finding more API issues and bugs. Unfortunately, important OpenAPI features added by the thesis' fuzzer are currently unsupported by RESTler. This especially affects security requirements as well as parameter patterns. These spec features therefore remain unused, but seem to be of special importance when looking for new bugs: Security requirements allow understanding and fuzzing of authentication logic, while parameter patterns could generally increase fuzzing effectivity and enable type-specific exploitation logic.

Indeed, one application has authentication-related bugs which could potentially result in security vulnerabilities, but are only found using a semi-automatic approach with manual initial aid. In spite of the source for this bug not being any OpenAPI spec directly, this clearly shows that authentication data is an important target for fuzzing. Current REST fuzzers (at least RESTler, but possibly more) should be extended to process the security requirements featured in OpenAPI specs. Further enrichment of specs towards API authentication, e.g. how to fetch valid auth tokens, could provide even more valuable information to REST fuzzers and eliminate the need for manual authentication setup (automatic cycle of (i) user creation, (ii) extracting auth data, (iii) using auth data for calling endpoints, etc.). However, modeling that type of information seems to be currently unsupported by the OpenAPI specification and might require custom spec extensions.

Another issue is resolving of dependency graphs: While two enriched OpenAPI specs in this evaluation contain dependency information, RESTler is unable to properly resolve these due to failing requests to producer endpoints, the root cause being unsatisfied request body property constraints. Similar to query parameters, whose parameter constraints are added by the thesis' fuzzer, that approach could be extended to request body properties. This could in turn allow more effective fuzzing through generation of acceptable request parameters and potentially result in higher success rates for resolving dependency graphs, enabling fuzzing of deeper service states.

Ultimately, it may be worth mentioning that it is hard to say whether there are no further security issues found because the OpenAPI specs cannot be used effectively enough, or because the API has no security issues. While respecting the fact that it is safe to assume that every software has (unknown) security issues, it is not trivial to prove that those can be triggered via the API layer.

7 Limitations

This chapter covers the limitations of the presented method and the evaluation approach. The approach heavily relies on traffic flows captured between an API client and the REST API. While it is very simple to capture network traffic for browser-based REST API clients using the integrated DevTools, this can be challenging for other types of applications, such as native mobile or embedded apps. Other approaches are necessary for these non-browser API clients. This includes using proxy servers like mitmproxy for traffic interception. Most often, API clients communicate using TLS-encrypted HTTPS which additionally requires traffic decryption in order to gain access to plaintext traffic flows. However, these (non-browser) API clients may use techniques that stop interception of encrypted traffic, e.g. certificate pinning. This requires the server's TLS certificate to be signed by a hardcoded CA, preventing MITM attacks through injection of a trusted, self-signed certificate. Circumvention of certificate pinning often demands advanced dynamic instrumentation techniques with tools like *frida*[1] and may require elevated privileges on the client's device, posing another challenge in itself on certain locked-down devices. Since traffic flows are the requirement for the whole method, not being able to acquire such flows makes the method inoperable.

Other limitations include different measures deployed by the REST API. Servers may require API messages to be cryptographically signed, e.g. using HTTP Message Signatures [RFC9421]. This requires the fuzzer to implement proper signing of outgoing messages and access to the relevant keying material. Depending on the exact implementation, this may also limit replaying of captured API requests to a specific time window through verification of the signature's expiration. An example for a similar implementation is the S3 protocol [S3], also based on the REST architecture. It uses a custom authentication and signature scheme which prevents tampering and limits replaying to a short time window. Additionally, rate limiting is an effective countermeasure against fuzzing since it limits the allowed number of requests in a specific period of time. None of these mechanisms can completely eliminate fuzzing, but they make it more difficult, increasing the fuzzer's complexity, and slow down the fuzzing process. The current fuzzer's implementation does not support these server-side measures and hence cannot effectively analyze REST APIs using them.

An even more restrictive approach involves limiting API access to trusted devices only, verified through remote attestation. This can be implemented by requiring messages to be signed with hardware-protected, cryptographic keys stored in a trusted execution environment (TEE) [Hei+21, p. 233]. Using predefined interfaces, these signing keys are only utilized for use cases allowed by the platform vendor. Key extraction is very hard since those keys never leave the TEE, and TEEs are specifically guarded against physical attacks [APS, pp. 9 sqq.]. For example, Apple implements this approach for some of their HTTP-based APIs used on their operating systems [Hei+21, p. 233]. In summary, APIs requiring remote attestation are hard to fuzz since messages are only accepted when properly signed by a trusted device. This restriction can make fuzzing and, as a consequence, the thesis' approach completely unviable for such APIs.

