ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
FACULTY OF INFORMATICS

THESIS WORK

DIPLOMA WORK

OE-NIK	Student's name:	**Tamás György Pozsonyi**
2024	Student's registration number:	**T/008236/FI12904/N**

Óbuda University
John von Neumann Faculty of Informatics
Institute for Cyber-physical Systems

THESIS

Name of the student: **Tamás György Pozsonyi**
Registration number: T/008236/EI12904/N
Neptun's code: [redacted]

Title of the thesis:

Network Protocol Testing Toolkit Development
Hálózati Protokol Tesztelő Szoftver Csomag Fejlesztés

Internal supervisor: Dr. Eszter Balázsné Kail
External supervisor: Dániel Lakatos

Deadline: 15 December 2024

Subjects of the final exam: Computer Architectures
Cloud service technologies and IT security specialization

The task:

Nowadays with the rapid advancements in the inner networks of vehicles, there is an increasing need to have efficient ways to check them for vulnerabilities. Protocol fuzzing is a good way to attack an ECU, mimicking real life data, and look for crashes, freezes or erratic behavior. There is an in-house software already for that at Bosch, but it is unrefined and is lacking in usability.
The goal of this thesis is to get familiar with protocol fuzzing as a concept, how it works and to further develop this software. During the development it is important to enhance the software and enhance the usage experience, the ideal output is to make it very easy and comfortable to use by utilizing a modern UI and other helpful features. The enhanced application should be evaluated based on operability, performance, security and user satisfaction.

The thesis must contain:

- investigation into the potential technologies and techniques to use (UI design, multi-threading, logging of large amounts of data),
- comparison of already existing solutions' performance,
- design and development of new updated functions and features,
- defining the test environment, and examining the realized system with pre-defined test-cases, comparing its performance and usability with the old system,
- evaluating the test results based on pre-defined parameters,
- discussion about future development possibilities.

Place of stamp

Fleiner R.
.....
Dr. habil. Rita Fleiner
head of institute

This specification is valid until: **15 December 2026**
ÓE HKR Section 54, paragraph (10)

I consider the thesis suitable for submission:

.....
external supervisor

Kell
.....
internal supervisor

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Faculty of Informatics

STUDENT'S DECLARATION

I the undersigned student hereby declare that this thesis work/ diploma work is the result of my own work; I have disclosed the references and tools used in an identifiable manner. The results indicated in my thesis work/ diploma work completed may be used by the university and the institution announcing the task for their own purposes free of charge.

Dated: Budapest, 12. 14. 20..

.....
student's signature

CONSULTATION LOG

Student's name: Pozsonyi Tamás

Neptun code: [redacted]

Specialty: Cloud service technologies and IT security specialization

Phone number: [redacted]

Mailing address (e.g. domicile): [redacted]

Thesis work / diploma work[1] title in Hungarian:

Hálózati Protokol Tesztelő Szoftver Csomag Fejlesztés

Thesis work / diploma work[2] title in English:

Network Protocol Testing Toolkit Development

Internal supervisor: Balázsné Dr. Kail Eszter

External supervisor: Dániel Lakatos

Please write the data in printed capital letters!

Occ.	Date	Content	Signature
1.	2024.05.01.	Thesis structure planning	Kail E
2.	2024.05.04.	Topic discussion and specification	Kail E
3.	2024.05.08.	Thesis revision, discussion of citing of sources and referencing figures	Kail E
4.	2024.05.15.	Thesis final discussion	Kail E

The Consultation log is required to be countersigned by either supervisor on a total of 4 occasions of consultation.

The student has complied with the requirements of the subject titled Thesis I / Thesis II (BSc) or Thesis work 1 / Thesis work 2 / Thesis work 3 / Thesis work 4[3], (MSc) and is allowed to proceed for reporting / defense[4].

Budapest, 2024.05.16

Kail E

Internal supervisor

[1] Underline as appropriate
[2] Underline as appropriate
[3] Underline as appropriate
[4] Underline as appropriate

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

John von Neumann Faculty of Informatics

Log of consultations

Name of the student: Pozsonyi Tamás ..... security specialization .....

Neptun code: [redacted]

Program: Cloud service technologies and IT

Phone number: [redacted]

Address: [redacted]

Title of the thesis work:

Network Protocol Testing Toolkit Development .....

Internal supervisor: Balázsné Dr. Kail Eszter.....

External supervisor: Dániel Lakatos .....

Please use BLOCK CAPITAL LETTERS to fill in this form!

Ocas-sion	Date	Topics discussed	Signature
1.	2024.11.20.	Implementation Discussion	Kail E
2.	2024.11.29.	Implementation Review	Kail E
3.	2024.12.07.	Citation and figures overview	Kail E
4.	2024.12.12.	General overview	Kail E

This log should be signed on each consultation (by a supervisor), 4 times altogether.

The student fulfilled the requirements of the Thesis work subject, I allow his/her participation on the presentation/defence[1].

Kail E

Internal supervisor

Budapest, 2024.12.10

[1] Underline the appropriate one.

ABSTRACT

Nowadays, with the rapid advancement of Electric Vehicles (EV) and Internal Combustion Engine Vehicles (ICEV), for the sake of this thesis, mainly in the area of internal electronics and communication networks and protocol of vehicles, there is an increasing need for efficient ways of running tests looking for vulnerabilities. Modern automotive cyber security standards, like the ISO/SAE 21434 [1], that is intending to specify engineering requirements for cybersecurity risk management are also pushing for higher testing and safety standards. One way this ISO recommends Original Equipment Manufacturers (OEM) to test Electronic Control Units (ECU) is Protocol Fuzzing. This tool can show you how an ECU would react to receiving a wide range of different type and amount of data akin to a real life communication and can reveal vulnerabilities arising from edge cases, flooding and damaged or malformed packets. After these vulnerabilities have been discovered and documented, it can be given to the client, the manufacturer who ordered the tests, as useful feedback for further development.
This thesis is mainly focused on expanding the internal Automotive Protocol Fuzzing software, later referred to as Fuzzer, developed at Robert Bosch GmbH. (Bosch) as this application is in it's early stage of development. Goals for this thesis include implementing an extensive Web User Interface (UI), attaching redundant storage in the form of a single or multiple databases to store large amounts of test data, upgrading the report creation functionality and, explore and implement, if feasible, multi-threading or asynchronous solutions.
The main goal for this application is to create a robust tool that can be used for testing automotive ECUs by internal teams at Robert Bosch GmbH. and potentially customers and or clients of Robert Bosch GmbH. Requirements for the tool is that it has to be able to handle multiple different protocols, run and change test cases and project configurations created by the user and provide reports dynamically generated from selected data slices.

ABSZTRAKT

Manapság, az elektromos és belsőégésű járművek rohamos fejlődésével, a szakdolgozat témája miatt, főként a belső elektronika és kommunikációs hálózatok és protokollok terén, folyamatosan növekszik az igény hatékonyabb tesztelési eszközökre, melyekkel sérülékenységeket lehet feltárni. Modern autóipari kiberbiztonsági szabályzások, mint a ISO/SAE 21434 [1], melyek mérnöki oldalról próbálják pontosítani az elvárásokat a kiberbiztonság kockázatkezelés terén, ezenfelül szintén dolgoznak a tesztelési és biztonsági színvonal emelésén a gyártóknál. Az ISO egyik javaslata az alkatrész gyártóknak (OEM) arra, hogyan teszteljék az elektromos vezérlő egységeiket (ECU), a protokoll penetráció tesztelő, másnéven Protocol Fuzzer. Ez az tesztelő eszköz meg tudja mutatni, hogyan reagálna a tesztelés alatt álló ECU különböző mennyiségű és fajtájú in-

gerekre, melyek érhetik a való világban is. Ezt a módszert használva, lehet találni olyan sérülékenységeket, melyek szokásos tesztelés alatt nem, csak szélsőséges vagy különleges esetben jöttek volna elő. Ilyen problémák lehetnek például ritkán előforduló helyzetek, melyek szélsőséges a szokásostól jelentősen eltérő adatokat generálnak az autóban. Ez lehet egy külső támadás, ahol elárasztják a kommunikációs hálózatot üzenetekkel így megbénítva a csatornát vagy valami miatt sérült, hiányos üzenetek érkezése az egyik ECU-hoz. A tesztelés során felfedezett sérülékenységeket a gyártók, akik rendelték a teszteket, ki tudják javítani és biztonságosabb járműveket gyártani.

Ez a szakdolgozat főként egy céges belső fejlesztésű autóipari protokoll tesztelő szoftver tovább fejlesztésére és kiegészítésére fókuszál. Ez a szoftver, későbbieknek csak Fuzzer, a Rovert Bosch Kft.-n belül a csapatom fejlesztése alatt áll és még korai fázisban jár a fejlesztése. A szakdolgozat céljaiba tartozik egy könnyen kezelhető részletes grafikus kezelési felület kiépítése, redundáns tárolóhely hozzá kapcsolása egy vagy több adatbázis formájában, melyekben nagy mennyiségű tesztelési adatot fogok tárolni, ezeken túl fejleszteni a kimutatás készítési és adat áttekintési képességeit az eszköznek. Potenciálisan felmérni, kipróbálni és ha lehetséges implementálni több-szálas és vagy aszinkron programozási megoldásokat.

A fő célom ezzel az applikációval, hogy létrehozzak könnyen használható, nagy teljesítményű eszközt, mely képes segíteni és megkönnyíteni a munkáját mindenkinek, akik a csapatban ezzel dolgoznak. Potenciális később pedig majd árulni egy teljesértékű termékként a Bosch-on belüli másik csapatoknak és klienseknek. Elvárások az eszköz felé, hogy képes legyen kezelni több különböző protokollt, tudjon futtatni és megváltoztatni tesztelési eseteket, projekt konfigurációkat, melyeket a felhasználók készítettek és tudjon dinamikusan generálni kimutatásokat a kiválasztott adatok alapján.

CONTENTS

1 CYBERSECURITY

Nowadays, with the IoT revolution changing every aspect of our life, while making everything "smarter", it also brings certain dangers. Putting embedded devices, that collect and exchange data through the internet, into every vehicle, household appliance and watch just to name a few things, means that these devices can become compromised. In the process of making everything smarter and connected to internet, we have created completely new surfaces that can be attacked by malicious third-parties. The need for more secure computer systems is higher then ever, especially now that it has almost become an everyday occurrence to see some kind of company or government institute get hacked by individual hackers, hacker groups or even state sponsored groups.

One such incident was the Mirai botnet[2]. In this case, mainly embedded and IoT devices, at the peek around 600K of them, were infected by a new Distributed Denial-of-Service(DDoS) botnet called Mirai.
Firstly, what is a DDoS botnet, it is a number of different, independent devices infected by the same malware and come under the control of a malicious third-party. These devices might not even show any signs of infection, as there are multiple levels of visibility for these malware. Some are designed in a way where they take over the whole device and some just hide and wait for instructions coming from a bot herder or a Command-and-Control Center(CnC). Once a target or targets have been set by the attacker, all of the infected devices, like zombies start sending requests to the target's IP address to potentially cause the server or network to become overwhelmed.
This botnet was first used in September of 2016 to attack security blog Krebs, that was hosted by Akamai Technologies. At the time, this attack reported to have been 665Gbps in volume, was record breaking among the known attacks, and while being unsuccessful, it still resulted in the site being taken down[3]. The attack was done by a network of enslaved IoT devices, including home security appliances like surveillance cameras and smart home systems. Included in the reasoning why this large scale attack happen were non-existent or poor security practices.

A big problem coming from technologies advancing too quickly is that companies have to try to keep up and adapt to be able to capitalize on new trends. This rush also means that not many of the have the time and resources to develop security testing procedures and protocols. A solution to this problem could be the use of fuzzing.

1.1 Fuzzing

Fuzzing is a major tool for bug discovery, it is an automated technique designed to find exploitable inputs and vulnerabilities. It is used in wide range of different industries like Web application and API testing, embedded systems security testing and automotive security testing. As new programs become more complex fuzzing, as an inexpensive tool, has become an easy choice for bug detection. There are three main parts of a fuzzer as you can see in Figure 1.1[4]

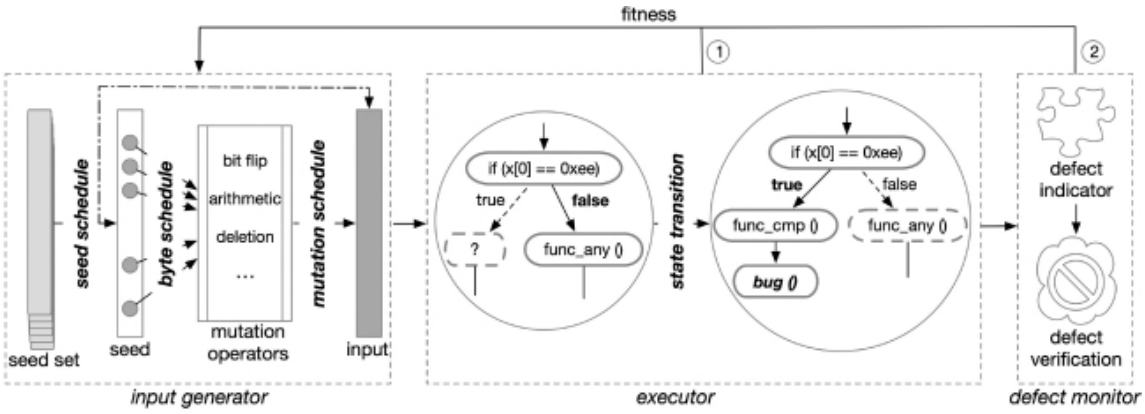

Figure 1.1: General Workflow of Fuzzing[4]

1.1.1 Input Generator

The Input Generator is responsible for creating an input for the Executor. In terms of synthesizing inputs there are two approaches, mutation-based and generation-based fuzzing[5]. Mutation-based is the easier approach as it doesn't require expert knowledge of the validation of valid inputs. Mutation-based fuzzing modifies or mutates one of the existing or previously created valid inputs that are likely to cause vulnerabilities or bugs to emerge. On the other hand generation-based fuzzing requires knowledge about the input verification to create a guideline with witch new valid inputs can be synthesized. Generative AI can be used in both cases to synthesize new inputs and identify valid test cases. New inputs are mutated constantly in an infinite loop and are provided to the Executor.

1.1.2 Executor

The Executor runs the target program with the provided valid inputs. Based on the information gained during the execution phase, there three types of implementation for a fuzzer. Firstly there is the Black-Box fuzzing, here the fuzzer has no knowledge of the

code base and can't optimize the fuzzing process using input formats or output states. This can result in low code coverage. Next there is White-Box fuzzing that obtains all the information about the internal state of the target program and can optimize the fuzzing to systematically target different areas. It can captures and analyses the structure of the target program and can use a constraint solver to optimize the fuzzing. And lastly there is Grey-Box fuzzing. It has some knowledge about the structure of the target program, but not about the inner workings. The way to gain this knowledge is called instrumentation. Instrumentation[6] in this context means that the fuzzer inserts different functions on clearly defined positions into the compiled target program that measures code coverage. With this the fuzzer know which paths are taken for each run with a given input and can use this knowledge to optimize the fuzzing.

1.1.3 Defect Monitor

The Defect Monitor based on the observed behavior and gained information, provides feedback for the Input Generation thus restarting the cycle. This part is responsible for recognizing and identifying if a software defect or a bug has been triggered, this could be crash detection if the target program has crashed because of the tested input, output validation problem meaning the expected output for a given input is different than the actual output that was produced, if the internal state can be monitored than an unexpected state change could also mean a bug.
Choosing the right Defect Monitor is also very important, as this is the main component for identifying bugs or defect in the target program. Using the correct one can significantly enhance the effectiveness of the Fuzz testing. There are multiple factors to consider when choosing the Defect Monitor, these can include the complexity of the program, for example if you have a very basic program where you are only checking to see if it can be crashed by an unexpected input combination or variation, a simple crash detection might be good enough. Another thing to consider is the type of bugs being targeted, for example, if looking for memory leaks, the a memory usage temperature monitor could help speed up the process.

2 THE PROJECT

2.1 Introduction

In the current world development of technologies is moving really fast. This is also true for the automotive industry, where manufacturers are creating and implementing new technologies to enhance the driving experience and security of vehicles. These new technologies need rigorous testing before being implemented and sold in new vehicles to the general public. Having easy-to-use comprehensive tools for testing can make the testing process simpler and faster.
Robert Bosch GmbH. is one of the biggest company dealing with automotive electronics testing. The Fuzzer mentioned in this thesis is one of the tools they utilize while testing the Electronic Control Units found in modern cars. Upgrading this tool is essential in making security testing faster and more efficient.

2.2 Project Description

This project topic is about upgrading a developmental tool into a close to production ready one. In the next few sections all the necessary and planned expansion and improvements will be mentioned and explained, furthermore the motivations and goal will also be touched upon. The program to expand is an automotive protocol fuzzer developed at Bosch, currently for internal use only. The fuzzer won't be changed in the sense of changing the fuzzing logic or the overall inner workings of the main algorithm, rather the output logging, management, along with the general use and control of the whole program and the report generating functionality.

2.3 Motivation

The current version of the program is working well and there is nothing to change currently in regards to the fuzzing, but the handling of the program is unrefined and lacking, as it is still in development. One of the main problems with the current version is that it is hard to use. In its current state, it is a command line tool and does not have a comprehensive Graphical User Interface(GUI). This is a problem for more than one reason, firstly someone outside of the developers will have a hard time trying to figure out how to set it up and start it, let alone create test cases and define new protocols. Another problem is that, in its current state, it is saving all the data into an HTML and or an XML file instead of something more reliable, like a database. This output management also causes a different issue, if someone needs test data generated with a given test case in a previous run, they would need to go through all the test files one-by-one instead of having everything a

central place, like a database. Once the project has moved closer to a final state and has been approved internally, it can be licensed to other projects and teams within Bosch or potentially outside as well.

2.4 Goals

The goals for this project are the following:

2.4.1 GUI

One of the first steps is to create web GUI for the program, because in its current form, a command line program is being used. Through this interface the user should be able to set the project and test cases being used for the current testing run, start and change the selected startup configuration, monitor the program while it's running, see the communication being sent back and forth and see and search in the collected data.

2.4.2 Logging

The Fuzzer is going to be communicating with the web application through the use of APIs and collect all the incoming and outgoing packets exchanged between the Fuzzer and the ECU. The data should be stored in a dedicated database running alongside the web application. This database should be able to keep up with the Fuzzer and not drop any of the messages. It also has to be able to provide data in response to queries with a reasonable response times even under load.

2.4.3 Reporting

Report generation is also very important, the goal is to be able to create a dynamic report based on the user's needs. A method has to be created for the user to interact with the database from the web GUI to see a preview of what the report would contain and create the report.

3 THE WEB APPLICATION

3.1 What is it

A web application is a software, that runs through a browser without any installation. It can deliver content dynamically and respond to user input. The logic for processing user input runs on the server in the background and communicates with the front-end through requests. These requests can be HTTP requests, REST API calls or WebSockets, and allow for real-time updates on the site. Web applications are dynamic versions of traditional static websites, and are commonly used today by big companies.
According to a publication [7], web applications nowadays are an essential part of every business. It is because of the growing complexity of services and applications that are provided to the users through the web, and as technologies are rapidly changing and evolving, the number of libraries and web frameworks needed to facilitate them also increases.

3.2 Framework

Before going into web frameworks, a software framework is a collection of code libraries, that provide a foundation for building software applications. Software frameworks provide benefits and functionalities that are very useful for development, like an abstraction layer between the programmer and the underlying complexities of the system, that provides a high level interface to work with, reusable code libraries for all the different functionalities needed, extensibility for new libraries from others or custom made ones to fit specific needs. Using a software framework can speed up the development of applications and can make them more structured and readable for future projects.

A web framework, according to [8], is a web oriented software framework and its goal is the facilitate the development of web applications and services. Web frameworks provide more libraries and functionalities needed for web application development, debugging and testing. According to [7], most of the modern frameworks are based on the separation of layers. One popular architecture is Model-View-Controller(MVC) pattern see Figure 3.1. It consists of 3 main components: the Controller, the Model and the View.

Firstly, the Controller acts as an intermediary between the View and the Model components, facilitating the communication and the data flow. It receives requests, that are often triggered by a user interacting with a template, generated by the View component. The received requests are then interpreted and translated into actions for the Model. This component might also perform some kind of data validation or manipulation. It is primar-

ily responsible for the application's business logic, that defines the core functionalities and rules, that control data processing. Retrieving the processed data from the Model and sending it to the View to update the UI is also the responsibility of the Controller.

Next is the Model component of the MVC design pattern. This component is responsible for the communicating with the database, representing the data structure and defining the data manipulation operations. It also performs data operations requested by the by the controller.

Finally, there is the View Component of MVC. This component is responsible for displaying the UI for the user to interact with, handling and sending user interaction to the controller for processing. It is also responsible for updating the data displayed on the UI based on the information received from the controller.

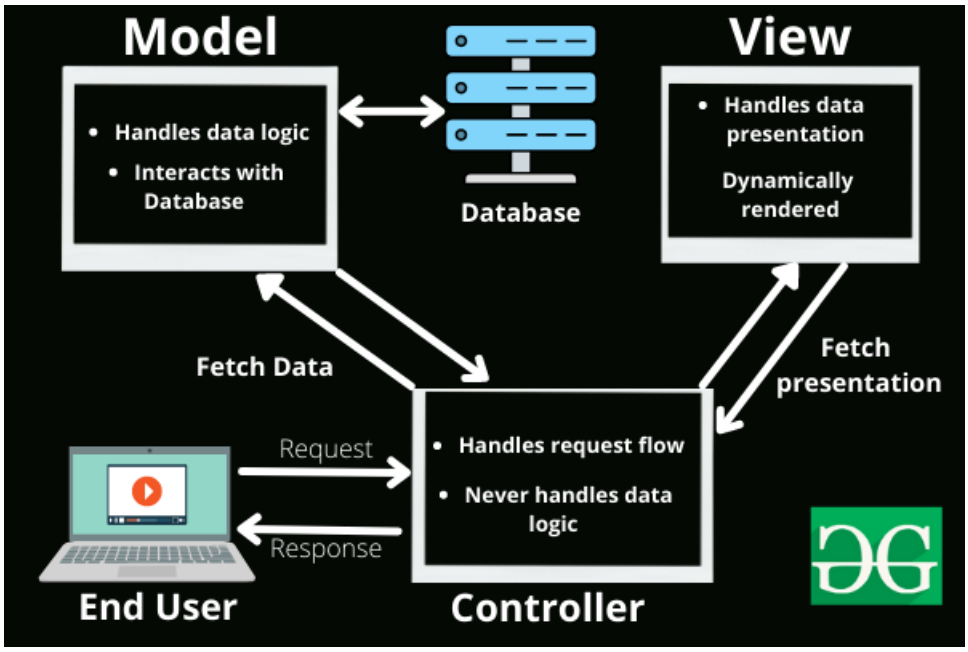

Figure 3.1: Model-View-Controller(MVC) example [9]

3.3 Related Work

There is a conference paper[10] by C. Arun, S. Priyenka, L. Sagarika and S. Sneha proposing a user friendly GUI using a Python based web application with the Django web framework. They are using HTML, CSS and JavaScript for the frontend, MySQL developed by Microsoft as the database to store and query the data, and Python based Django as

the web framework. They are aiming to make the data collection easy by letting trusted users, authorized using Django Authentication, to update the database, thus lessening the load on the administrator. For the reporting the paper proposes that the software is more sophisticated and efficient than spreadsheets and can use dynamic SQL queries to let the user fully customize the search and reporting parameters.

Conclusion

This paper[10] is similar in nature to my thesis, but there are some differences and problems. Firstly, it is using manual data submission by users, my thesis is focusing on connecting the Fuzzer to the web application through the use of APIs and having the data entry be automatic. Having dynamic queries for report generation is something is needed this project too, although the implementation is different. The goal for this project's implementation is to give options to the user based on the collected data, and based on those options and some input parameters, the user can create a report according to their needs.

4 TECHNOLOGIES

This project involves using multiple components. The next section contains the potential programs and technologies, their comparisons, descriptions and a conclusion on which one and why they were chosen for this project out of the other possibilities.

4.1 Python Web Framework

Starting out this project the first step is to find an appropriate Python web framework. What is needed for now is a framework that is not overly complex and very light, so it is easy to get familiar with it and start building the application. Another viewpoint to consider is that it also needs to be easily expandable for the future if a new idea for a feature comes up. Speed is also very important, but it won't be pushed to the limit as the fuzzer application is communicating with the ECUs over an outside communication channel and it needs to wait for the packets to arrive to the ECU, for the ECU to process the input and send a response back, so that the new packet with the next input doesn't get mixed up with the old input's response. The fuzzer also needs to wait for the web application to process and send the data to the database. Only when the fuzzer got the OK signal for the web application can the test continue with the next input.

4.1.1 Flask

Flask is a small, lightweight web framework that is based on Werkzeug, which is a collection of libraries used to create a Web Server Gateway Interface(WSGI), Jinja2, which is template engine, that can be used to create dynamic web pages by placing python-like expressions inside the HTML template. Through the use of these expressions, the page can access and use data injected by the backend. It is easy to learn, easy to use and very extensible. Flask is also one of the most popular framework next to Django. According to the Python Developers Survey 2022 [11], in 2022 Flask and Django were both at 39% usage. It is important to note for this survey, that the results were obtained through the use of multiple-choice questions. Flask in itself doesn't offer many built-in features, but it has a lot of third-party extensions built for it for all the different functionalities you might need. This includes databases, form and input validation, communication between the server and client side and the database [12].

Werkzeug

Werkzeug is Python library that contains tools and utilities that useful for building web frameworks and applications. Its core functionalities are request processing, response

handling, URL routing, middleware, HTTP utilities and exception handling. This library provides the development server for Flask and is responsible for serving the static files from the dedicated static folder. Static files are files that don't need to change and can be delivered to the user as they are stored on the server. These can include CSS files that dictate the visual of the current page, JavaScript files that give functionality for the current page and images and videos that are displayed on the page [13].