1. https://frida.re

Other than traffic capturing and fuzzing-related limitations, the current method's implementation only considers a subset of API behavior: In the static analysis step, one common pattern of producer-consumer dependencies is considered, however, REST APIs may implement more patterns and those are currently unsupported. A larger number of REST APIs should be evaluated in order to gain knowledge about other common patterns. Moreover, the algorithm for the parameter classification problem (appearing in multiple places in this thesis) may find false positives when static path segments have an extraordinarily high entropy, resulting in them being wrongly treated as variable path segments. The same issue can also happen for variable path segments containing a concrete value with low entropy (included in the traffic flows), in which case the variable is classified as static path segment.

Regarding the evaluation algorithm, an official OpenAPI spec can only be treated as an approximation of the API implementation. As seen in the evaluation, official specs must not necessarily document the actual API implementation, but rather a desirable API design. Due to spec conflicts, this discrepancy may result in (slightly) falsified metrics for the reverse-engineered specs because they list the actually implemented behavior and not a theoretical API design. This issue mostly affects specific parts of OpenAPI specs, e.g. parameter types. Additionally, some official specs might not even cover all API paths. The evaluation approach ignores such unmapped paths and therefore has no way of rewarding discovery of undocumented, or hidden, API paths. Furthermore, increasing path coverage during traffic capturing can be an exhaustive process, depending on API complexity. It may be more effective to automate this crawling process in order to maximize path coverage, but this poses another problem.

8 Conclusion

This thesis aims to investigate the effectivity of reverse-engineering REST APIs through fuzzing, starting with a base OpenAPI spec and resulting in an enriched spec. Additionally, it is analyzed whether reverse-engineered, enriched specs help in finding more bugs and problems in REST APIs. Main contributions of this thesis are (i) the creation of a static traffic analysis algorithm that can infer producer-consumer dependencies between API endpoints, (ii) development of new fuzzing techniques that find authentication requirements, parameter patterns and other interesting API characteristics, as well as (iii) the establishment of a novel approach that evaluates OpenAPI specs regarding reverse-engineered information gain.

The evaluation demonstrates that the base specs, created by the current state of the art, already score relatively high numbers with an average of 73.5% of reconstructed information from the official specs for the evaluated applications. Enrichment through static traffic analysis and fuzzing further improves results, with enriched specs containing an average of 80.3% of reconstructed information compared to the official specs, which is an increase of 9.3% over the base specs. This boost showcases the potential of regaining information through additional, fuzzing-based reverse engineering. Furthermore, enriched specs contain more information that is completely absent from official specs (8.9%), compared to base specs (5.2%), which marks a significant increase of 71.2%. This suggests a high potential for discovering API characteristics that are not documented in official OpenAPI specs.

The second part of the evaluation reveals that using Microsoft's REST fuzzer, RESTler, with enriched specs does not expose new bugs or problems automatically compared to base specs. However, the evaluation indicates multiple potential issues and quirks in target applications, which are not automatically discovered due to current limitations in REST fuzzers. For instance, despite being included in enriched specs, RESTler fails to utilize authentication requirements from OpenAPI specs. Through a semi-automatic approach, problematic behavior is found in one target application's authentication logic, potentially indicating authentication-related vulnerabilities. The use of other enriched information, such as regex-conform value generation based on OpenAPI's *pattern* key, is also unsupported by RESTler, but could lead to better understanding of API constraints and more efficient fuzzing.

In conclusion, reverse-engineering REST APIs through fuzzing with enrichment of OpenAPI specs leads to better understanding of undocumented or poorly documented REST APIs. The potential of finding more bugs and vulnerabilities through usage of the resulting enriched specs is shown, however, cannot be achieved automatically due to current limitations of exploitative REST fuzzers. Further development of REST fuzzers and their support for more functionality defined in the OpenAPI specification is required to make the process fully automatic and potentially more effective.

Future work In addition to the explained problems in current exploitative REST fuzzers, which are out of scope for this thesis, there are mainly three open problems identified for the proposed methods: (i) More fuzzing targets to explore REST API characteristics, (ii) better

understanding of request body constraints, and (iii) discovery and support of more dependency patterns, both for fuzzing and static traffic analysis.

For (i), there are certainly more fuzzing possibilities to reverse-engineer API characteristics which remain to be explored. The thesis proposes some fuzzing techniques with specific targets, like security requirements, but other areas may be uncovered.

(ii) While connected to the first open problem, better understanding of request body constraints can be seen as a specific subproblem: It is hard for exploitative REST fuzzers to generate valid JSON request bodies that adhere to the API's constraints. As seen in the evaluation, this can restrict dependency graphs from being resolved correctly, and therefore deeper service states are kept uncovered. Currently, the thesis proposes a method for better understanding of HTTP query parameters through use of a language learning algorithm. It would be useful to extend that approach to request bodies, so that each request body property receives an own regular expression, enabling generation of acceptable fuzzing values. As OpenAPI's *pattern* field only supports OpenAPI parameters, e.g. queries or headers, this probably requires definition via a custom spec extension. Exploitative REST fuzzers need to support both that possible spec extension as well as pattern-conform value generation in order to effectively generate valid request bodies.