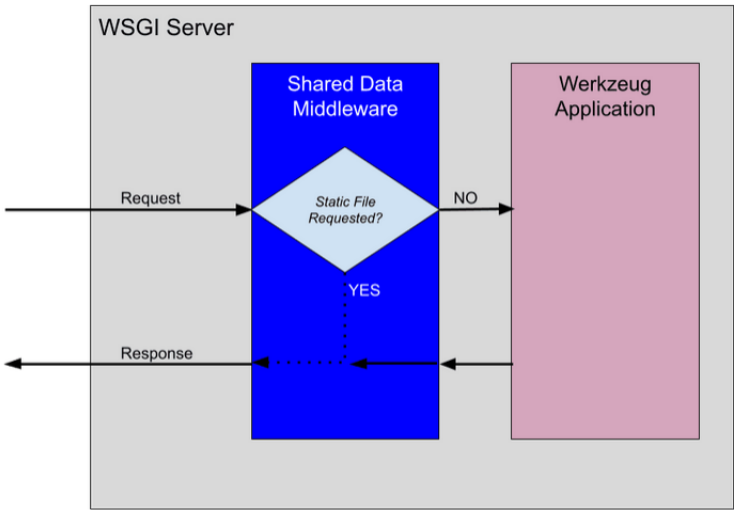

Figure 4.1: Werkzeug request handling

Jinja2

Jinja2 is a powerful templating language created for Python that was made to be like the Django template language. It is the default templating engine used by Flask and Flask has built-in function and configurations for using Jinja2 templates.
Jinja2 has some very beneficial features like inheritance. With inheritance, a base HTML template can be created, that could have the navigation bar, here can be included a site wide formatting with only one CSS file or a site wide JavaScript script with only one JavaScript file, widgets or logic that can be accessed from every page and local JavaScript code snippets that should be available from every page. Once the base HTML template has been completed, the HTML templates for the other pages can be created by first using the *extends* keyword and a string stating the name of the base file you want to extend see Figure 4.2. The page with the *extends* keyword will be loaded in together with the page it extends, like the code written between the *block content* and *endblock* keywords was originally written on the base page between the the same keyword pair seen in Figure 4.3.

```
{% extends "base.html" %}

{% block content %}
<br>
<br>
<br>
<h1><a>This is a test home page for the webinterface</a></h1>

{% endblock %}
```

Figure 4.2: Jinja Base Template Extension

```
{% block content %}
{% endblock %}
```

Figure 4.3: Jinja Base Template Content Block

Another useful feature of Jinja2 is the include statement that allows for importing other templates into the current one. The way it works is that template "shards" or in other words smaller incomplete HTML templates can be created and than used in a loop to display dynamic data. For example as seen in Figure 4.4, a variable with the name of "pag" has been passed to the template from the Flask application, with the help of Jinja2, and that variable which, in this case is an array, can be iterated through with a for loop inside of the HTML template and with the *include* keyword that many template "shard" will appear in the screen. What is important is that the f variable that holds the current iteration's values passes those values down to the template "shard" and fills out the table with appropriate data. The table seen in Figure 4.5 is the content of the template '_item.html'.
On the Figure 4.5, an other Jinja2 feature can also be seen, the If Else statement. Jinja2 can use the If Else statement to dynamical change the HTML code of the page based on the passed down variable, in this case it checks f's event value and based on that, it changes the style and formatting of the table. The entry called "Name" in the table also only shows up if f's name value is not empty.

```
{% for f in pag %}
    {% include '_item.html' %}
{% endfor %}
```

Figure 4.4: Jinja For Loop with Include

```
<table id="itemboxes" {% if f.event == "Send CAN" %} style="outline: rgba(0, 2, 145, 0.918) solid 2px;" {% else %} style="outline:
    <tbody style="padding-bottom: 5px;">
        <tr valign="top">
            <td id="allitemslabel">Event: </td>
            <td id="allitemscontent">{{ f.event }}</td>
        </tr>
        <tr valign="top">
            <td id="allitemslabel">Frame: </td>
            <td id="allitemscontent">{{ f.frame }}</td>
        </tr>
        <tr valign="top">
            <td id="allitemslabel">Frame info: </td>
            <td id="allitemscontent">{{ f.frame_info }}</td>
        </tr>
        <tr valign="top">
            <td id="allitemslabel">Frame timestamp: </td>
            <td id="allitemscontent">{{ f.frame_timestamp }}</td>
        </tr>
        <tr valign="top">
            <td id="allitemslabel">Timestamp base: </td>
            <td id="allitemscontent">{{ f.timestamp_base }}</td>
        </tr>
        <tr valign="top">
            <td id="allitemslabel">Frame ID: </td>
            <td id="allitemscontent">{{ f.Frame_ID }}</td>
        </tr>
        {% if f.name %}
        <tr valign="top">
            <td id="allitemslabel">Name: </td>
            <td id="allitemscontent">{{ f.Name }}</td>
        </tr>
        {% endif %}
    </tbody>
</table>
```

Figure 4.5: Template "shard" with Jinja2 IF statement

4.1.2 Django

Django is a high-level framework, open source Python web framework that focuses on quick development, consistency. Django's "Design philosophies" [14] include "Loose coupling", meaning that different layers of the framework operate on a need to know bases, they only know about each other if necessary. "Less code", meaning Django should use as little code as possible to function normally. "Don't repeat yourself (DRY)", meaning everything, be it distinct data or concept should only be present at one place at a time. It is also using the MVC design pattern, but the Django developers call it Model-View-Template(MVT), and it is a bit specialized. The main difference lies in MVT having a Template component in place of MVC's View, and MVT moving the View component to the place of MVC's Controller component. There is a difference in function, as well in how to the user interaction is being handled. In MVT the Template component is for presentation while handling the data and user interactions falls on the View component. On the other hand, in MVC, the Controller component is handling the user interactions and updates the Model, which then updates the View. One other difference is that for Django, there is a dedicated file for explicitly mapping the view to URLs, while in frameworks like Flask, that use MVC, you can use a decorator above the function or View to set the route. Django has a built-in Object Relational Mapping(ORM) that lets you interact with databases like you would in Python.

Django template language

Django templating language is a powerful mini language that is built into the Django web framework. It is for defining the user-facing layer of the application. It allows the developer to mix Python code with HTML. It has features such as variables, with which it can lookup values from the context provided by the backend. Tags that can provide arbitrary logic in the rendering process, such as if statements of loops. There are also filters that can transform the values of tags and variables.

4.1.3 FastAPI

This is the third most popular web framework after Flask and Django according to the Python Developers Survey 2022 [11] with a usage rate of 25%, with a 4% rise from 2021. According to the aforementioned statistics, one in four Python developer participating in the survey, has experience with this framework. FastAPI is specifically designed for building modern, high performance Application Programming Interfaces(API). It has multiple features to help speed up the process of developing APIs, like automatic API documentation, that can automatically create an interactive documentation based on the underlying code see Figure 4.6. The documentation contains all of the endpoints with the request and response parameters, expected data types and the data validation based on the Python format strings [15].

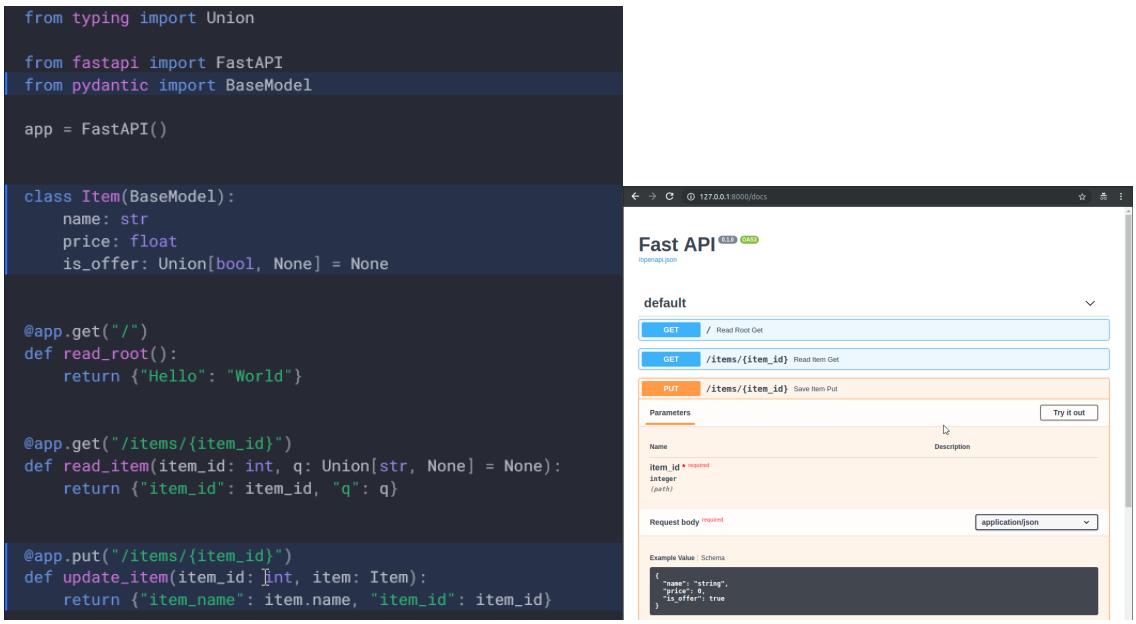

Figure 4.6: API code and automatic Documentation generated from it

Another feature to help API development is Type Hints. Type hints, also called Type Annotations are a basic language feature in Python that can specify the expected data types for variables, function arguments and return values [16]. Type Hints can also help speed up the development process as many Integrated Development Environments(IDE) and code editors can recognize and utilize Type Hints for code completion and type checking.
FastAPI also has built-in Cross-Site Request Forgery(CSRF) protection that is enabled by default. CSRF, also known as Session Riding, is an attack used on users, that have already been authenticated by the target site. This means that using the authenticated or "trusted" users credential, the site can be tricked into thinking, that it received a legitimate request, since the request was sent by an authenticated user. This kind of an attack can start by tricking the user with social engineering, like sending and email with a compromised link to executing some action that the attacker wants, like transfer funds or change an email address. A potential scenario could be that a user clicks on a link in an email, that has malicious code embedded in it, while also logged into their bank account in an other tab, opening the link could send a request to their bank account in with their credentials so the site wouldn't be able to identify the forged request [17].
FastAPI has an advantage over both Flask and Django in that it is using the newer Asynchronous Server Gateway Interface(ASGI) standard, that was designed for asynchronous programming, instead of the older Web Server Gateway Interface(WSGI) standard that the other two are using. This means that it is able to handle multiple requests at the same time because of its asynchronous operation. Being asynchronous enables FastAPI to handle more data faster and more efficiently. This efficiency is also coming from the Pydantic Integration. Pydantic is a powerful data validation and settings management library for Python. FastAPI uses this for efficient automatic data type checking and conversion, reducing the need for manually coded data processing and validation.

4.1.4 Conclusion

The cybersecurity and performance features of FastAPI are very useful, but are not necessary for this kind of application. As this web application will be used locally, inside of the company network without any publicly exposed ports, it is very unlikely to be targeted by attackers. This application doesn't contain any sensitive data or client information and has no way of doing any kind of damage, thus the security aspect is for future consideration. The performance aspect of FastAPI where it can deal with multiple request at the same time in parallel is very useful and might be needed in the future, but in the present there is going to be only one ECU being tested at a time thus the parallel processing is not needed yet. FastAPI would be ideal for when the project gets approved for full scale production.

Feature	FastAPI	Django	Flask
Focus	High-performance, asynchronous APIs	Full-featured web framework	Lightweight, flexible web framework
Learning Curve	Steeper (type hints, ASGI concepts)	Easier for beginners (familiar web concepts)	Easiest
Performance	Highest	Good	Good (depends on implementation)
Extensibility	Good (third-party libraries, plugins)	Good (large ecosystem of apps and libraries)	Good (third-party extensions)
Security Features	Built-in CSRF protection, JWT support	Requires additional security considerations	Requires attention to security
Ideal for	Scalable, performance-critical APIs	Complex web applications, CMS	Smaller web apps, prototypes, flexible projects

Figure 4.7: Web Framework Comparison

4.2 Database

After selecting Flask as the core of our web application the next step is to choose a database to store and retrieve the test data gained from running the test cases against the tested ECU. We need a lightweight, not very complex database that is easy to deploy, has low overhead and can accommodate the custom data structure needed for storing the test data.

4.2.1 SQLite3 with SQLAlchemy

Flask relies on Flask-SQLAlchemy, which is an extension for Flask, that adds support for a variety of relational database management systems. The base program called SQLAlchemy is an open-source Object-Relational Mapper(ORM) for Python that maps Python objects

to database tables. It also serves as an abstraction layer between the database and the code, thus allowing database-agnostic code to be written. Database-agnostic code means code that interacts with the database, but is not specific to any one database, rather can be used to interact with most of the supported database engines. Because Flask-SQLAlchemy is not database specific there are multiple different compatible databases that it can work with. I choose SQLite3 because it was easy to setup and good for small scale operations. It is lightweight and self-contained so I don't need to setup a separate server just a database file alongside my application.
With Flask-SQLAlchemy, a table structure can be defined by making a Python class, as can be seen in Figure 4.8. The Figure also show that the structure of data, after being queried, can also be easily managed with functions added to the class representing the data tables. Even the relationships to other tables like foreign keys can be declared as a line of code with the "sqlalchemy.orm.relationships()" command.

```python
class testrun(db.Model): #test runs
    __bind_key__ = 'fuzzkey'
    __tablename__ = 'testruntable'
    id: so.Mapped[int] = so.mapped_column(primary_key=True, autoincrement=True)
    run_id: so.Mapped[int] = so.mapped_column(index=True, nullable=False)
    project_id: so.Mapped[int] = so.mapped_column(nullable=False, index=True)
    testcase_id: so.Mapped[int] = so.mapped_column(nullable=False, index=True)
    stats: so.Mapped["stat"] = so.relationship(back_populates="testrun", uselist=False)

    messages: so.Mapped[List["fuzz"]] = so.relationship(back_populates="testrun")
    failmessages: so.Mapped[List["fuzzFail"]] = so.relationship(back_populates="testrun")

    def to_dict_no_messages(self):
        return {
            'id': self.id,
            'run_id': self.run_id,
            'project_id': self.project_id,
            'testcase_id': self.testcase_id,
            'stats': [self.stats.to_dict()]
        }

    def to_dict(self):
        return {
            'id': self.id,
            'run_id': self.run_id,
            'project_id': self.project_id,
            'testcase_id': self.testcase_id,
            'stats': [self.stats.to_dict()],
            'messages': [a.to_dict() for a in self.messages],
            'failmessages': [a.to_dict() for a in self.failmessages]
        }
```

Figure 4.8: Flask-SQLAlchemy table definition and data structure definition

4.2.2 PostgreSQL with Docker

PostgreSQL is an ACID [18] compliant [19][20], robust and reliable open-source relational database. ACID has four major properties atomicity, consistency, isolation and durability and a database being ACID compliant means, that it supports these four main properties and ensures that they are upheld for every single database transaction. According to an article [18] written by Theo Haerder and Andreas Reuter, titled "Principles

of transaction-oriented database recovery", the ACID test can be a good indicator of a database system's quality. These ACID values in more details:

- **Atomicity** [18]: It ensures that each transaction is treated as a single "unit" and is indivisible, meaning that either every operation connected to the transaction occurs or none of them. This all-or-nothing approach prevents situation where updates to the database would only be partially executed.
- **Consistency** [18]: It ensures that every constraints and rules are upheld for every transaction. Every committed transaction is legal.
- **Isolation** [18]: Isolation level determines the visibility of concurrent transactions to other users. Different levels change how each concurrent transaction see the effects of other concurrent transactions. For the ACID compliant databases, isolation needs to be at the highest level, meaning that each transaction has to be executed as if they were executed serially.
- **Durability** [18]: It ensures that each completed and committed transaction will remain that way even in the event of a system failure.

Postgres is also scalable both in terms of quantity of data and amount of concurrent users it can manages [20]. It is a popular choice for big companies like Netflix, Uber and Instagram according to multiple publications [21][22]. It supports SQL (relational) and JSON (non-relational) querying [20]. There is also a community maintained Docker Image for PostgreSQL that I am using for this project.

Docker:

Docker is a tool for creating and running software packages that include everything you need for your application to work [23]. These software packages are called Docker Images. For example, the PostgreSQL image I use for this project is a template which contains all the necessary components, libraries, configurations and dependencies to create a fully functioning Postgres database instance. These images can be run in Docker Containers. A container allows these images to run as microservices on any system without having to tailor it for any specific one. A container is a runtime environment that is equipped with all the necessary software, libraries and dependencies required by the image [23]. These properties make it portable and flexible as they can run on many different kind of machines or virtual machines.

4.2.3 Conclusion

Earlier when starting this thesis, I choose the SQLAlchemy option with the embedded database for this project, but as the scale grew, I have realized that the selected approach

won't be able to handle the load. To accommodate the needs of the day-to-day operation of the using this program I needed something more robust that can handle real world data. During testing the data showed that after the first few million lines, the embedded sqlite3 database slowed down significantly. At first it mainly showed when trying to query data from the database where it wasn't as essential for the response time to be quick, but as it reached the 10-12 seconds, it was evident that this will quickly become a problem. After testing, I decided to switch to PostgreSQL running in a Docker container. Switching to using a dedicated server running the database gives it more stability as it is not tied to the web application and more flexibility as it can be given more resources or move to better machines. Furthermore, as Docker containers can run in different environments without having to recreate it tailored for the specific system, the portability it provides is very valuable. Postgres has proved itself for many years and I have found a lot of documentation and tutorial for it which makes it easier to develop and maintain.

4.3 Object store

4.3.1 MinIO

MinIO is an open-source object storage solution that was written in the GO language and is known for being high performance, AWS S3 API compatible and easy to setup. It can work with unstructured data like photos, videos or documents. It supports a number of deployment options like standalone versions on Windows, Linux or MacOS, or containerized deployments through Kubernetes or Docker. To use this Docker image, we can execute the "docker pull minio/minio:latest" command in either Docker Desktop, Docker CLI or after installing the Docker extension, Visual Studio Code. Now, the image has been install and can be referenced in future builds with the "minio/minio" name. The MinIO setup is explored further at the implementation section of this thesis.

Object storage:

Object storage is a kind of data storage solution specialized in storing unstructured data as distinct units coupled with metadata and a unique identifier (UID) [24]. Data can be accessed through the use of APIs, HTTP or HTTPS requests.

5 DESIGN AND IMPLEMENTATION

This project can be divided up to 3 main parts in terms of what they are trying to achieve. Firstly we have the Fuzzer application which is supplying that data by communicating with an ECU. The second element is the Flask web application that, among other functionalities, is mainly responsible for taking the incoming data, putting it into the desired data structure and storing it in a database and later displaying the gathered data in a structure searchable view. And the last part of this project is an open source object storage server called MinIO, that is used to store the produced documents.

5.1 Design

When designing this project 3 major parts were outlined. First, getting data by collecting the communication from the Fuzzer and ECU in a structured manner. Storing this structured data in an easily accessible and retrievable manner, where custom reports can be created based on the curated data. And lastly, the persistent long term storage solution that contains the generated documents and reports in multiple formats. The figure 5.1 contains a flowchart describing the original process for this project. In the early phases of development all the messages were store and retrieved from the SQLite3 database embedded in the Flask web application.

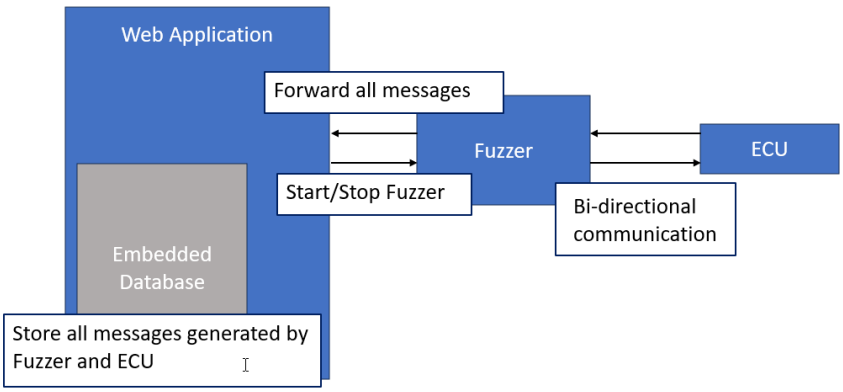

Figure 5.1: Original process flow for this project

After testing and receiving feedback from the team, I have decided to upgrade this embedded database into a dedicated one and also add a dedicated storage for the generated reports. A picture of the new infrastructure can be seen on Figure 5.2. The different components communicate and transfer data with each other using API calls. The user can choose and change Fuzzer configuration, then start it from the web application. The Fuzzer then starts the ECU testing which generates the data we need to store. The messages generated by the Fuzzer are sent to the web application in a JSON object. This

JSON object is processed and transformed into an SQL statement, then sent to the Postgres database. Once the ECU testing is done or while it is ongoing the user can access the collected messages from the database through the web application. Once the current run has finished the Fuzzer has built in Loggers that create logs from the test, these logs get uploaded into the object storage into their respective containers based on log types. Custom reports can also be generated through the web application that contain certain selected parts of the data generated. These report also get stored and can be retrieved from the object storage.

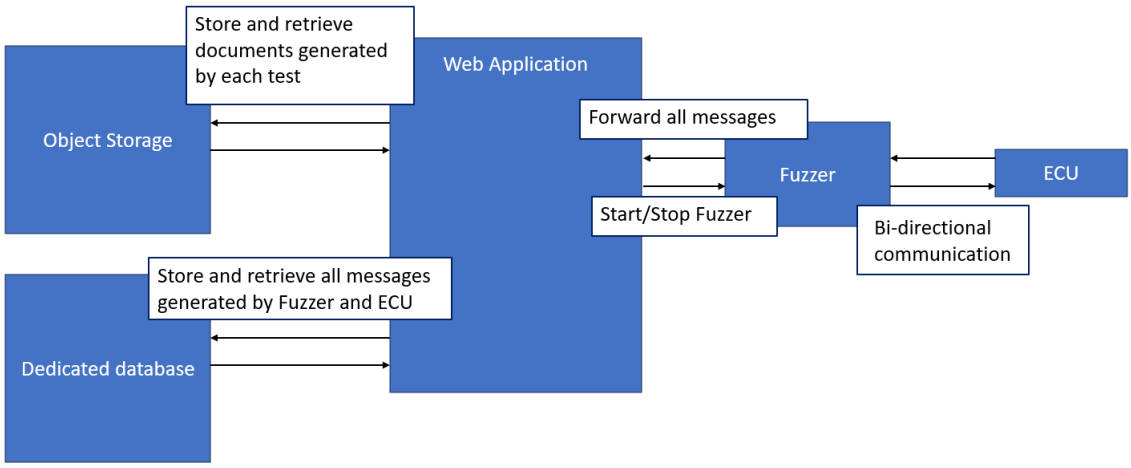

Figure 5.2: Updated flowchart showcasing the new infrastructure

The ECU and the Fuzzer are physically connected with the cabling and the web application also has to be on the same machine as the Fuzzer, but MinIo object storage and the PostgreSQL database are running in a Docker container. In the future as a possible expansion, the storage solution could be placed in the cloud allowing for more Fuzzer sending data at the same time.

5.2 Building Blocks

5.2.1 Fuzzer

The Fuzzer mention in this project is a proprietary software developed internally within Robert Bosch Kft. My thesis, as previously stated, revolves around developing this software not necessarily in the penetration testing direction. This python script is used to test a lot of different type of automotive ECUs in ways the client is requesting. Currently, there is no easy-to-use interface to change the configuration of the project or the selected protocol, to control and monitor the state of each test cases and to see all the collected data in a searchable, filterable and structured way.

My first task was to find a way to collect and store the messages going to and coming from the ECU. This solution to this problem was creating API calls after each message or once a bulk of messages has been collected, an example can be seen on Figure 5.3.

As the order and proof of delivery for each and every message was very important, the code was designed in a way, where it is waiting for an HTTP response status code that is either a 200 or a 201 after each message. In case there was an error, the message will be resent. The HTTP response status code is being sent by the Web Application based on the custom APIs created to handle the incoming data. The response codes are only being sent after there has been a successful database entry creation for the message, while this process is happening everything has to wait.

```
response = requests.post('http://127.0.0.1:8081/addfuzzitem', json=jsondata, headers={'Content-Type': 'application/json'})
```

Figure 5.3: Fuzzer message upload API call

There are unified patterns for how to handle and log each message, these patterns are called Loggers. It is an abstract Python class declared in the code designed to make sure every new data handler created is compatible with the already existing code and implements all of the required functions. These required functions that log data into long term storage options include the logging a single item, logging a bulk of items, logging the status and checkpoint update. For this project a new Logger descendant class was created, called DBLogger. The previously mentioned functions were overwritten to accommodate the integration of API calls and the creation of data structures needed by said API.

The next task was to create an easily modifiable configuration file. For this, after some research I have found a Python library called "xml.etree.ElementTree — The ElementTree XML API" [25]. It is a simple API implementation for parsing and creating XML files. Using this library allowed my to convert the configuration files into an XML format, which will be useful at the Web Application section because of better compatibility and ease of use.

5.2.2 Docker Desktop

In my implementation, I use Docker Desktop which is the Desktop application version of Docker. Through a Docker container I'm running multiple images in a separated independent environment and can expose ports so these images can communicate with the outside world. The configuration used for my implementation can be seen on Figure 5.4. This file contains all the startup instructions and run configurations used to create the container. A common factor in all of the services is a parameter called env_file. Using this command you can specify a file used to store all the environmental variables. In my implementa-

tion, this file contains the credentials for the MinIO admin user, credentials for the admin Postgres user and the name of the database I use, and finally credentials for the pgAdmin interface.

docker-compose.yml

```yaml
services:
  db:
    image: postgres
    restart: unless-stopped
    container_name: pg-db-container
    env_file:
      - .env
    ports:
      - '5432:5432'
    volumes:
      - db-data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/create_tables.sql

  pgadmin:
    container_name: container-pgadmin
    image: dpage/pgadmin4
    depends_on:
      - db
    ports:
      - "5050:80"
    env_file:
      - .env
    restart: unless-stopped

  minio:
    image: minio/minio
    command: server --console-address ":9001"
    ports:
      - 9000:9000
      - 9001:9001
    env_file:
      - .env
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
      interval: 30s
      timeout: 20s
      retries: 3
    hostname: minio
    volumes:
      - minio-data:/mnt/data

volumes:
  db-data:
    driver: local
  minio-data:
    driver: local
```

Figure 5.4: Docker container configuration file

Postgres Database

Version 17 of the PostgreSQL is run by Docker Desktop via the Postgres Docker image. Another image called pgAdmin 4 is also added to the container to have better controls over the database. Through the pgAdmin interface, I have created the database structure seen on Figure 5.5.