(iii) REST APIs make use of different dependency patterns. These describe the exact relationship between multiple API endpoints, and how they are connected to each other. The thesis' fuzzer implements one such pattern, and it can be extended by supporting more of them. Some are discovered in the evaluation, but there are probably more to be found in other APIs. Additionally, the usage of HTTP status codes differs between REST APIs. Since the proposed static traffic analysis currently detects producer endpoints through their usage of HTTP response status code 201, non-compliant producers (i.e. those using other status codes) remain undetected. It could be worthwile to explore more possibilities to detect such producers, e.g. by searching for POST requests with a successful response status code not equal to 201 (assuming the POST method always creates resources).

All of this could further improve and extend the proposed methods in meaningful ways, and remains open for future work.

References

[AAB21] Aseel Alsaedi, Abeer Alhuzali, and Omaimah Bamasag. *Black-box Fuzzing Approaches to Secure Web Applications: Survey*. In: *International Journal of Advanced Computer Science and Applications* 12.5 (2021).

[AAB22] Aseel Alsaedi, Abeer Alhuzali, and Omaimah Bamasag. *Effective and scalable black-box fuzzing approach for modern web applications*. In: *Journal of King Saud University-Computer and Information Sciences* 34.10 (2022), pp. 10068–10078.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[AFL] *american fuzzy lop*. URL: https://github.com/google/AFL (visited on 06/19/2024).

[AGP19] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. *RESTler: Stateful REST API Fuzzing*. In: *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 748–758.

[AGP20] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. *Checking Security Properties of Cloud Service REST APIs*. In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2020, pp. 387–397.

[Ang87] Dana Angluin. *Learning Regular Sets from Queries and Counterexamples*. In: *Information and computation* 75.2 (1987), pp. 87–106.

[APS] *Apple Platform Security*. URL: https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf (visited on 10/28/2024).

[AWS24] *What is a RESTful API?* URL: https://aws.amazon.com/what-is/restful-api/ (visited on 03/19/2024).

[Bas+17] Osbert Bastani et al. *Synthesizing Program Input Grammars*. In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 95–110.

[BB] *Bunnybook*. URL: https://github.com/pietrobassi/bunnybook (visited on 10/15/2024).

[CATF] *OpenAPI Formats*. URL: https://endava.github.io/cats/docs/getting-started/openapi-formats (visited on 10/24/2024).

[CATS] *CATS*. URL: https://github.com/endava/cats (visited on 10/08/2024).

[Cro03] Scott Crosby. *Denial of Service through Regular Expressions*. In: Washington, D.C.: USENIX Association, Aug. 2003.

[DAT24] I Putu Arya Dharmaadi, Elias Athanasopoulos, and Fatih Turkmen. *Fuzzing Frameworks for Server-side Web Applications: A Survey*. In: *arXiv preprint arXiv:2406.03208* (2024).

[Duc+18] Julien Duchêne et al. *State of the art of network protocol reverse engineering tools*. In: *Journal of Computer Virology and Hacking Techniques* 14 (2018), pp. 53–68.

[EWL] *words_alpha.txt*. URL: https://raw.githubusercontent.com/dwyl/english-words/a77cb15f4f5beb59c15b945f2415328a6b33c3b0/words_alpha.txt (visited on 06/11/2024).

[FFUF] *ffuf - Fuzz Faster U Fool*. URL: https://github.com/ffuf/ffuf (visited on 10/08/2024).

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.

[Gau+21] François Gauthier et al. *BackREST: A Model-Based Feedback-Driven Greybox Fuzzer for Web Applications*. In: *arXiv preprint arXiv:2108.08455* (2021).

[GCA] *Why and when to use API keys*. URL: https://cloud.google.com/endpoints/docs/openapi/when-why-api-key (visited on 03/21/2024).

[GL24] *GitLab OpenAPI specification*. URL: https://gitlab.com/gitlab-org/gitlab/-/blob/46415b3efe503e6dd5b719e0a3f90e71ae5273f2/doc/api/openapi/openapi.yaml?plain=1 (visited on 03/19/2024).

[GPS17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. *Learn&fuzz: Machine Learning for Input Fuzzing*. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 50–59.

[Hei+21] Alexander Heinrich et al. *Who Can Find My Devices? Security and Privacy of Apple's Crowd-Sourced Bluetooth Location Tracking System*. In: *Proceedings on Privacy Enhancing Technologies*. 2021, pp. 227–245.