- **projecttable:** This data table holds all the projects and their file path. Every existing or new project gets added to this and gets queried from by the web application.
- **testcasetable:** This datatable holds all the testcases along with their file path, file name and concatenated string. The concatenated string contains the project folder name, the subfoldername and the name of the testcase pythong file connected together with dots. An example for this is "functional_tests.uds_analysis.example".
- **testruntable:** This data table holds all the testruns. For each run, as many of these entries are created as many test cases are selected. The id is a unique key pointing to the individual runs and the run_id is the same among all test cases. It also has two foreign key constrains pointing to the project it belongs to and test case it runs.
- **stattable:** This data table holds the statistics belonging to each individual test case in a run. There will be as many entries of this as many test case is selected for the given test run. The included statistics include the number of sent and received messages, the time it took to run the test case and the state of the run. When an entry is created, the state, which is a boolian field, starts out as false and later gets updated once the test case is finished. The other metrics are null until updated at the end of the test case.
- **fuzz:** This data table holds the messages generated by the Fuzzer and the ECU. The testrun_id in this table is a foreign key pointing to the id of testrun table. This means that each message is assigned to test case inside of the test run. The event field contains if the message is sent, received or a checkpoint, the description is the content of the checkpoint or the payload of the packet, the frame_timestamp is the elapsed time since start and the timestamp_base is the starting timestamp and finally the result is an Enum that can only be one of four values. The result Enum can be Passed, Failed, Non or Warning.
- **fuzzfail:** This data table is almost the same as fuzz with the exception of the failure_id. The failure_id is a number generated by the Fuzzer, used for grouping the messages into failure blocks. These are connected messages that get analyzed together later on. A failure block contains messages connected to a pattern, meaning a special order of messages that needs to be checked and needs to give back a given response specified by the manufacturer. If the response is not as it should be, that means that the testing found an error which should be investigated and reported back to the client.

MinIO Object Storage

I am using Docker Desktop to run a MinIO Docker image. The setup the MinIO Docker image is used in can be seen on the Docker container configuration in Figure 5.4.

- **Service declaration:** Name the service as "minio".
- **Specify Image:** Here we can use a mirror link, through which Docker will down-

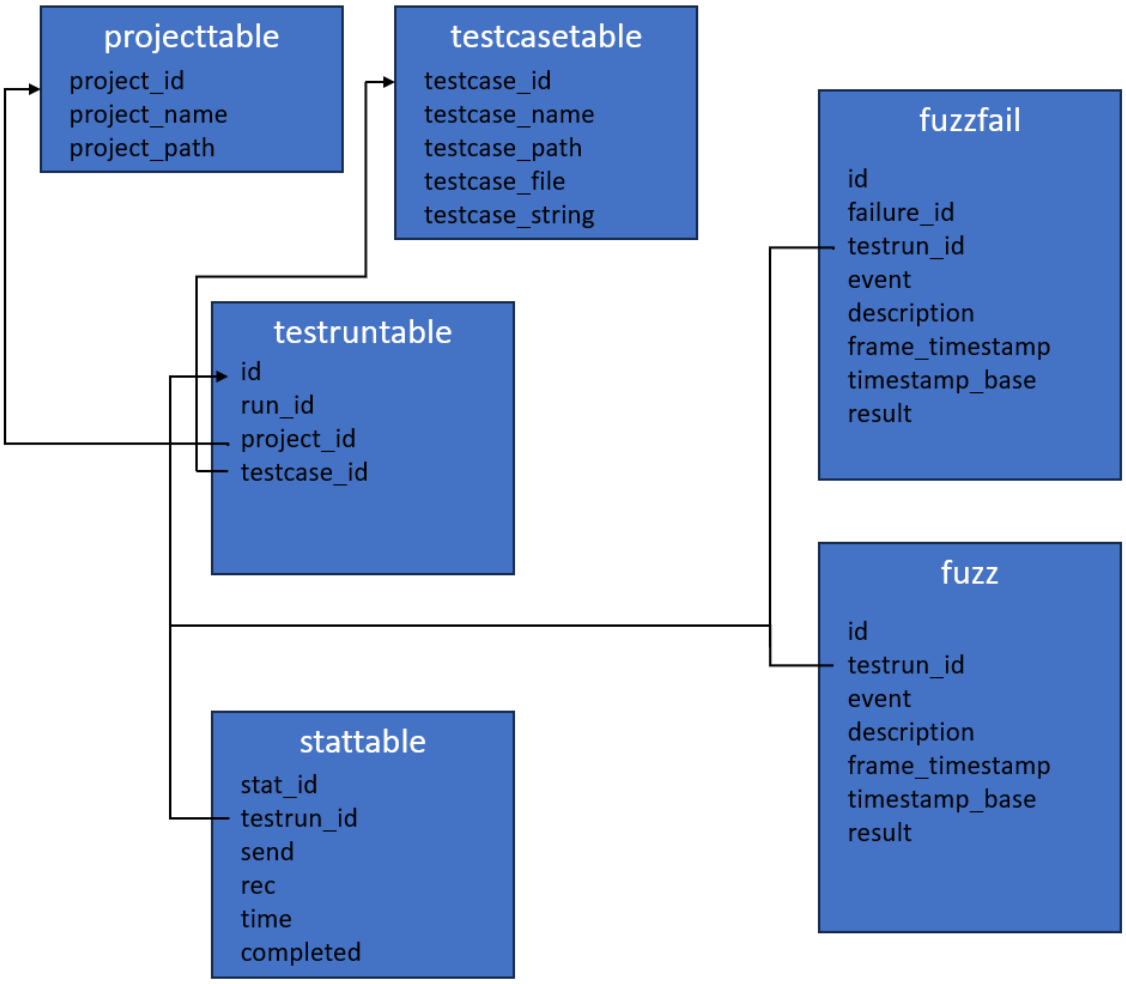

Figure 5.5: Database structure used in the Postgres DB

load the image or, if already downloaded, we can use the name of the image.
- **Startup command:** A command that will be executed when the container starts up. This line ensures that the MinIO server console inside of the container can be reached through the port 9001.
- **Ports:** Here we can specify which port it will use and map it to the container's outside ports
- **Env_file:** We can give it an environment file it can use to establish environmental variables like credentials.
- **Healthcheck:** We can setup a healthcheck that periodically checks if the service is operational.
- **Hostname:** We can set the hostname of the service.
- **Volumes:** Sets which local directory we mount the directory inside the container.

First, we declare the service called minio. Then we specify the image to used, in this case it's the downloaded MinIo image. If we give it a name instead of a URL to pull the image from, it will look for it locally. Next, we give it a command that will be executed when the container starts up. This line ensures that the MinIO server console inside of the container can be reached through the port 9001.

5.2.3 Flask Web Application

The main part of the project is a Flask web application written in Python. This application connects to the Fuzzer and the Docker container through API calls. This application has multiple purposes. These include the following:

Fuzzer control:

One of the main reason this project started was to provide better control over the Fuzzer. This finer control is about being able to start and stop the Fuzzer along with being able to change the starting parameters for both the Fuzzer, the selected test case specification and other configuration settings all in one central place, which in this case, is a Web UI.

The first component used for control is the "start_process()" function. It is registered as an endpoint for the web application under "/runprocess" and is limited only to POST requests. This limitation is enforced by the "methods=['POST']" parameter provided in the endpoint declaration. To start the Fuzzer we need call this endpoint from the client side of the web application. I have created a button on the page called Socket to utilize this function. The Socket page can be seen on Figure 5.6. Through the use of JavaScript I have attached a function that calls the "/runprocess" endpoint, this can be seen on Figure 5.7.

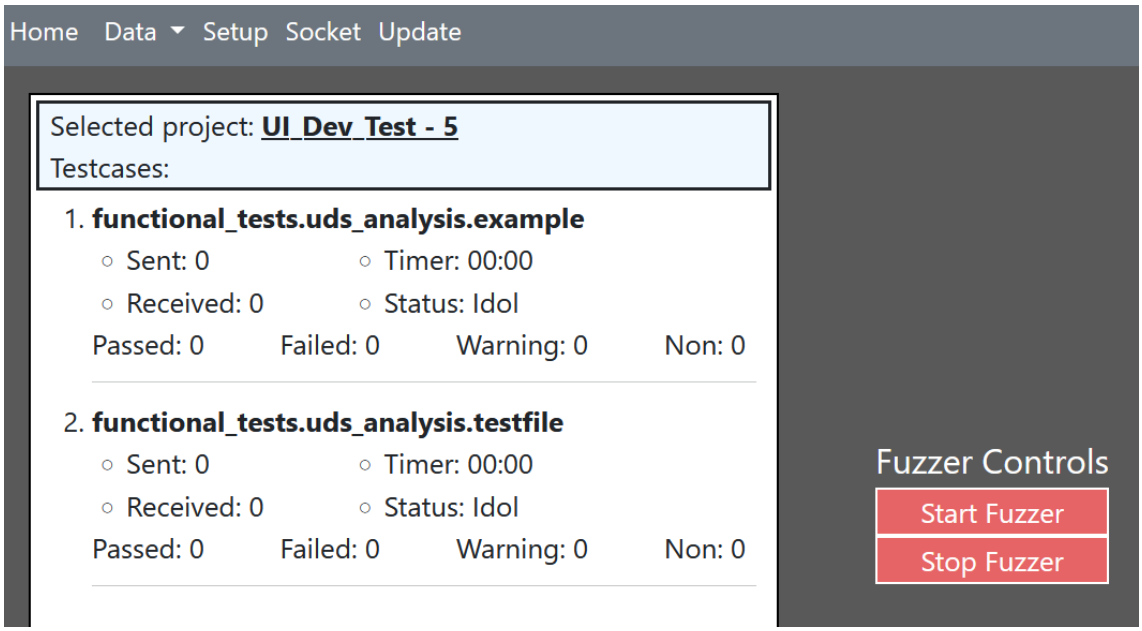

Figure 5.6: Landing page called Socket, used for Fuzzer control

```
<script>
    const starterbutton = document.getElementById('start-button')
    const stopperbutton = document.getElementById('stop-button')

    stopperbutton.addEventListener('click', () => {
        socket.emit('stopprocess')
    })

    starterbutton.addEventListener('click', () => {
        fetch('/runprocess', {method: 'POST'})
        starterbutton.disabled = true;
    });

    socket.on('buttonenable', () => {
        starterbutton.disabled = false
    })
</script>
```

Figure 5.7: JavaScript function for Fuzzer control

To start the Fuzzer an activation string has to be provided. As can be seen on the first line of Figure 5.8, the activation string is made up of 2 components, the name of the interpreter that is going to run the application and second is a path to the target application. This file path is made up of 2 parts, an environmental variable that is dynamically set each time the application is launched, and the static string that points to the entry point. This application is going to be packaged together with the Fuzzer, thus the environmental variable will always be able to point to the correct place as it is a relative path based on this web application's location.

```
cmd = ['python', os.environ.get('basedir') + '\\ACFuzz\\main.py']


try:
    if process is None or process.poll() is not None:
        process = await aio.create_subprocess_exec(*cmd, stdout=aio.subprocess.PIPE, stdin=aio.subprocess.PIPE, creationflags=subprocess.CREATE_NEW_PROCESS_GROUP)

        while True:
            line = await process.stdout.readline()
            if not line:
                break

except Exception as e:
    print(e)
```

Figure 5.8: Code snippet: Fuzzer starter

The other part of the code snippet seen on Figure 5.8, uses a Python library called AsyncIO[26]. This library provides seemingly asynchronous process execution for I/O bound applications using clever task scheduling and taking advantage of execution down times, when the program is waiting for an I/O operation like a response from another application or API.

The command "asyncio.create_subprocess_exec()" is used to create and start a subprocess for the Fuzzer. The inputs shown on Figure 5.8 in order:

- *cmd: list of input strings, passed down to the subprocess –> Python to run the application found on the given path.
- asyncio.subprocess.PIPE: According to the documentation [27], this subprocess.PIPE is a constant, that can be passed down as a parameter to the stdin and stdout inputs to handle the input and output streams of the subprocess. This output handling can be seen 2 lines below the "create_subprocess_exec()" command on Figure 5.8. Using the "readline()" function of stdout, we can read exactly one line from the output stream.
- subprocess.CREATE_NEW_PROCESS_GROUP: This creationflags parameter according to the documentation [28], is used to create a new process group. This new process group is necessary for having better control over the subprocess as this parameter is what enables the use of os.kill() on the subprocess. It also helps in sending signals to the subprocess using the signal Python library. As can be seen on the Figure 5.9, where we are sending a termination signal to the subprocess through the "send_signal()" function of the process.

The second component in the Fuzzer control is the "stopprocess()" function. A reference can be seen on Figure 5.9. It is also attached to a button on the Socket page featured on Figure 5.6. Here I have used a slightly different approach utilizing the Flask-SocketIO Python library, which is a Flask compatible integration of the Socket.IO Library. According to its documentation, Socket.IO provides low-latency, bidirectional, event-based communication between client and servers. I use this library to transport information between frontend and backend through WebSockets. This is useful for updating the statistics on the Socket page and for displaying the communication between the Fuzzer and the ECU.

```python
@io.on('stopprocess')
def stopprocess():
    global process
    try:
        if process is not None:
            process.send_signal(signal.SIGTERM)
        else:
            print('asdasdasd')
    except Exception as e:
        print(e)

    io.emit('stopped')
```

Figure 5.9: Code snippet: Fuzzer stopper

An example showcasing the Socket page after a run can be seen on Figure 5.10.

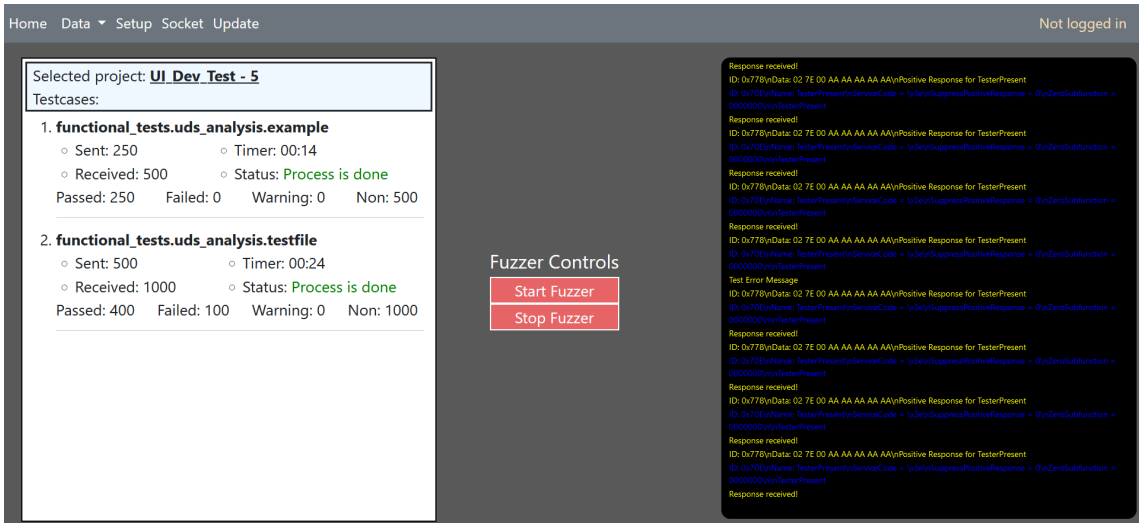

Figure 5.10: Statistics and message display example

The third point mentioned for Fuzzer control is the setup configuration change from the web application. This can be done through a different page, called Setup. On this page there is an auto-populating list of projects that can be selected for the run. A picture can be seen on Figure 5.11. Some project names have been blurred to protect sensitive information.

The list is auto-populated from the "projecttable" data table in the Postgres database seen on Figure 5.5. The "projeccttable" table is being queried each time the page is access to compare the stored entries with the locally stored files. This comparison is done by getting all the sub-folders in the Fuzzer's project library using the glob Python library [29] and checking if there are any that is not yet present in the database. The logic for

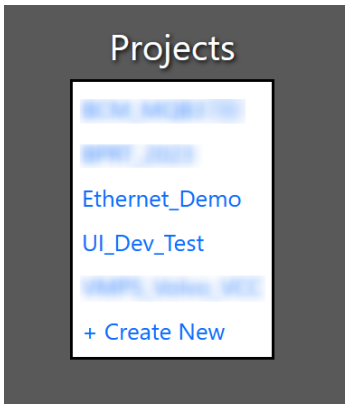

Figure 5.11: Setup page with project list

```
files = glob.glob(os.environ.get('basedir') + "\\ACFuzz\\projects\\*\\*.yml")
das = [{
    'Project': os.path.basename(os.path.dirname(os.path.abspath(x))),
    'Path': x
    } for x in files]
data = [(x['Project'], x['Path']) for x in das]
```

Figure 5.12: Code snippet: getting all projects

finding and adding the missing items to the database can be seen on Figure 5.13

The collection of projects is done by using the "glob()" function that takes a target file path containing a specific search pattern and returns a list of filenames that fit the pattern provided as input. Figure 5.12 shows the pattern I used to get all the projects file paths that contain a yaml file.

The items displayed on the Setup page are auto-populated based on the data rendered along with the template. On Figure 5.13, the structured data in the variable projectlist is being passed along with the template to the user. A copy of this variable, under the name of list, becomes usable on the client side and the data inside can used in JavaScript and HTML. The white text inside of the HTML code is the previously mentioned Jinja2 templating engine. On Figure 5.14, I use Jinja2 to go through all elements of the list variable using a for loop. For each element include another template called "_file.html" 5.15. This new template after the inclusion, acts like it was inserted into the other HTML file and can use the local variables.

After loading all the projects, the user can select or create a new one. if we select one of the available choices, it will redirect us to the next page that has the URL of endpoint declaration of "@app.route('/setup/<filename>', methods=['POST', 'GET'])". This page is also auto-populated based on the configuration file. Here we have a few options:

– **Add testcases:** Above the buttons, a list can be seen that is also dynamically filled

```
    else:
        # projecttable is not empty
        projectpaths = [a[2] for a in projectrows]

        appenddata = []
        for a in data:
            if a[1] not in projectpaths:
                appenddata.append(a[0], a[1])

        if appenddata:
            sql = "INSERT INTO projecttable (project_name, project_path) VALUES (%s, %s)"

            try:
                cur.executemany(sql, appenddata)
                postgreconn.commit()
            except (Exception, psycopg2.DatabaseError) as error:
                print(error)
                postgreconn.rollback()

    sql = "SELECT * FROM projecttable;"
    cur.execute(sql)
    projectrows = cur.fetchall()


    projectlist = [{
        'Project': p[1],
        'Path': p[2],
        'Id': p[0]
    } for p in projectrows]


return render_template('setup.html', title='Setup Page', list=projectlist)
```

Figure 5.13: Code snippet: project comparison, uploading missing and rendering template with updated information

```
{% extends "base.html" %}

{% block content %}


<div id="setupfilesdiv">
    <h1 style="color: white; text-shadow: 2px 2px 5px black;">Projects</h1>
    <table id="setupitems">
        {% for f in list %}
            {% include '_file.html' %}
        {% endfor %}
        <tr>
            <td><a href="{{ url_for('filesetup', filename='new') }}" style="text-decoration: none;">+ Create New</a></td>
        </tr>
    </table>
</div>


{% endblock %}
```

Figure 5.14: Code snippet: HTML code with Jinja2 statements used in Setup page

```
<tr>
    <script>
        console.log('{{ f.Id }}')
    </script>
    <td><a href="{{ url_for('filesetup', filename=f.Project) }}" style="text-decoration: none;">{{ f.Project }}</a></td>
</tr>
```

Figure 5.15: Code snippet: HTML code for _file.html

```
const updateButton = document.getElementById("update");
const form = document.getElementById("configForm");

updateButton.addEventListener("click", async () => {
    event.preventDefault();

    const inpform = document.getElementById("configForm");
    const formdata = [];

    for (const element of inpform.elements){

        if (element.tagName == "INPUT"){

            if (element.getAttribute("type") == "checkbox"){
                const obj = {
                    "name": element.getAttribute("name"),
                    "value": element.checked,
                    "path": element.id
                }
                formdata.push(obj)
            } else {
                const obj = {
                    "name": element.getAttribute("name"),
                    "value": element.value,
                    "path": element.id
                }
                formdata.push(obj)
            }
        }
    }

    socket.emit('xmlupdate', formdata, "{{ path }}")
    await sleep(300)
    form.innerHTML = "";
    await fetchParameters();
});
```

```
const xmlpath = "{{ path }}"
const fetchButton = document.getElementById("fetchButton");

fetchButton.addEventListener("click", async () => {
    form.innerHTML = "";
    await fetchParameters();
});

async function fetchParameters() {
    const response = await fetch(xmlpath);

    if (!response.ok) {
        throw new Error("Error fetching XML data!");
    }

    const xmlString = await response.text();
    parseXML(xmlString)
    //console.log(AllElements)

    const form = document.getElementById("configForm");
    const tags = []
    const asd = []
    let CurrentParent;
    let CurrentNode;
    let idstring = "";
    let IsLast = false;
    let h = 3
    for (const [name, e] of Object.entries(AllElements)) {
```

Figure 5.16: Code snippet: (left) Update configuration file (right) Fetch configuration file

with all the available testcase options. The testcases are queried from the testcasetable. These list entries have a checkbox next to them and once the "Add TestCases" button is pressed, the selected testcases get written to the configuration file using the "xml.etree.ElementTree" [25] Python library. An example can be seen on Figure 5.21

– **Update configuration file:** If we want to change any configuration in file, we can change it on this page and push the "Update Configuration" button. This will read the data from the input form and gather all the data into an array. Once everything has been collected and structured, this array gets sent to the application using SocketIO. A code snippet of this logic can be seen on the left side of Figure 5.16. The data getting sent to the application from this input form will be processed by a function that is listening on the 'xmlupdate' event. A code snipped of the Function connected to this event handler can be seen on Figure 5.17. As the code is currently not in live production, for testing purposes the configuration files are not the actual files used by the Fuzzer. The folders and files created, loaded or updated are inside of the web application and are dummies made for demonstration purposes.
– **Load configuration file:** To load the current state of the configuration file. we can press the "Get Configuration" button. Pressing it will clear the content of the input form completely and fetch the XML file again for the designated project folder. A

code snippet of this logic can be seen on the right side of Figure 5.16.

– **Start:** The "Start" button is a redirect to the Socket page, basically a shortcut.

Another important configuration is under the "REPORTING" header. Here are the different options for logging the data. The current application is of them and here we can turn of the logging into the database. The DBLogger option is responsible for telling the Fuzzer to use the logger that send the data to the application and the from there the database. The other options are already existing solutions. The ConsoleLogger is for debugging, it writes out the messages into the console. If this option is turned off, it will also turn off the console logging in the web application.

```python
@io.on('xmlupdate')
def updateconfig(data, path): #case

    tree = ET.parse(os.environ.get('basedir') + f"\\ACFuzz-UI-Development-Server\\app{path}")
    root = tree.getroot()

    for a in data:
        print(a['path'])
        target = root.find(f".{a['path']}/parameter[@name='{a['name']}']")
        if target is not None:
            if a['value'] in {'True', 'False','true','false'}:
                target.text = str(a['value']).lower()
            else:
                if a['name'] in 'TestCase':
                    print(a['name'])
                else:
                    target.text = str(a['value'])

    tree.write(os.environ.get('basedir') + f"\\ACFuzz-UI-Development-Server\\app{path}")
    return {'message': "good"}, 200
```

Figure 5.17: Code snippet: Code for processing the changes made to the configuration

Here on this page we will have a different HTML template loaded in if we chose to create a new project. As this endpoint accepts both POST and GET requests, there are 2 completely different pages under the same address. An example for the project creation page can be seen on Figure 5.18. Here we can give it a name and select a configuration file we want to start with. This option was added instead of starting with an empty template, because a lot of the project configuration files are similar in terms of what kind of variables they contain. This means that, it saves you the trouble of going through all the different templates to copy from if you forgot something or you have to create the same project just with different values and name. There is still an option for an empty template if needed.

This page is generated from a class acting as a form. The class is called "NewProjectForm2" and takes a "FlaskForm" clas as an input and can be seen on the right side of Figure 5.19. The "FlaskForm" class makes this form Flask compatible and can be passed as input when rendering the template. Using this approach means that the dynamically pre-populated values of the drop-down menu will not have to be added with JavaScript or other ways, it will just work out of the box. The code snippet of how the fields of the form are being used can be seen on the left of Figure 5.19.