[Hua+22] Yuyao Huang et al. *Protocol Reverse-Engineering Methods and Tools: A Survey*. In: *Computer Communications* 182 (2022), pp. 238–254.

[Lia+18] Hongliang Liang et al. *Fuzzing: State of the Art*. In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 1199–1218.

[M2SP] *mitmproxy2swagger.py*. URL: https://github.com/alufers/mitmproxy2swagger/blob/70aa772d6cd3684aaf2c5b03d0bd81064f327f88/mitmproxy2swagger/mitmproxy2swagger.py#L94 (visited on 11/20/2024).

[MP2S] *mitmproxy2swagger*. URL: https://github.com/alufers/mitmproxy2swagger (visited on 11/18/2024).

[MTTR] *Mattermost*. URL: https://github.com/mattermost/mattermost (visited on 10/10/2024).

[Mur96] Allan H. Murphy. *The Finley Affair: A Signal Event in the History of Forecast Verification*. In: *Weather and Forecasting* 11.1 (1996), pp. 3–20.

[OABA] *Bearer Authentication*. URL: https://swagger.io/docs/specification/v3_0/authentication/bearer-authentication/ (visited on 10/21/2024).

[OAD24] *Data Types*. URL: https://swagger.io/docs/specification/data-models/data-types/ (visited on 08/13/2024).

[OAI] *OpenAPI Initiative*. URL: https://www.openapis.org/participate/how-to-contribute/governance (visited on 10/08/2024).

[OAL24] *Links*. URL: https://swagger.io/specification/links/ (visited on 07/29/2024).

[OAS24] *OpenAPI Specification*. URL: https://swagger.io/specification/ (visited on 03/19/2024).

[OASP] *OWASP API Security Project*. URL: https://owasp.org/www-project-api-security/ (visited on 03/21/2024).

[OFFAT] *OWASP OFFAT*. URL: https://github.com/OWASP/OFFAT (visited on 10/08/2024).

[OPTI] *Optic - the CI tool that improves your APIs*. URL: https://github.com/opticdev/optic (visited on 11/18/2024).

[OWA24] *REST Security Cheat Sheet*. URL: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html (visited on 01/24/2024).

[OWABA] *API2:2023 Broken Authentication*. URL: https://owasp.org/API-Security/editions/2023/en/0xa2-broken-authentication/ (visited on 03/21/2024).

[OWADE] *API3:2019 Excessive Data Exposure*. URL: https://owasp.org/API-Security/editions/2019/en/0xa3-excessive-data-exposure/ (visited on 03/26/2024).

[OWAIN] *API8:2019 Injection*. URL: https://owasp.org/API-Security/editions/2019/en/0xa8-injection/ (visited on 03/22/2024).

[OWARL] *API4:2019 Lack of Resources & Rate Limiting*. URL: https://owasp.org/API-Security/editions/2019/en/0xa4-lack-of-resources-and-rate-limiting/ (visited on 03/22/2024).

[OWUI] *Open WebUI*. URL: https://github.com/open-webui/open-webui (visited on 10/10/2024).

[REAN] *RESTler annotations*. URL: https://github.com/microsoft/restler-fuzzer/blob/694cc9e63dc61b372ddab5c6648223e4329ee6fa/docs/user-guide/Annotations.md (visited on 07/04/2024).

[RFC6750] *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. URL: https://www.rfc-editor.org/rfc/rfc6750 (visited on 07/16/2024).

[RFC6901] *JavaScript Object Notation (JSON) Pointer*. URL: https://www.rfc-editor.org/rfc/rfc6901 (visited on 07/29/2024).

[RFC9110] *RFC 9110 HTTP Semantics*. URL: https://www.rfc-editor.org/rfc/rfc9110 (visited on 07/05/2024).

[RFC9421] *RFC 9421 HTTP Message Signatures*. URL: https://www.rfc-editor.org/rfc/rfc9421 (visited on 10/28/2024).

[RLH] *RateLimit header fields for HTTP*. URL: https://www.ietf.org/archive/id/draft-ietf-httpapi-ratelimit-headers-07.html (visited on 07/16/2024).

[S3] *Authenticating Requests (AWS Signature Version 4)*. URL: https://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html (visited on 10/28/2024).

[ZA23] Man Zhang and Andrea Arcuri. *Open Problems in Fuzzing RESTful APIs: A Comparison of Tools*. In: *ACM Transactions on Software Engineering and Methodology*. Vol. 32. 6. ACM New York, NY, 2023, pp. 1–45.

Versicherung an Eides statt

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen - benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe.

Hamburg, 25. November 2024

Einstellung in die Bibliothek des Fachbereichs Informatik

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, 25. November 2024