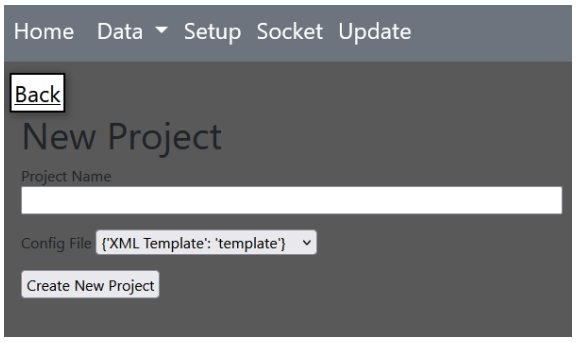

Figure 5.18: New project creation page

```
<div style="padding-left: 20px;">
    <h1>New Project</h1>
    <form action="" method="post" novalidate>
        {{ form.hidden_tag() }}
        <p>
            {{ form.name.label }}<br>
            {{ form.name(size=64) }}
        </p>
        <p>
            {{ form.GenerateFrom.label}}
            {{ form.GenerateFrom()}}
        </p>
        <p>
            {{ form.submit() }}
        </p>
    </form>
</div>
```

```
class NewProjectForm2(FlaskForm):
    name = StringField('Project Name')
    pjs = glob.glob(os.environ.get('basedir') + "/ACFuzz/projects/*/*.yml")

    das = [{
        'Project': x.split('\\')[1]
        } for x in pjs]

    das.append({"XML Template": "template"})

    GenerateFrom = SelectField("Config File", choices=das)
    submit = SubmitField('Create New Project')
```

Figure 5.19: Code snippet: (left) HTML and Jinja2 code for form generation (right) Project creation form

Pressing the "Create New Project" button will trigger the submit function of the form and send the information to the web application. At the backend, the form gets validated and the name of the project as well as the template selected is used to create a new project folder with the configuration file in it. This process can be seen on Figure 5.20.

```
try:
    if form.validate_on_submit():
        proj = form.name.data
        temp = form.GenerateFrom.data

        if proj:
            if not os.path.exists(f"/app/static/xml/{proj}"):
                os.makedirs(os.getcwd() + f"/app/static/xml/{proj}")
                shutil.copy2(os.getcwd() + f"/app/static/xml/{temp.split("'")[3]}/project_config.xml",
                             os.getcwd() + f"/app/static/xml/{proj}/project_config.xml")

        else:
            return render_template('newproject.html', form=form)

        return render_template('setupPage.html', path=f"/static/xml/{proj}/project_config.xml",
                               cases=cases, filename=filename)
    else:
        raise Exception()
```

Figure 5.20: Code snippet: Data from the form is used to create project folder

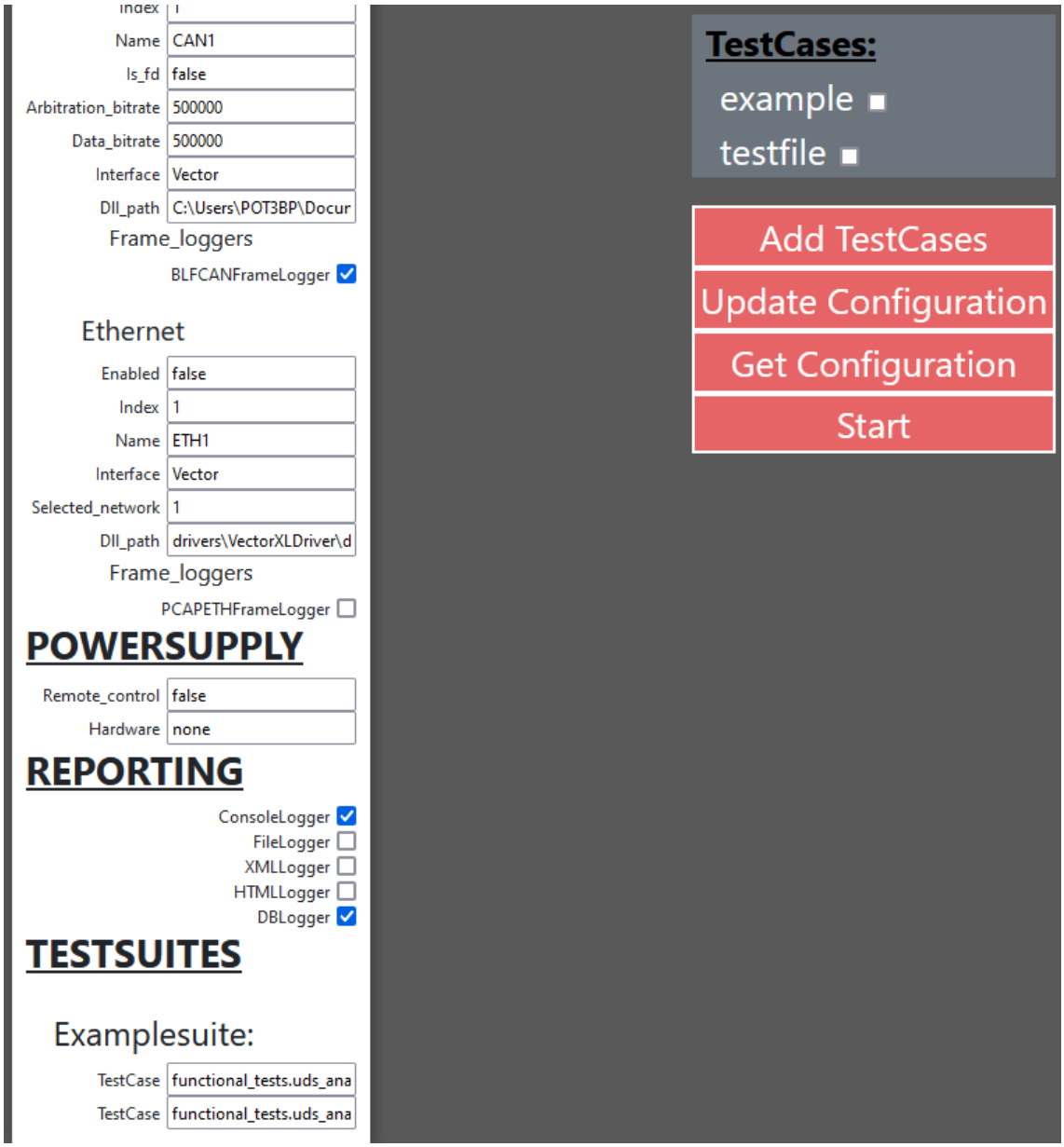

Figure 5.21: Setup page with specific project's configuration

5.3 How does it work

The process of using the applications is the following:

0. When the application is opened we arrive at the Home page. The Home page doesn't contain anything functional.
1. The first step is to go to the Setup page and select or create a new Project.
2. Once a project has been selected or created, we get redirected to the Project configuration page. Here we can add or remove test cases or change other configurations.

Once done with the changes, save it and press start.

3. The start button will bring you to the Socket page, there pressing the "Start Fuzzer" button will start it. On the right, all the massages are displayed. On the left, some statistics are shown.
4. once the runs is finished, the data can be seen on either the "All Messages" or the "Library" page, which are under the "Data" dropdown menu.

6 SUMMARY

6.1 Testing

As part of this thesis, I needed to test the capabilities and the stability of the setup I put together. There are two main tests that are required for this application to be usable in a professional setting. The first test is a stress test and passing this is essential. This test is about seeing how this infrastructure performs under full load and running for an extended amount of time. The second test is a performance test. Prior to creating this application, the Fuzzer could run as fast as it could without being limited by outside factors, and the only bottleneck it faced was the machine running the application being slow or the ECU it was testing responding slowly. Now however, I have introduced multiple variables into the mix making it more complex and created multiple potential bottlenecks into the system. By testing we can figure out if these new variables cause any performance or stability issues.

The testing environment is in a professional tech lab inside of the Bosch office. The application being tested is run on a high-performance computer alongside the Fuzzer and the Docker container. The high-performance computer, referred to as tower later on, is For this early stage testing, we are mainly focusing on how the building blocks work together without interference or other problems caused by other not connected components.

6.1.1 Stress testing

For this test, I have decided to see how long would it take to run a dummy test case that sends 450000 messages to the ECU. This dummy case is very simplistic with the basic test being to see if the ECU would respond after each message sent. This would be saved as a failed or passed checkpoint which would also be recorded into the database. A sample of the data collected during this test run can be seen on Figure 6.1. The 450000 messages sent to the ECU would produce another 450000 messages as answers and another 450000 as checkpoints. This amounts to 1350000 messages into the fuzz data table and another 1350000 messages into the fuzzfail data table which, in this dummy test case is sending the same messages and only adding a failure_id. Later on, in live testing this will be different. The end if this test the Fuzzer sent 2700000 messages into two data tables without crashing or slowing down. During the runs, we can change to other pages without any problems, even pages where it is sending a query to the data table we are uploading data to.

I ran a similar high data quantity tests 2 more times, with this data already in the database. There were no crashes or problems. During testing the CPU usage deviated by

Query Query History

```
1 select * from fuzzfail where testrun_id = 63 order by id desc
```

Data Output Messages Notifications

Showing rows: 1 to 1000

	id [PK] bigint	event text	description text	frame_timestamp double precision	timestamp_base double precision	result result	testrun_id integer	failure_id integer
1	1365537	Chenckpoint	Response received!	0	0	Passed	63	43548
2	1365536	Recv CAN	ID: 0x778\nData: 02 7E 00 AA AA AA A...	25107.05174088478	1734216850.1852014	Non	63	43548
3	1365535	Send CAN	ID: 0x70E\nName: TesterPresent\nSer...	25107.050625562668	1734216850.1852014	Non	63	43548
4	1365534	Chenckpoint	Response received!	0	0	Passed	63	43548
5	1365533	Recv CAN	ID: 0x778\nData: 02 7E 00 AA AA AA A...	25107.02167034149	1734216850.1852014	Non	63	43548
6	1365532	Send CAN	ID: 0x70E\nName: TesterPresent\nSer...	25107.01650261879	1734216850.1852014	Non	63	43548
7	1365531	Chenckpoint	Response received!	0	0	Passed	63	43548
8	1365530	Recv CAN	ID: 0x778\nData: 02 7E 00 AA AA AA A...	25106.98328590393	1734216850.1852014	Non	63	43548
9	1365529	Send CAN	ID: 0x70E\nName: TesterPresent\nSer...	25106.97069644928	1734216850.1852014	Non	63	43548
10	1365528	Chenckpoint	Response received!	0	0	Passed	63	43548
11	1365527	Recv CAN	ID: 0x778\nData: 02 7E 00 AA AA AA A...	25106.94349527359	1734216850.1852014	Non	63	43548
12	1365526	Send CAN	ID: 0x70E\nName: TesterPresent\nSer...	25106.936052799225	1734216850.1852014	Non	63	43548
13	1365525	Chenckpoint	Response received!	0	0	Passed	63	43547
14	1365524	Recv CAN	ID: 0x778\nData: 02 7E 00 AA AA AA A...	25106.848731279373	1734216850.1852014	Non	63	43547
15	1365523	Send CAN	ID: 0x70E\nName: TesterPresent\nSer...	25106.845531463623	1734216850.1852014	Non	63	43547

Figure 6.1: Stress testing query results in pgAdmin

approximately 2 to 3 percent and the memory usage went up by 2 percent. It is important to note, that the testing was performed on the tower, which was custom built for testing purposes and is well equipped for handling these kinds of operations.

One of the requirements for the thesis was to be able to look at the data while the fuzzer is running. There are 2 pages for this. Firstly, there is a page called "All messages" where all the messages from the fuzz data table are present in a paginated view. It can be seen on Figure 6.2. And the other option is a page called Library. It can be seen on Figure 6.3.

There is a statistics entry for each test case of each test run. The structure and content of the stattable data table is explained in the implementation part of this thesis. This data table collects and stores data about each run, including the time it took for each test case to finish. The data collected about the first stress testing run can be seen on Figure 6.4.

6.1.2 Performance testing

Prior to creating this build the Fuzzer was going as fast as it was possible given the other components participating in the process. When tested without enabling the DBLogger, the Fuzzer collected 13-15 messages per second. Compared to this a previous version without the AsyncIO library creating a subprocess to run the Fuzzer in, achieved 7-9 messages per second and with AsyncIO it achieved 14-15 messages per second, sometimes even higher.

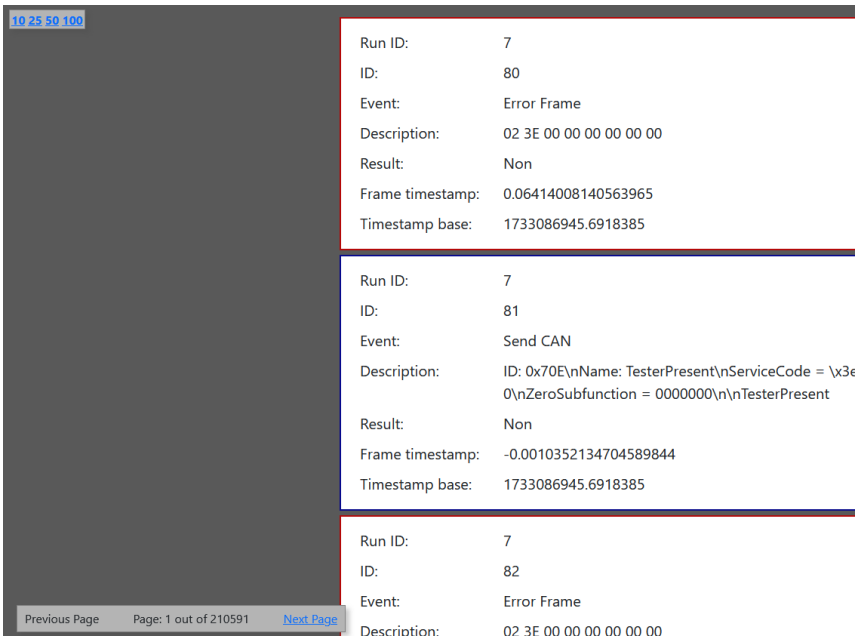

Figure 6.2: Paginated view of all messages

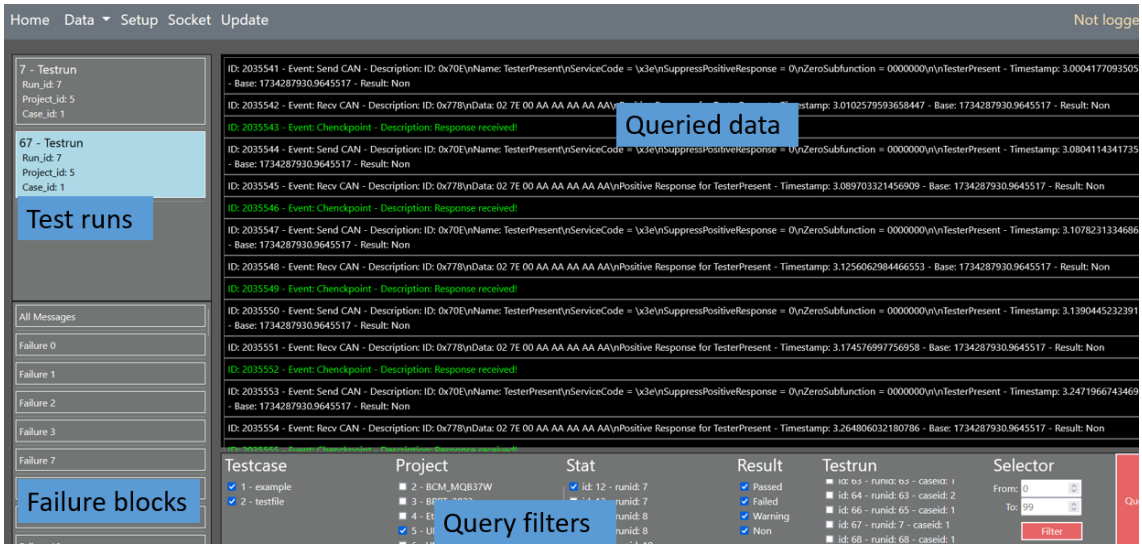

Figure 6.3: Library page

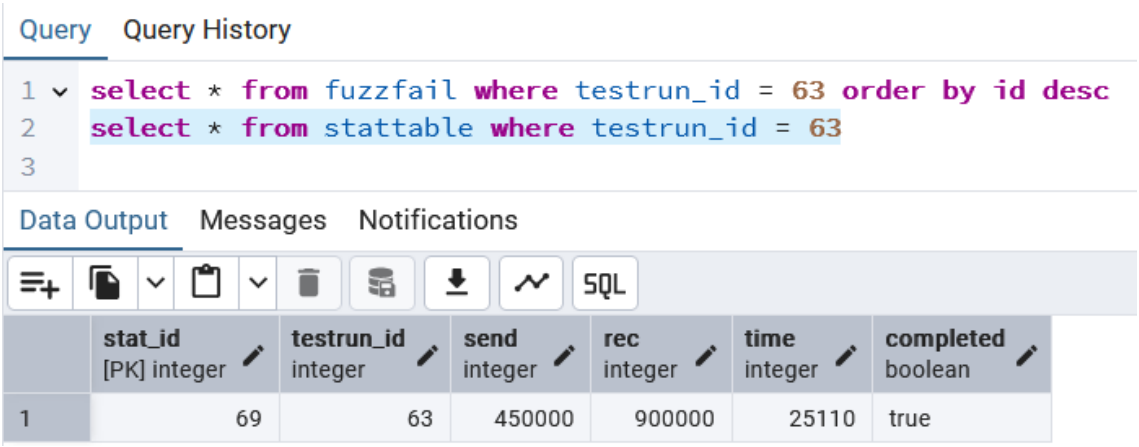

Figure 6.4: First stress testing run results in pgAdmin

6.2 Future plans

Future potential expansions for the project can come in a few different ways. Even though, what I made in it's current form is usable and is a huge improvement over the previous iteration of Fuzzer, I can't say it's close to perfect. The way it works currently, relies on a number of back and forward calls between the Fuzzer and the web application. I believe this could be simplified which would eliminate complexity and make it less error prone.

For the data review part, currently there is a limitation of only being able to select a single failure block at a time. Adding the ability to freely add or remove blocks from the view can make the review process faster and more convenient.

The storage solution could be hosted in the cloud, making it much more scalable and flexible.

As for the application, currently with the way it's written if more then one Fuzzer would be started at the same time, it would get confused. In the future, this might be something to improve on.

6.3 Conclusion

In this part of the thesis, I will summarize what has been created. After this section will be a reflection compared to the goals set for this thesis.

Control:

The application can start and stop the Fuzzer from the client side with the press of a button found on the Socket page that can be seen on Figure 5.10. It also has a screen to

show the messages and checkpoints being recorded into the database, as soon they arrive at the application in real time.

Configuration:

The configuration changing functionality is not completely implemented. The functionality itself works perfectly, but because of time constraints and other outside factors, the complete rebuilding of configuration files into a different structure and format and the adaptation of the Fuzzer to use the new format haven't been completed. The current version of the code reads and changes a partially recreated configuration file that was based on a real one. Using this partial file is not much different from how it would work, if I changed the target location where the application is looking for configuration files and remade the real configuration files inside of the fuzzer into the compatible format.

Data review and report creation:

The ability to query and display data dynamically based on user input is working properly. A picture describing the different options and buttons can be seen on Figure 6.3. The main concern from my team's part was to be able to see the different failure blocks separated and be able to go through them individually. This can be done by selecting the chosen failure block from the list. Also the checkpoints are dynamically color-coded based on their status. The user can also set how many entries or what interval of IDs they want to see displayed. The report creation function is not perfect, currently it only supports the txt file format. Currently there is only one way for report creation and that is to create a report containing all of the messages.

Storage Solutions:

I created a separate dedicated database to store all the data needed to run and generated by the Fuzzer. Using Docker, I set up a PostgreSQL database running in a Docker container. The web application can connect to this database and send and receive data without any problems. Database is stable and can support the high I/O tasks it needs to.

I also set up an Object storage system called MinIO to store all the documents generated by the runs. Web application can connect to MinIO and store files and create containers in it. This part of the thesis has not been explored too deeply because of time constraints and also because it plays a minor part in the big picture.

Performance and usability:

Performance of the Fuzzer has not been impacted with the infrastructure built around it. Using AsyncIO, the Fuzzer runs within the required speed.

Adding the UI has made operating the Fuzzer more straightforward. There is a designated page for data review, changing configuration and running test runs.

6.4 Konklúzió

A szakdolgozat ebben a részében összefoglalom, hogy mit értem el. A következő részben összehasonlítom a célokhoz az elkészített

Irányítás:

Az applikáció képes elindítani és megállítani a Fuzzer-t a kliens oldalról a Socket weboldalon gombnyomásra. Ez látható a képen is 5.10. Csináltam egy kijelzőt az indító gombok mellé, melyen látszanak valós időben az üzenetek és a Checkpoint-ok, ahogyan töltődnek fel az adatbázisba.

Konfiguráció: A konfiguráció változtató funkcionalitás nincsen teljesen implementálva. Maga a funkció tökéletes működik, de határidő problémák miatt, nem lettek átdolgozva a valós projectek konfigurációs fájljai és a Fuzzer-ben sem lett teljesen átalakítva, hogy a másik fájl típust tudja használni. Ha a Fuzzer át lenne alakítva, hogy működjen XML-el is, csak át kellene írni az elérési utat és tökéletesen működne a funkcionalitás. A jelenlegi kód képes beolvasni, dinamikusan megjeleníteni és után a visszaírni a megváltoztatott adatokat az XML fájlba.

Adat átnézés és kimutatás kreálás: A program képes megjeleníteni az eltárolt információt dinamikusan felhasználó által megadott keresési feltételek alapján. A felület látható ezen a képen 6.3. A fő pontja ennek az oldalnak, hogy lehessen szeparáltan látni a különböző bukási blokkokat. Ezek a blokkok bizonyos üzenet küldési mintát jelentenek, melyekre egy a gyártó által meghatározott választ kell kapni. Ezekben a blokkokban meg van szabva, hogy hány sikeres és sikertelen Checkpoint-nak kell szerepelnie vagy különben jelenteni kell a gyártónak. Jelenleg a kimutatás gyártás úgy máködik, hogy összegyűjti az összes üzenetet txt formátumba és feltölti a MinIO tárolóba.

Tárolók: Csináltam egy szeparált egyedülálló PostgreSQL adatbázist, melyben tárolom az összes adatot, amit a Fuzzer generál. Ez az adatbázis Docker technológiával van futtatva egy Docker konténeren belűl. A web applikáció akadálymentesen, teljes sebességgel tud üzeneteket küldeni és fogadni az adatbázistól és bírja sok üzenet gyors küldése által okozott terhelést.

Szintén készítettem egy MinIO nevezetű objektum tárolót, melyeben tárolom a kimutatásokat. Ehhez a tárolóhoz, is tud kapcsolódni és adatokat feltölteni a web appliká-

ció. Ez a része a szakdolgozatnak nem volt olyan mélyen kikutatva, mivel ez egy alacsonyabb prioritású komponense a projektnek. Teljesítmény és használhatóság: A teljesítménye a Fuzzer-nek nem változott a köré épített infrastruktúrától. Az AsyncIO könyvtár használatával elvárt kereteken belül működik. A grafikai kezelőfelülettel sokkal egyszerűbb lett a használata a Fuzzer-nek. Minden funkcionalitásnak van külön saját oldala.

7 REFERENCES

[1] *ISO/SAE 21434:2021 - Road vehicles, Cybersecurity engineering*. ISO - International Organization for Standardization, 2021.

[2] "Understanding the mirai botnet.," *In 26th USENIX security symposium (USENIX Security 17) (pp. 1093-1110).*, 2017.

[3] C. Osborne, *Krebs on security booted off akamai network after ddos attack proves pricey*, Last accessed 11th of May 2024, 2016.

[4] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, 2022, ISSN: 0360-0300. DOI: 10.1145/3512345.

[5] Y. Jitsunari and Y. Arahori, "Coverage-guided learning-assisted grammar-based fuzzing," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019.

[6] S. Dechand, *The magic behind feedback-based fuzzing*, Last accessed 12th of May 2024, 2016.

[7] J. Rabcan, A. Vakhitova, and G. Kamalova, "Modern frameworks for web-application development," *Ġylym žane bìlìm*, vol. 3, no. 3 (72), pp. 33–40, 2023.

[8] I. P. Vuksanovic and B. Sudarevic, "Use of web application frameworks in the development of small applications," in *2011 Proceedings of the 34th International Convention MIPRO*, 2011.

[9] *Mvc framework introduction*, (https://www.geeksforgeeks.org/mvc-framework-introduction), Last accessed 8th of May 2024.

[10] "A web-based application for data collection and report generation using django," in *In ICT Systems and Sustainability: Proceedings of ICT4SD 2021, Volume 1*, 2022.

[11] *Python developers survey 2022 results*, (https://lp.jetbrains.com/python-developers-survey-2022/), Last accessed 10th of May 2024, 2022.

[12] "Comparative study on python web frameworks: Flask and django.," *Theseus - Interdisciplinary Studies Journal*, 2020.

[13] P. Kennedy, *What is werkzeug?* (https://testdriven.io/blog/what-is-werkzeug/), Last accessed 14th of May 2024, 2024.

[14] *Django documentation: Design philosophies*, Last accessed 10th of May 2024, (https://docs.djangoproject.com/en/5.0/misc/design-philosophies/).

[15] *Fastapi: Documentation*, (https://fastapi.tiangolo.com/tutorial), Last accessed 12th of May 2024.

[16] *Python documentation: Typing — support for type hints*, Last accessed 12th of May 2024, (https://docs.python.org/3/library/typing.html).

[17] KirstenS, *Cross site request forgery (csrf)*, (https://owasp.org/www-community/attacks/csrf), Last accessed 10th of May 2024.

[18] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983, ISSN: 0360-0300. DOI: 10.1145/289.291.

[19] *Postgres - about*, (https://www.postgresql.org/about), accessed 6th of December, 2024.

[20] *What is postgresql - aws*, (https://stackshare.io/postgresql), accessed 6th of December, 2024.

[21] *Companies that use postgresql*, (https://stackshare.io/postgresql), accessed 6th of December, 2024.

[22] *Postgresql - stackshare*, (https://stackshare.io/postgresql), accessed 6th of December, 2024.

[23] *What's the difference between docker images and containers? - aws*, (https://aws.amazon.com/compare/the-difference-between-docker-images-and-containers/), accessed 7th of December, 2024.

[24] *What is object storage? - google cloud*, (https://cloud.google.com/learn/what-is-object-storage), accessed 7th of December, 2024.

[25] *Xml.etree.elementtree — the elementtree xml api*, (https://docs.python.org/3/library/xml.etree.elementtree.html), accessed 12th of November, 2024.

[26] *Asyncio — asynchronous i/o*, (https://docs.python.org/3/library/asyncio.html), accessed 3th of December, 2024.

[27] *Asyncio.subprocess.pipe - asyncio documentation*, (https://docs.python.org/3/library/asyncio-subprocess.html#asyncio.subprocess.PIPE), accessed 3th of December, 2024.

[28] *Subprocess.create_new_process_group - asyncio documentation*, https://docs.python.org/3/library/subprocess.html#subprocess.CREATE_NEW_PROCESS_GROUP, accessed 4th of December, 2024.

[29] *Glob — unix style pathname pattern expansion*, (https://docs.python.org/3/library/glob.html), accessed 4th of December, 2024.

8 LIST OF FIGURES